

RTI Connex DDS

Core Libraries

User's Manual

Version 5.2.3



© 2016 Real-Time Innovations, Inc.
All rights reserved.
Printed in U.S.A. First printing.
April 2016.

Trademarks

Real-Time Innovations, RTI, NDDS, RTI Data Distribution Service, DataBus, Connex, Micro DDS, the RTI logo, IRTI and the phrase, "Your Systems. Working as one," are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

Third-Party Copyright Notices

Note: In this section, "the Software" refers to third-party software, portions of which are used in Connex DDS; "the Software" does not refer to Connex DDS.

This product implements the DCPS layer of the Data Distribution Service (DDS) specification version 1.2 and the DDS Interoperability Wire Protocol specification version 2.1, both of which are owned by the Object Management, Inc. Copyright 1997-2007 Object Management Group, Inc. The publication of these specifications can be found at the Catalog of OMG Data Distribution Service (DDS) Specifications. This documentation uses material from the OMG specification for the Data Distribution Service, section 7.

Reprinted with permission. Object Management, Inc. © OMG. 2005.

Portions of this product were developed using ANTLR (www.ANTLR.org). This product includes software developed by the University of California, Berkeley and its contributors.

Portions of this product were developed using AspectJ, which is distributed per the CPL license. AspectJ source code may be obtained from Eclipse. This product includes software developed by the University of California, Berkeley and its contributors.

Portions of this product were developed using MD5 from Aladdin Enterprises.

Portions of this product include software derived from Fmatch, (c) 1989, 1993, 1994 The Regents of the University of California. All rights reserved. The Regents and contributors provide this software "as is" without warranty.

Portions of this product were developed using EXPAT from Thai Open Source Software Center Ltd and Clark Cooper Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper Copyright (c) 2001, 2002 Expat maintainers. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Copyright © 1994–2013 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Technical Support

Real-Time Innovations, Inc.

232 E. Java Drive

Sunnyvale, CA 94089

Phone: (408) 990-7444

Email: support@rti.com

Website: <https://support.rti.com/>

Available Documentation

To get you up and running as quickly as possible, the *RTI® Connex™ DDS* documentation is divided into several parts.

- [RTI Connex DDS Core Libraries Getting Started Guide](#) — This document describes how to install Connex DDS. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. Developers should read this document first. Addendums cover:
 - [RTI Connex DDS Core Libraries Getting Started Guide Addendum for Android Systems](#)
 - [RTI Connex DDS Core Libraries Getting Started Guide Addendum for Database Setup](#)
 - [RTI Connex DDS Core Libraries Getting Started Guide Addendum for Embedded Systems](#)
 - [RTI Connex DDS Core Libraries Getting Started Guide Addendum for Extensible Types](#)
 - [RTI Connex DDS Core Libraries Getting Started Guide Addendum for iOS Systems](#)
- [RTI Connex DDS Core Libraries Whats New in 5.2.0](#) — This document describes changes and enhancements in the most recent major release of Connex DDS. Those upgrading from a previous version should read this document first. (Note: For what's new in maintenance and patch releases, see the [RTI Connex DDS Core Libraries Release Notes](#).)
- [RTI Connex DDS Core Libraries Release Notes](#) — This document describes system requirements, compatibility, what's fixed, and known issues.
- [RTI Connex DDS Core Libraries Platform Notes](#) — This document provides platform-specific information, including specific information required to build your applications using Connex DDS, such as compiler flags and libraries.

-
- RTI Connex DDS Core Libraries User's Manual — This document describes the features of the product and how to use them. It is organized around the structure of the Connex DDS APIs and certain common high-level tasks.
 - **API Reference HTML Documentation (README.html)** — This extensively cross-referenced documentation, available for all supported programming languages, is your in-depth reference to every operation and configuration parameter in the middleware. Even experienced Connex DDS developers will often consult this information.
 - **The Programming How To's provide a good place to begin learning the APIs.** These are hyper-linked code snippets to the full API documentation. From the **README.html** file, select one of the supported programming languages, then scroll down to the Programming How To's. Start by reviewing the Publication Example and Subscription Example, which provide step-by step examples of how to send and receive data with Connex DDS.

Many readers will also want to look at additional documentation available online. In particular, RTI recommends the following:

- Use the [RTI Customer Portal \(http://support.rti.com\)](http://support.rti.com) to download RTI software, access documentation and contact RTI Support. The RTI Customer Portal requires a username and password. You will receive this in the email confirming your purchase. If you do not have this email, please contact license@rti.com. Resetting your login password can be done directly at the RTI Customer Portal.
- The [RTI Community portal \(http://community.rti.com\)](http://community.rti.com) provides a wealth of knowledge to help you use Connex DDS, including:
 - Best Practices
 - Example code for specific features, as well as more complete use-case examples,
 - Solutions to common questions,
 - A glossary,
 - Downloads of experimental software,
 - And more.
- [Whitepapers and other articles are available from http://www.rti.com/resources.](http://www.rti.com/resources)

About this Document

Paths Mentioned in Documentation	xxxviii
Programming Language Conventions	xxxix
Traditional vs. Modern C++	xxxix
Extensions to the DDS Standard	xl
Environment Variables	xl
Additional Resources	xli
Part 1: Welcome to RTI Connext DDS	1
Chapter 1 Overview	
1.1 What is Connext DDS?	2
1.2 Network Communications Models	3
1.3 What is Middleware?	6
1.4 Features of Connext DDS	7
Chapter 2 Data-Centric Publish-Subscribe Communications	
2.1 What is DCPS?	10
2.1.1 DCPS for Real-Time Requirements	11
2.2 DDS Data Types, Topics, Keys, Instances, and Samples	12
2.3 Data Topics — What is the Data Called?	13
2.3.1 DDS Samples, Instances, and Keys	14
2.4 DataWriters/Publishers and DataReaders/Subscribers	15
2.5 DDS Domains and DomainParticipants	18
2.6 Quality of Service (QoS)	19
2.6.1 Controlling Behavior with Quality of Service (QoS) Policies	19
2.7 Application Discovery	20
Part 2: Core Concepts	22
Chapter 3 Data Types and DDS Data Samples	
3.1 Introduction to the Type System	25
3.1.1 Sequences	26
3.1.2 Strings and Wide Strings	28
3.1.3 Introduction to TypeCode	29
3.1.3.1 Sending TypeCodes on the Network	30
3.2 Built-in Data Types	30
3.2.1 Registering Built-in Types	30
3.2.2 Creating Topics for Built-in Types	31
3.2.2.1 Topic Creation Examples	31
3.2.3 String Built-in Type	33

3.2.3.1	Creating and Deleting Strings	33
3.2.3.2	String DataWriter	33
3.2.3.3	String DataReader	35
3.2.4	KeyedString Built-in Type	38
3.2.4.1	Creating and Deleting Keyed Strings	39
3.2.4.2	Keyed String DataWriter	40
3.2.4.3	Keyed String DataReader	43
3.2.5	Octets Built-in Type	46
3.2.5.1	Creating and Deleting Octets	47
3.2.5.2	Octets DataWriter	48
3.2.5.3	Octets DataReader	50
3.2.6	KeyedOctets Built-in Type	53
3.2.6.1	Creating and Deleting KeyedOctets	55
3.2.6.2	Keyed Octets DataWriter	55
3.2.6.3	Keyed Octets DataReader	59
3.2.7	Managing Memory for Built-in Types	62
3.2.7.1	Examples—Setting the Maximum Size for a String Programmatically	64
3.2.7.2	Unbounded Built-in Types	67
3.2.8	Type Codes for Built-in Types	68
3.3	Creating User Data Types with IDL	69
3.3.1	Variable-Length Types	70
3.3.1.1	Sequences	71
3.3.1.2	Strings and Wide Strings	71
3.3.2	Value Types	72
3.3.3	Type Codes	73
3.3.4	Translations for IDL Types	73
3.3.5	Escaped Identifiers	111
3.3.6	Namespaces In IDL Files	111
3.3.7	Referring to Other IDL Files	114
3.3.8	Preprocessor Directives	115
3.3.9	Using Custom Directives	115
3.3.9.1	The @key Directive	116
3.3.9.2	The @copy and Related Directives	117
3.3.9.3	The @resolve-name Directive	119
3.3.9.4	The @top-level Directive	120
3.4	Creating User Data Types with Extensible Markup Language (XML)	121

3.4.1 Primitive Types	128
3.5 Creating User Data Types with XML Schemas (XSD)	128
3.6 Using RTI Code Generator (rtiddsgen)	138
3.7 Using Generated Types without Connnext DDS (Standalone)	139
3.7.1 Using Standalone Types in C	139
3.7.2 Using Standalone Types in C++	140
3.7.3 Standalone Types in Java	140
3.8 Interacting Dynamically with User Data Types	141
3.8.1 Type Schemas and TypeCode Objects	141
3.8.2 Defining New Types	141
3.8.3 Sending Only a Few Fields	143
3.8.4 Sending Type Codes on the Network	143
3.8.4.1 Type Codes for Built-in Types	143
3.9 Working with DDS Data Samples	145
3.9.1 Objects of Concrete Types	145
3.9.2 Objects of Dynamically Defined Types	147
3.9.3 Serializing and Deserializing Data Samples	149
3.9.4 Accessing the Discriminator Value in a Union	150
Chapter 4 DDS Entities	
4.1 Common Operations for All DDS Entities	152
4.1.1 Creating and Deleting DDS Entities	153
4.1.2 Enabling DDS Entities	154
4.1.2.1 Rules for Calling enable()	155
4.1.3 Getting an Entity's Instance Handle	157
4.1.4 Getting Status and Status Changes	157
4.1.5 Getting and Setting Listeners	158
4.1.6 Getting the StatusCondition	158
4.1.7 Getting, Setting, and Comparing QoS Policies	158
4.1.7.1 Changing the QoS Defaults Used to Create DDS Entities: set_default_*_qos()	160
4.1.7.2 Setting QoS During Entity Creation	160
4.1.7.3 Changing the QoS for an Existing Entity	161
4.1.7.4 Default QoS Values	162
4.2 QoS Policies	162
4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property	167
4.2.2 Special QoS Policy Handling Considerations for C	168
4.3 Statures	169

4.3.1	Types of Communication Status	170
4.3.1.1	Changes in Plain Communication Status	173
4.3.1.2	Changes in Read Communication Status	174
4.3.2	Special Status-Handling Considerations for C	176
4.4	Listeners	177
4.4.1	Types of Listeners	177
4.4.2	Creating and Deleting Listeners	179
4.4.3	Special Considerations for Listeners in C	180
4.4.4	Hierarchical Processing of Listeners	180
4.4.4.1	Processing Read Communication Statuses	181
4.4.5	Operations Allowed within Listener Callbacks	182
4.5	Exclusive Areas (EAs)	182
4.5.1	Restricted Operations in Listener Callbacks	185
4.6	Conditions and WaitSets	187
4.6.1	Creating and Deleting WaitSets	188
4.6.2	WaitSet Operations	189
4.6.3	Waiting for Conditions	190
4.6.3.1	How WaitSets Block	191
4.6.4	Processing Triggered Conditions—What to do when Wait() Returns	192
4.6.5	Conditions and WaitSet Example	193
4.6.6	GuardConditions	194
4.6.7	ReadConditions and QueryConditions	195
4.6.7.1	How ReadConditions are Triggered	196
4.6.7.2	QueryConditions	197
4.6.8	StatusConditions	197
4.6.8.1	How StatusConditions are Triggered	199
4.6.9	Using Both Listeners and WaitSets	199

Chapter 5 Topics

5.1	Topics	200
5.1.1	Creating Topics	202
5.1.2	Deleting Topics	204
5.1.3	Setting Topic QoS Policies	204
5.1.3.1	Configuring QoS Settings when the Topic is Created	206
5.1.3.2	Comparing QoS Values	207
5.1.3.3	Changing QoS Settings After the Topic Has Been Created	207
5.1.4	Copying QoS From a Topic to a DataWriter or DataReader	208

5.1.5	Setting Up TopicListeners	208
5.1.6	Navigating Relationships Among Entities	209
5.1.6.1	Finding a Topic's DomainParticipant	209
5.1.6.2	Retrieving a Topic's Name or DDS Type Name	209
5.2	Topic QosPolicies	209
5.2.1	TOPIC_DATA QosPolicy	209
5.2.1.1	Example	210
5.2.1.2	Properties	210
5.2.1.3	Related QosPolicies	211
5.2.1.4	Applicable DDS Entities	211
5.2.1.5	System Resource Considerations	211
5.3	Status Indicator for Topics	211
5.3.1	INCONSISTENT_TOPIC Status	211
5.4	ContentFilteredTopics	212
5.4.1	Overview	212
5.4.2	Where Filtering is Applied—Publishing vs. Subscribing Side	213
5.4.3	Creating ContentFilteredTopics	214
5.4.3.1	Creating ContentFilteredTopics for Built-in DDS Types	217
5.4.4	Deleting ContentFilteredTopics	218
5.4.5	Using a ContentFilteredTopic	219
5.4.5.1	Getting the Current Expression Parameters	219
5.4.5.2	Setting an Expression's Filter and Parameters	220
5.4.5.3	Appending a String to an Expression Parameter	220
5.4.5.4	Removing a String from an Expression Parameter	221
5.4.5.5	Getting the Filter Expression	221
5.4.5.6	Getting the Related Topic	221
5.4.5.7	'Narrowing' a ContentFilteredTopic to a TopicDescription	222
5.4.6	SQL Filter Expression Notation	222
5.4.6.1	Example SQL Filter Expressions	222
5.4.6.2	SQL Grammar	224
5.4.6.3	Token Expressions	225
5.4.6.4	Type Compatibility in the Predicate	227
5.4.6.5	SQL Extension: Regular Expression Matching	228
5.4.6.6	Composite Members	229
5.4.6.7	Strings	229
5.4.6.8	Enumerations	230

5.4.6.9 Pointers	230
5.4.6.10 Arrays	230
5.4.6.11 Sequences	231
5.4.7 STRINGMATCH Filter Expression Notation	231
5.4.7.1 Example STRINGMATCH Filter Expressions	232
5.4.7.2 STRINGMATCH Filter Expression Parameters	232
5.4.8 Custom Content Filters	233
5.4.8.1 Filtering on the Writer Side with Custom Filters	233
5.4.8.2 Registering a Custom Filter	234
5.4.8.3 Unregistering a Custom Filter	236
5.4.8.4 Retrieving a ContentFilter	237
5.4.8.5 Compile Function	237
5.4.8.6 Evaluate Function	238
5.4.8.7 Finalize Function	239
5.4.8.8 Writer Attach Function	239
5.4.8.9 Writer Detach Function	239
5.4.8.10 Writer Compile Function	239
5.4.8.11 Writer Evaluate Function	240
5.4.8.12 Writer Return Loan Function	241
5.4.8.13 Writer Finalize Function	241
Chapter 6 Sending Data	
6.1 Preview: Steps to Sending Data	242
6.2 Publishers	243
6.2.1 Creating Publishers Explicitly vs. Implicitly	248
6.2.2 Creating Publishers	249
6.2.3 Deleting Publishers	250
6.2.3.1 Deleting Contained DataWriters	251
6.2.4 Setting Publisher QoS Policies	251
6.2.4.1 Configuring QoS Settings when the Publisher is Created	252
6.2.4.2 Comparing QoS Values	254
6.2.4.3 Changing QoS Settings After the Publisher Has Been Created	254
6.2.4.4 Getting and Setting the Publisher's Default QoS Profile and Library	255
6.2.4.5 Getting and Setting Default QoS for DataWriters	256
6.2.4.6 Other Publisher QoS-Related Operations	257
6.2.5 Setting Up PublisherListeners	257
6.2.6 Finding a Publisher's Related DDS Entities	259

6.2.7	Waiting for Acknowledgments in a Publisher	260
6.2.8	Statuses for Publishers	260
6.2.9	Suspending and Resuming Publications	261
6.3	DataWriters	261
6.3.1	Creating DataWriters	266
6.3.2	Getting All DataWriters	268
6.3.3	Deleting DataWriters	268
6.3.3.1	Special Instructions for deleting DataWriters if you are using the ‘Timestamp’ APIs and BY_SOURCE_TIMESTAMP Destination Order:	268
6.3.4	Setting Up DataWriterListeners	269
6.3.5	Checking DataWriter Status	270
6.3.6	Statuses for DataWriters	271
6.3.6.1	APPLICATION_ACKNOWLEDGMENT_STATUS	272
6.3.6.2	DATA_WRITER_CACHE_STATUS	272
6.3.6.3	DATA_WRITER_PROTOCOL_STATUS	273
6.3.6.4	LIVELINESS_LOST Status	276
6.3.6.5	OFFERED_DEADLINE_MISSED Status	277
6.3.6.6	OFFERED_INCOMPATIBLE_QOS Status	277
6.3.6.7	PUBLICATION_MATCHED Status	278
6.3.6.8	RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension)	279
6.3.6.9	RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension)	281
6.3.7	Using a Type-Specific DataWriter (FooDataWriter)	281
6.3.8	Writing Data	283
6.3.8.1	Blocking During a write()	286
6.3.9	Flushing Batches of DDS Data Samples	287
6.3.10	Writing Coherent Sets of DDS Data Samples	287
6.3.11	Waiting for Acknowledgments in a DataWriter	288
6.3.12	Application Acknowledgment	288
6.3.12.1	Application Acknowledgment Kinds	289
6.3.12.2	Explicitly Acknowledging a Single DDS Sample (C++)	290
6.3.12.3	Explicitly Acknowledging All DDS samples (C++)	290
6.3.12.4	Notification of Delivery with Application Acknowledgment	290
6.3.12.5	Application-Level Acknowledgment Protocol	291
6.3.12.6	Periodic and Non-Periodic AppAck Messages	293
6.3.12.7	Application Acknowledgment and Persistence Service	293
6.3.12.8	Application Acknowledgment and Routing Service	294

6.3.13	Required Subscriptions	294
6.3.13.1	Named, Required and Durable Subscriptions	295
6.3.13.2	Durability QoS and Required Subscriptions	295
6.3.13.3	Required Subscriptions Configuration	296
6.3.14	Managing Data Instances (Working with Keyed Data Types)	296
6.3.14.1	Registering and Unregistering Instances	297
6.3.14.2	Disposing of Data	299
6.3.14.3	Looking Up an Instance Handle	299
6.3.14.4	Getting the Key Value for an Instance	299
6.3.15	Setting DataWriter QoS Policies	300
6.3.15.1	Configuring QoS Settings when the DataWriter is Created	303
6.3.15.2	Comparing QoS Values	305
6.3.15.3	Changing QoS Settings After the DataWriter Has Been Created	305
6.3.15.4	Using a Topic's QoS to Initialize a DataWriter's QoS	306
6.3.16	Navigating Relationships Among DDS Entities	309
6.3.16.1	Finding Matching Subscriptions	309
6.3.16.2	Finding the Matching Subscription's ParticipantBuiltinTopicData	311
6.3.16.3	Finding Related DDS Entities	311
6.3.17	Asserting Liveliness	311
6.3.18	Turbo Mode and Automatic Throttling for DataWriter Performance—Experimental Features	312
6.4	Publisher/Subscriber QoS Policies	312
6.4.1	ASYNCHRONOUS_PUBLISHER QoS Policy (DDS Extension)	313
6.4.1.1	Properties	314
6.4.1.2	Related QoS Policies	314
6.4.1.3	Applicable DDS Entities	314
6.4.1.4	System Resource Considerations	315
6.4.2	ENTITYFACTORY QoS Policy	315
6.4.2.1	Example	316
6.4.2.2	Properties	317
6.4.2.3	Related QoS Policies	317
6.4.2.4	Applicable DDS Entities	317
6.4.2.5	System Resource Considerations	317
6.4.3	EXCLUSIVE_AREA QoS Policy (DDS Extension)	318
6.4.3.1	Example	319
6.4.3.2	Properties	320
6.4.3.3	Related QoS Policies	320

6.4.3.4	Applicable DDS Entities	320
6.4.3.5	System Resource Considerations	320
6.4.4	GROUP_DATA QosPolicy	320
6.4.4.1	Example	321
6.4.4.2	Properties	322
6.4.4.3	Related QosPolicies	322
6.4.4.4	Applicable DDS Entities	323
6.4.4.5	System Resource Considerations	323
6.4.5	PARTITION QosPolicy	323
6.4.5.1	Rules for PARTITION Matching	325
6.4.5.2	Pattern Matching for PARTITION Names	325
6.4.5.3	Example	326
6.4.5.4	Properties	329
6.4.5.5	Related QosPolicies	329
6.4.5.6	Applicable DDS Entities	329
6.4.5.7	System Resource Considerations	329
6.4.6	PRESENTATION QosPolicy	330
6.4.6.1	Coherent Access	331
6.4.6.2	Ordered Access	332
6.4.6.3	Example	333
6.4.6.4	Properties	334
6.4.6.5	Related QosPolicies	335
6.4.6.6	Applicable DDS Entities	336
6.4.6.7	System Resource Considerations	336
6.5	DataWriter QosPolicies	336
6.5.1	AVAILABILITY QosPolicy (DDS Extension)	337
6.5.1.1	Availability QoS Policy and Collaborative DataWriters	338
6.5.1.2	Availability QoS Policy and Required Subscriptions	339
6.5.1.3	Properties	340
6.5.1.4	Related QosPolicies	340
6.5.1.5	Applicable DDS Entities	341
6.5.1.6	System Resource Considerations	341
6.5.2	BATCH QosPolicy (DDS Extension)	341
6.5.2.1	Synchronous and Asynchronous Flushing	343
6.5.2.2	Batching vs. Coalescing	344
6.5.2.3	Batching and ContentFilteredTopics	344

6.5.2.4 Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature	344
6.5.2.5 Performance Considerations	345
6.5.2.6 Maximum Transport Datagram Size	345
6.5.2.7 Properties	345
6.5.2.8 Related QosPolicies	346
6.5.2.9 Applicable DDS Entities	346
6.5.2.10 System Resource Considerations	346
6.5.3 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)	347
6.5.3.1 High and Low Watermarks	352
6.5.3.2 Normal, Fast, and Late-Joiner Heartbeat Periods	353
6.5.3.3 Disabling Positive Acknowledgements	354
6.5.3.4 Configuring the Send Window Size	355
6.5.3.5 Propagating Serialized Keys with Disposed-Instance Notifications	356
6.5.3.6 Virtual Heartbeats	357
6.5.3.7 Resending Over Multicast	357
6.5.3.8 Example	358
6.5.3.9 Properties	358
6.5.3.10 Related QosPolicies	359
6.5.3.11 Applicable DDS Entities	359
6.5.3.12 System Resource Considerations	359
6.5.4 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension)	359
6.5.4.1 Example	362
6.5.4.2 Properties	362
6.5.4.3 Related QosPolicies	363
6.5.4.4 Applicable DDS Entities	363
6.5.4.5 System Resource Considerations	363
6.5.5 DEADLINE QosPolicy	363
6.5.5.1 Example	364
6.5.5.2 Properties	365
6.5.5.3 Related QosPolicies	365
6.5.5.4 Applicable DDS Entities	365
6.5.5.5 System Resource Considerations	365
6.5.6 DESTINATION_ORDER QosPolicy	365
6.5.6.1 Properties	367
6.5.6.2 Related QosPolicies	368

6.5.6.3	Applicable DDS Entities	368
6.5.6.4	System Resource Considerations	368
6.5.7	DURABILITY QosPolicy	368
6.5.7.1	Example	370
6.5.7.2	Properties	371
6.5.7.3	Related QosPolicies	371
6.5.7.4	Applicable Entities	371
6.5.7.5	System Resource Considerations	372
6.5.8	DURABILITY SERVICE QosPolicy	372
6.5.8.1	Properties	374
6.5.8.2	Related QosPolicies	374
6.5.8.3	Applicable Entities	374
6.5.8.4	System Resource Considerations	374
6.5.9	ENTITY_NAME QosPolicy (DDS Extension)	374
6.5.9.1	Properties	375
6.5.9.2	Related QosPolicies	375
6.5.9.3	Applicable Entities	375
6.5.9.4	System Resource Considerations	376
6.5.10	HISTORY QosPolicy	376
6.5.10.1	Example	379
6.5.10.2	Properties	379
6.5.10.3	Related QosPolicies	380
6.5.10.4	Applicable Entities	380
6.5.10.5	System Resource Considerations	380
6.5.11	LATENCYBUDGET QoS Policy	380
6.5.11.1	Applicable Entities	381
6.5.12	LIFESPAN QoS Policy	381
6.5.12.1	Properties	382
6.5.12.2	Related QoS Policies	382
6.5.12.3	Applicable Entities	382
6.5.12.4	System Resource Considerations	382
6.5.13	LIVELINESS QosPolicy	382
6.5.13.1	Example	385
6.5.13.2	Properties	385
6.5.13.3	Related QosPolicies	386
6.5.13.4	Applicable Entities	386

6.5.13.5 System Resource Considerations	386
6.5.14 MULTI_CHANNEL QosPolicy (DDS Extension)	386
6.5.14.1 Example	389
6.5.14.2 Properties	389
6.5.14.3 Related Qos Policies	389
6.5.14.4 Applicable Entities	389
6.5.14.5 System Resource Considerations	389
6.5.15 OWNERSHIP QosPolicy	389
6.5.15.1 How Connex DDS Selects which DataWriter is the Exclusive Owner	391
6.5.15.2 Example	391
6.5.15.3 Properties	392
6.5.15.4 Related QosPolicies	392
6.5.15.5 Applicable Entities	393
6.5.15.6 System Resource Considerations	393
6.5.16 OWNERSHIP_STRENGTH QosPolicy	393
6.5.16.1 Example	393
6.5.16.2 Properties	393
6.5.16.3 Related QosPolicies	394
6.5.16.4 Applicable Entities	394
6.5.16.5 System Resource Considerations	394
6.5.17 PROPERTY QosPolicy (DDS Extension)	394
6.5.17.1 Properties	397
6.5.17.2 Related QosPolicies	397
6.5.17.3 Applicable Entities	397
6.5.17.4 System Resource Considerations	397
6.5.18 PUBLISH_MODE QosPolicy (DDS Extension)	397
6.5.18.1 Properties	399
6.5.18.2 Related QosPolicies	399
6.5.18.3 Applicable Entities	400
6.5.18.4 System Resource Considerations	400
6.5.19 RELIABILITY QosPolicy	400
6.5.19.1 Example	403
6.5.19.2 Properties	403
6.5.19.3 Related QosPolicies	404
6.5.19.4 Applicable Entities	404
6.5.19.5 System Resource Considerations	404

6.5.20 RESOURCE_LIMITS QosPolicy	405
6.5.20.1 Configuring Resource Limits for Asynchronous DataWriters	406
6.5.20.2 Configuring DataWriter Instance Replacement	407
6.5.20.3 Example	407
6.5.20.4 Properties	408
6.5.20.5 Related QosPolicies	408
6.5.20.6 Applicable Entities	408
6.5.20.7 System Resource Considerations	408
6.5.21 SERVICE QosPolicy (DDS Extension)	408
6.5.21.1 Properties	409
6.5.21.2 Related QosPolicies	409
6.5.21.3 Applicable Entities	409
6.5.21.4 System Resource Considerations	409
6.5.22 TRANSPORT_PRIORITY QosPolicy	409
6.5.22.1 Example	410
6.5.22.2 Properties	410
6.5.22.3 Related QosPolicies	411
6.5.22.4 Applicable Entities	411
6.5.22.5 System Resource Considerations	411
6.5.23 TRANSPORT_SELECTION QosPolicy (DDS Extension)	411
6.5.23.1 Example	412
6.5.23.2 Properties	412
6.5.23.3 Related QosPolicies	412
6.5.23.4 Applicable Entities	412
6.5.23.5 System Resource Considerations	412
6.5.24 TRANSPORT_UNICAST QosPolicy (DDS Extension)	412
6.5.24.1 Example	415
6.5.24.2 Properties	415
6.5.24.3 Related QosPolicies	415
6.5.24.4 Applicable Entities	415
6.5.24.5 System Resource Considerations	415
6.5.25 TYPESUPPORT QosPolicy (DDS Extension)	416
6.5.25.1 Properties	416
6.5.25.2 Related QoS Policies	417
6.5.25.3 Applicable Entities	417
6.5.25.4 System Resource Considerations	417

6.5.26	USER_DATA QoSPolicy	417
6.5.26.1	Example	418
6.5.26.2	Properties	418
6.5.26.3	Related QoS Policies	418
6.5.26.4	Applicable Entities	419
6.5.26.5	System Resource Considerations	419
6.5.27	WRITER_DATA_LIFECYCLE QoS Policy	419
6.5.27.1	Properties	421
6.5.27.2	Related QoS Policies	422
6.5.27.3	Applicable Entities	422
6.5.27.4	System Resource Considerations	422
6.6	FlowControllers (DDS Extension)	422
6.6.1	Flow Controller Scheduling Policies	424
6.6.2	Managing Fast DataWriters When Using a FlowController	426
6.6.3	Token Bucket Properties	426
6.6.3.1	max_tokens	427
6.6.3.2	tokens_added_per_period	427
6.6.3.3	tokens_leaked_per_period	427
6.6.3.4	period	427
6.6.3.5	bytes_per_token	428
6.6.4	Prioritized DDS Samples	428
6.6.4.1	Designating Priorities	429
6.6.4.2	Priority-Based Filtering	430
6.6.5	Creating and Configuring Custom FlowControllers with Property QoS	431
6.6.5.1	Example	432
6.6.6	Creating and Deleting FlowControllers	433
6.6.7	Getting/Setting Default FlowController Properties	434
6.6.8	Getting/Setting Properties for a Specific FlowController	435
6.6.9	Adding an External Trigger	435
6.6.10	Other FlowController Operations	435
Chapter 7 Receiving Data		
7.1	Preview: Steps to Receiving Data	437
7.2	Subscribers	440
7.2.1	Creating Subscribers Explicitly vs. Implicitly	444
7.2.2	Creating Subscribers	445
7.2.3	Deleting Subscribers	446

7.2.3.1	Deleting Contained DataReaders	447
7.2.4	Setting Subscriber QoS Policies	447
7.2.4.1	Configuring QoS Settings when the Subscriber is Created	448
7.2.4.2	Comparing QoS Values	450
7.2.4.3	Changing QoS Settings After Subscriber Has Been Created	450
7.2.4.4	Getting and Settings Subscriber's Default QoS Profile and Library	451
7.2.4.5	Getting and Setting Default QoS for DataReaders	452
7.2.4.6	Subscriber QoS-Related Operations	453
7.2.5	Beginning and Ending Group-Ordered Access	453
7.2.6	Setting Up SubscriberListeners	454
7.2.7	Getting DataReaders with Specific DDS Samples	456
7.2.8	Finding a Subscriber's Related Entities	457
7.2.9	Statuses for Subscribers	458
7.2.9.1	DATA_ON_READERS Status	458
7.3	DataReaders	459
7.3.1	Creating DataReaders	463
7.3.2	Getting All DataReaders	465
7.3.3	Deleting DataReaders	466
7.3.3.1	Deleting Contained ReadConditions	466
7.3.4	Setting Up DataReaderListeners	466
7.3.5	Checking DataReader Status and StatusConditions	468
7.3.6	Waiting for Historical Data	469
7.3.7	Statuses for DataReaders	470
7.3.7.1	DATA_AVAILABLE Status	471
7.3.7.2	DATA_READER_CACHE_STATUS	471
7.3.7.3	DATA_READER_PROTOCOL_STATUS	472
7.3.7.4	LIVELINESS_CHANGED Status	475
7.3.7.5	REQUESTED_DEADLINE_MISSED Status	476
7.3.7.6	REQUESTED_INCOMPATIBLE_QOS Status	477
7.3.7.7	SAMPLE_LOST Status	478
7.3.7.8	SAMPLE_REJECTED Status	479
7.3.7.9	SUBSCRIPTION_MATCHED Status	482
7.3.8	Setting DataReader QoS Policies	482
7.3.8.1	Configuring QoS Settings when the DataReader is Created	485
7.3.8.2	Comparing QoS Values	487
7.3.8.3	Changing QoS Settings After DataReader Has Been Created	487

7.3.8.4	Using a Topic's QoS to Initialize a DataWriter's QoS	488
7.3.9	Navigating Relationships Among Entities	489
7.3.9.1	Finding Matching Publications	489
7.3.9.2	Finding the Matching Publication's ParticipantBuiltinTopicData	490
7.3.9.3	Finding a DataReader's Related Entities	490
7.3.9.4	Looking Up an Instance Handle	490
7.3.9.5	Getting the Key Value for an Instance	491
7.4	Using DataReaders to Access Data (Read & Take)	491
7.4.1	Using a Type-Specific DataReader (FooDataReader)	491
7.4.2	Loaning and Returning Data and SampleInfo Sequences	492
7.4.2.1	C, Traditional C++, Java and .NET	492
7.4.2.2	Modern C++	493
7.4.3	Accessing DDS Data Samples with Read or Take	493
7.4.3.1	Read vs. Take	494
7.4.3.2	General Patterns for Accessing Data	496
7.4.3.3	read_next_sample and take_next_sample	497
7.4.3.4	read_instance and take_instance	497
7.4.3.5	read_next_instance and take_next_instance	498
7.4.3.6	read_w_condition and take_w_condition	500
7.4.3.7	read_instance_w_condition and take_instance_w_condition	500
7.4.3.8	read_next_instance_w_condition and take_next_instance_w_condition	501
7.4.3.9	The select() API (Modern C++)	501
7.4.4	Acknowledging DDS Samples	502
7.4.5	The Sequence Data Structure	502
7.4.6	The SampleInfo Structure	504
7.4.6.1	Reception Timestamp	506
7.4.6.2	Sample States	506
7.4.6.3	View States	506
7.4.6.4	Instance States	507
7.4.6.5	Generation Counts and Ranks	508
7.4.6.6	Valid Data Flag	510
7.5	Subscriber QosPolicies	510
7.6	DataReader QosPolicies	510
7.6.1	DATA_READER_PROTOCOL QosPolicy (DDS Extension)	511
7.6.1.1	Receive Window Size	515
7.6.1.2	Round-Trip Time For Filtering Redundant NACKs	516

7.6.1.3	Example	516
7.6.1.4	Properties	517
7.6.1.5	Related QosPolicies	517
7.6.1.6	Applicable Dds Entities	517
7.6.1.7	System Resource Considerations	517
7.6.2	DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension)	517
7.6.2.1	max_total_instances and max_instances	522
7.6.2.2	Example	522
7.6.2.3	Properties	523
7.6.2.4	Related QosPolicies	523
7.6.2.5	Applicable Dds Entities	523
7.6.2.6	System Resource Considerations	523
7.6.3	READER_DATA_LIFECYCLE QoS Policy	523
7.6.3.1	Properties	525
7.6.3.2	Related QoS Policies	525
7.6.3.3	Applicable Dds Entities	525
7.6.3.4	System Resource Considerations	526
7.6.4	TIME_BASED_FILTER QosPolicy	526
7.6.4.1	Example	528
7.6.4.2	Properties	528
7.6.4.3	Related QosPolicies	528
7.6.4.4	Applicable Dds Entities	528
7.6.4.5	System Resource Considerations	528
7.6.5	TRANSPORT_MULTICAST QosPolicy (DDS Extension)	529
7.6.5.1	Example	531
7.6.5.2	Properties	531
7.6.5.3	Related QosPolicies	531
7.6.5.4	Applicable DDS Entities	532
7.6.5.5	System Resource Considerations	532
7.6.6	TYPE_CONSISTENCY_ENFORCEMENT QosPolicy	532
7.6.6.1	Properties	534
7.6.6.2	Related QoS Policies	534
7.6.6.3	Applicable Entities	534
7.6.6.4	System Resource Considerations	535
Chapter 8 Working with DDS Domains		
8.1	Fundamentals of DDS Domains and DomainParticipants	536

8.2 DomainParticipantFactory	539
8.2.1 Setting DomainParticipantFactory QoS Policies	543
8.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library	544
8.2.2 Getting and Setting Default QoS for DomainParticipants	545
8.2.3 Freeing Resources Used by the DomainParticipantFactory	546
8.2.4 Looking Up DomainParticipants	546
8.2.5 Getting QoS Values from a QoS Profile	547
8.3 DomainParticipants	547
8.3.1 Creating a DomainParticipant	556
8.3.2 Deleting DomainParticipants	558
8.3.3 Deleting Contained Entities	559
8.3.4 Choosing a Domain ID and Creating Multiple DDS Domains	559
8.3.5 Setting Up DomainParticipantListeners	560
8.3.6 Setting DomainParticipant QoS Policies	562
8.3.6.1 Configuring QoS Settings when DomainParticipant is Created	564
8.3.6.2 Comparing QoS Values	565
8.3.6.3 Changing QoS Settings After DomainParticipant Has Been Created	566
8.3.6.4 Getting and Setting DomainParticipant's Default QoS Profile and Library	567
8.3.6.5 Getting and Setting Default QoS for Child Entities	568
8.3.7 Looking up Topic Descriptions	568
8.3.8 Finding a Topic	569
8.3.9 Getting the Implicit Publisher or Subscriber	569
8.3.10 Asserting Liveliness	570
8.3.11 Learning about Discovered DomainParticipants	571
8.3.12 Learning about Discovered Topics	571
8.3.13 Other DomainParticipant Operations	571
8.3.13.1 Verifying Entity Containment	571
8.3.13.2 Getting the Current Time	571
8.3.13.3 Getting All Publishers and Subscribers	572
8.4 DomainParticipantFactory QoS Policies	572
8.4.1 LOGGING QoS Policy (DDS Extension)	572
8.4.1.1 Example	572
8.4.1.2 Properties	573
8.4.1.3 Related QoS Policies	573
8.4.1.4 Applicable DDS Entities	573
8.4.1.5 System Resource Considerations	573

8.4.2	PROFILE QosPolicy (DDS Extension)	573
8.4.2.1	Example	574
8.4.2.2	Properties	575
8.4.2.3	Related QosPolicies	575
8.4.2.4	Applicable Entities	575
8.4.2.5	System Resource Considerations	575
8.4.3	SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension)	575
8.4.3.1	Example	576
8.4.3.2	Properties	576
8.4.3.3	Related QoS Policies	577
8.4.3.4	Applicable Dds Entities	577
8.4.3.5	System Resource Considerations	577
8.5	DomainParticipant QosPolicies	577
8.5.1	DATABASE QosPolicy (DDS Extension)	577
8.5.1.1	Example	579
8.5.1.2	Properties	579
8.5.1.3	Related QosPolicies	579
8.5.1.4	Applicable Dds Entities	580
8.5.1.5	System Resource Considerations	580
8.5.2	DISCOVERY QosPolicy (DDS Extension)	580
8.5.2.1	Transports Used for Discovery	581
8.5.2.2	Setting the ‘Initial Peers’ List	581
8.5.2.3	Adding and Removing Peers List Entries	581
8.5.2.4	Configuring Multicast Receive Addresses	582
8.5.2.5	Meta-Traffic Transport Priority	583
8.5.2.6	Controlling Acceptance of Unknown Peers	583
8.5.2.7	Example	583
8.5.2.8	Properties	584
8.5.2.9	Related QosPolicies	584
8.5.2.10	Applicable Entities	584
8.5.2.11	System Resource Considerations	584
8.5.3	DISCOVERY_CONFIG QosPolicy (DDS Extension)	585
8.5.3.1	Resource Limits for Builtin-Topic DataReaders	589
8.5.3.2	Controlling Purging of Remote Participants	591
8.5.3.3	Controlling the Reliable Protocol Used by Builtin-Topic DataWriters/DataReaders	592
8.5.3.4	Example	592

8.5.3.5	Properties	593
8.5.3.6	Related QosPolicies	593
8.5.3.7	Applicable Dds Entities	593
8.5.3.8	System Resource Considerations	593
8.5.4	DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)	593
8.5.4.1	Configuring Resource Limits for Asynchronous DataWriters	600
8.5.4.2	Configuring Memory Allocation	600
8.5.4.3	Example	601
8.5.4.4	Properties	602
8.5.4.5	Related QosPolicies	602
8.5.4.6	Applicable DDS Entities	602
8.5.4.7	System Resource Considerations	602
8.5.5	EVENT QosPolicy (DDS Extension)	602
8.5.5.1	Example	603
8.5.5.2	Properties	604
8.5.5.3	Related QosPolicies	604
8.5.5.4	Applicable DDS Entities	604
8.5.5.5	System Resource Considerations	604
8.5.6	RECEIVER_POOL QosPolicy (DDS Extension)	604
8.5.6.1	Example	606
8.5.6.2	Properties	606
8.5.6.3	Related QosPolicies	606
8.5.6.4	Applicable Dds Entities	606
8.5.6.5	System Resource Considerations	606
8.5.7	TRANSPORT_BUILTIN QosPolicy (DDS Extension)	606
8.5.7.1	Example	607
8.5.7.2	Properties	607
8.5.7.3	Related QosPolicies	607
8.5.7.4	Applicable DDS Entities	607
8.5.7.5	System Resource Considerations	608
8.5.8	TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension)	608
8.5.8.1	Formatting Rules for Addresses	609
8.5.8.2	Example	610
8.5.8.3	Properties	610
8.5.8.4	Related QosPolicies	610
8.5.8.5	Applicable DDS Entities	610

8.5.8.6	System Resource Considerations	610
8.5.9	WIRE_PROTOCOL QosPolicy (DDS Extension)	610
8.5.9.1	Choosing Participant IDs	611
8.5.9.2	Host, App, and Instance IDs	613
8.5.9.3	Ports Used for Discovery	613
8.5.9.4	Controlling How the GUID is Set (rtps_auto_id_kind)	614
8.5.9.5	Example	618
8.5.9.6	Properties	618
8.5.9.7	Related QosPolicies	619
8.5.9.8	Applicable DDS Entities	619
8.5.9.9	System Resource Considerations	619
8.6	Clock Selection	619
8.6.1	Available Clocks	619
8.6.2	Clock Selection Strategy	619
8.7	System Properties	620
Chapter 9 Building Applications		
9.1	Running on a Computer Not Connected to a Network	623
9.2	Connex DDS Header Files — All Architectures	623
9.3	UNIX-Based Platforms	624
9.3.1	Required Libraries	625
9.3.2	Compiler Flags	625
9.4	Windows Platforms	625
9.4.1	Using Visual Studio	626
9.5	Java Platforms	627
9.5.1	Java Libraries	627
9.5.2	Native Libraries	627
Part 3: Advanced Concepts		628
Chapter 10 Reliable Communications		
10.1	Sending Data Reliably	629
10.1.1	Best-effort Delivery Model	629
10.1.2	Reliable Delivery Model	630
10.2	Overview of the Reliable Protocol	631
10.3	Using QosPolicies to Tune the Reliable Protocol	635
10.3.1	Enabling Reliability	637
10.3.1.1	Blocking until the Send Queue Has Space Available	637
10.3.2	Tuning Queue Sizes and Other Resource Limits	638

10.3.2.1	Understanding the Send Queue and Setting its Size	639
10.3.2.2	Understanding the Receive Queue and Setting Its Size	642
10.3.3	Controlling Queue Depth with the History QosPolicy	644
10.3.4	Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy	645
10.3.4.1	How Often Heartbeats are Resent (heartbeat_period)	645
10.3.4.2	How Often Piggyback Heartbeats are Sent (heartbeats_per_max_samples)	647
10.3.4.3	Controlling Packet Size for Resent DDS Samples (max_bytes_per_nack_response)	649
10.3.4.4	Controlling How Many Times Heartbeats are Resent (max_heartbeat_retries)	650
10.3.4.5	Treating Non-Progressing Readers as Inactive Readers (inactivate_nonprogressing_readers)	650
10.3.4.6	Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay)	651
10.3.4.7	Disabling Positive Acknowledgements (disable_positive_acks_min_sample_keep_duration)	652
10.3.5	Avoiding Message Storms with DataReaderProtocol QosPolicy	653
10.3.6	Resending DDS Samples to Late-Joiners with the Durability QosPolicy	653
10.3.7	Use Cases	654
10.3.7.1	Importance of Relative Thread Priorities	654
10.3.7.2	Aperiodic Use Case: One-at-a-Time	655
10.3.7.3	Aperiodic, Bursty	659
10.3.7.4	Periodic	664
10.4	Auto Throttling for DataWriter Performance—Experimental Feature	668
Chapter 11 Collaborative DataWriters		
11.1	Collaborative DataWriters Use Cases	671
11.2	DDS Sample Combination (Synchronization) Process in a DataReader	672
11.3	Configuring Collaborative DataWriters	673
11.3.1	Associating Virtual GUIDs with DDS Data Samples	673
11.3.2	Associating Virtual Sequence Numbers with DDS Data Samples	673
11.3.3	Specifying which DataWriters will Deliver DDS Samples to the DataReader from a Logical Data Source	673
11.3.4	Specifying How Long to Wait for a Missing DDS Sample	673
11.4	Collaborative DataWriters and Persistence Service	674
Chapter 12 Mechanisms for Achieving Information Durability and Persistence		
12.1	Introduction	675
12.1.1	Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History)	676
12.1.2	Scenario 2: DataReader Restarts While DataWriter Stays Up (Durable Reader State)	677
12.1.3	Scenario 3. DataReader Joins after DataWriter Leaves Domain (Durable Data)	679
12.2	Durability and Persistence Based on Virtual GUIDs	680
12.3	Durable Writer History	681
12.3.1	Durable Writer History Use Case	682

12.3.2	How To Configure Durable Writer History	683
12.4	Durable Reader State	686
12.4.1	Durable Reader State With Protocol Acknowledgment	687
12.4.1.1	Bandwidth Utilization	688
12.4.2	Durable Reader State with Application Acknowledgment	688
12.4.2.1	Bandwidth Utilization	689
12.4.3	Durable Reader State Use Case	689
12.4.4	How To Configure a DataReader for Durable Reader State	690
12.5	Data Durability	692
12.5.1	RTI Persistence Service	692
Chapter 13 Guaranteed Delivery of Data		
13.1	Introduction	695
13.1.1	Identifying the Required Consumers of Information	697
13.1.2	Ensuring Consumer Applications Process the Data Successfully	698
13.1.3	Ensuring Information is Available to Late-Joining Applications	699
13.2	Scenarios	700
13.2.1	Scenario 1: Guaranteed Delivery to a-priori Known Subscribers	701
13.2.2	Scenario 2: Surviving a Writer Restart when Delivering DDS Samples to a priori Known Subscribers	703
13.2.3	Scenario 3: Delivery Guaranteed by Persistence Service (Store and Forward) to a priori Known Subscribers	704
13.2.3.1	Variation: Using Redundant Persistence Services	706
13.2.3.2	Variation: Using Load-Balanced Persistent Services	707
Chapter 14 Discovery		
14.1	What is Discovery?	710
14.1.1	Simple Participant Discovery	710
14.1.2	Simple Endpoint Discovery	711
14.2	Configuring the Peers List Used in Discovery	711
14.2.1	Peer Descriptor Format	713
14.2.1.1	Locator Format	714
14.2.1.2	Address Format	715
14.2.2	NDDS_DISCOVERY_PEERS Environment Variable Format	716
14.2.3	NDDS_DISCOVERY_PEERS File Format	717
14.3	Discovery Implementation	717
14.3.1	Participant Discovery	718
14.3.1.1	Refresh Mechanism	722

14.3.1.2	Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT	724
14.3.2	Endpoint Discovery	728
14.3.3	Discovery Traffic Summary	733
14.3.4	Discovery-Related QoS	734
14.4	Debugging Discovery	735
14.5	Ports Used for Discovery	738
14.5.1	Inbound Ports for Meta-Traffic	739
14.5.2	Inbound Ports for User Traffic	740
14.5.3	Automatic Selection of participant_id and Port Reservation	740
14.5.4	Tuning domain_id_gain and participant_id_gain	740
Chapter 15 Transport Plugins		
15.1	Builtin Transport Plugins	743
15.2	Extension Transport Plugins	744
15.3	The NDDSTransportSupport Class	745
15.4	Explicitly Creating Builtin Transport Plugin Instances	746
15.5	Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties()	746
15.6	Setting Builtin Transport Properties with the PropertyQoSPolicy	748
15.6.1	Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6	763
15.6.2	Formatting Rules for IPv6 ‘Allow’ and ‘Deny’ Address Lists	765
15.7	Installing Additional Builtin Transport Plugins with register_transport()	765
15.7.1	Transport Lifecycles	766
15.7.2	Transport Aliases	767
15.7.3	Transport Network Addresses	768
15.8	Installing Additional Builtin Transport Plugins with PropertyQoSPolicy	768
15.9	Other Transport Support Operations	769
15.9.1	Adding a Send Route	769
15.9.2	Adding a Receive Route	770
15.9.3	Looking Up a Transport Plugin	771
Chapter 16 Built-In Topics		
16.1	Listeners for Built-in Entities	772
16.2	Built-in DataReaders	773
16.2.1	LOCATOR_FILTER QoS Policy (DDS Extension)	782
16.3	Accessing the Built-in Subscriber	783
16.4	Restricting Communication—Ignoring Entities	784
16.4.1	Ignoring Specific Remote DomainParticipants	785
16.4.2	Ignoring Publications and Subscriptions	786

16.4.3 Ignoring Topics	788
16.4.4 Resource Limits Considerations for Ignored Entities	788
16.4.5 Supervising Endpoint Discovery	788
Chapter 17 Configuring QoS with XML	
17.1 Example XML File	791
17.2 QoS Libraries	792
17.3 QoS Profiles	793
17.3.1 Built-in QoS Profiles	794
17.3.2 Overwriting Default QoS Values	796
17.3.3 QoS Profile Inheritance	797
17.3.4 Topic Filters	799
17.3.5 QoS Profiles with a Single QoS	802
17.4 Configuring QoS with XML	803
17.4.1 QoS Policies	803
17.4.2 Sequences	804
17.4.3 Arrays	807
17.4.4 Enumeration Values	808
17.4.5 Time Values (Durations)	808
17.4.6 Transport Properties	808
17.4.7 Thread Settings	809
17.4.8 Entity Names	809
17.5 How to Load XML-Specified QoS Settings	810
17.5.1 Loading, Reloading and Unloading Profiles	811
17.6 XML File Syntax	812
17.6.1 Using Environment Variables in XML	813
17.7 XML String Syntax	814
17.8 URL Groups	814
17.9 How the XML is Validated	815
17.9.1 Validation at Run-Time	815
17.9.2 XML File Validation During Editing	816
17.10 Using QoS Profiles in Your Connex DDS Application	817
17.10.1 Retrieving a List of Available Libraries	823
17.10.2 Retrieving a List of Available QoS Profiles	823
17.11 Configuring Logging Via XML	823
Chapter 18 Multi-channel DataWriters	
18.1 What is a Multi-channel DataWriter?	825

18.2 How to Configure a Multi-channel DataWriter	828
18.2.1 Limitations	829
18.3 Multi-Channel Configuration on the Reader Side	830
18.4 Where Does the Filtering Occur?	832
18.4.1 Filtering at the DataWriter	832
18.4.2 Filtering at the DataReader	832
18.4.3 Filtering on the Network Hardware	833
18.5 Fault Tolerance and Redundancy	833
18.6 Reliability with Multi-Channel DataWriters	834
18.6.1 Reliable Delivery	834
18.6.2 Reliable Protocol Considerations	834
18.7 Performance Considerations	835
18.7.1 Network-Switch Filtering	835
18.7.2 DataWriter and DataReader Filtering	835
Chapter 19 Connex DDS Threading Model	
19.1 Database Thread	837
19.2 Event Thread	838
19.3 Receive Threads	839
19.4 Exclusive Areas, Connex DDS Threads and User Listeners	841
19.5 Controlling CPU Core Affinity for RTI Threads	842
19.6 Configuring Thread Settings with XML	842
19.7 User-Managed Threads	844
Chapter 20 DDS Sample-Data and Instance-Data Memory Management	
20.1 DDS Sample-Data Memory Management for DataWriters	846
20.1.1 Memory Management without Batching	847
20.1.2 Memory Management with Batching	849
20.1.3 Writer-Side Memory Management when Using Java	851
20.1.4 Writer-Side Memory Management when Working with Large Data	851
20.2 DDS Sample-Data Memory Management for DataReaders	853
20.2.1 Memory Management for DataReaders Using Generated Type-Plugins	854
20.2.2 Reader-Side Memory Management when Using Java	856
20.2.3 Memory Management for DynamicData DataReaders	857
20.2.4 Memory Management for Fragmented DDS Samples	859
20.2.5 Reader-Side Memory Management when Working with Large Data	859
20.3 Instance-Data Memory Management for DataWriters	861
20.4 Instance-Data Memory Management for DataReaders	861

Chapter 21 Troubleshooting

21.1 What Version am I Running?	863
21.1.1 Finding Version Information in Revision Files	863
21.1.2 Finding Version Information Programmatically	864
21.2 Controlling Messages from Connex DDS	865
21.2.1 Format of Logged Messages	868
21.2.1.1 Timestamps	868
21.2.1.2 Thread identification	869
21.2.1.3 Hierarchical Context	869
21.2.1.4 Explanation of Context Strings	869
21.2.2 Configuring Logging via XML	871
21.2.3 Customizing the Handling of Generated Log Messages	872

Part 4: Request-Reply Communication Pattern 873

Chapter 22 Introduction to the Request-Reply Communication Pattern

22.1 The Request-Reply Pattern	875
22.1.1 Request-Reply Correlation	877
22.2 Single-Request, Multiple-Replies	877
22.3 Multiple Repliers	878
22.4 Combining Request-Reply and Publish-Subscribe	879

Chapter 23 Using the Request-Reply Communication Pattern

23.1 Requesters	881
23.1.1 Creating a Requester	882
23.1.2 Destroying a Requester	883
23.1.3 Setting Requester Parameters	883
23.1.4 Summary of Requester Operations	884
23.1.5 Sending Requests	885
23.1.6 Processing Incoming Replies with a Requester	886
23.1.6.1 Waiting for Replies	886
23.1.6.2 Getting Replies	887
23.1.6.3 Receiving Replies	889
23.2 Repliers	890
23.2.1 Creating a Replier	890
23.2.2 Destroying a Replier	891
23.2.3 Setting Replier Parameters	891
23.2.4 Summary of Replier Operations	892
23.2.5 Processing Incoming Requests with a Replier	893

23.2.5.1	Waiting for Requests	894
23.2.5.2	Reading and Taking Requests	894
23.2.5.3	Receiving Requests	895
23.2.6	Sending Replies	896
23.3	SimpleRepliers	896
23.3.1	Creating a SimpleReplier	897
23.3.2	Destroying a SimpleReplier	897
23.3.3	Setting SimpleReplier Parameters	897
23.3.4	Getting Requests and Sending Replies with a SimpleReplierListener	898
23.4	Accessing Underlying DataWriters and DataReaders	898
Part 5: RTI Secure WAN Transport	900
Chapter 24 Introduction to Secure WAN Transport		
24.1	WAN Traversal via UDP Hole-Punching	902
24.1.1	Protocol Details	903
24.2	WAN Locators	907
24.3	Datagram Transport-Layer Security (DTLS)	908
24.3.1	Security Model	909
24.3.2	Liveliness Mechanism	909
24.4	Certificate Support	909
24.5	License Issues	911
Chapter 25 Configuring RTI Secure WAN Transport		
25.1	Example Applications	914
25.2	Setting Up a Transport with the Property QoS	915
25.3	WAN Transport Properties	917
25.4	Secure Transport Properties	925
25.5	Explicitly Instantiating a WAN or Secure Transport Plugin	930
25.5.1	Additional Header Files and Include Directories	931
25.5.2	Additional Libraries	931
25.5.3	Compiler Flags	931
Part 6: RTI Persistence Service	932
Chapter 26 Introduction to RTI Persistence Service	933
Chapter 27 Configuring Persistence Service		
27.1	How to Load the Persistence Service XML Configuration	935
27.2	XML Configuration File	936
27.2.1	Configuration File Syntax	937
27.2.2	XML Validation	938

27.2.2.1	Validation at Run Time	938
27.2.2.2	Validation During Editing	938
27.3	QoS Configuration	939
27.4	Configuring the Persistence Service Application	940
27.5	Configuring Remote Administration	942
27.6	Configuring Persistent Storage	943
27.7	Configuring Participants	946
27.8	Creating Persistence Groups	947
27.8.1	QoSs	952
27.8.2	DurabilityService QoS Policy	953
27.8.3	Sharing a Publisher/Subscriber	953
27.8.4	Sharing a Database Connection	954
27.8.5	Memory Management	954
27.9	Configuring Durable Subscriptions in Persistence Service	955
27.9.1	DDS Sample Memory Management With Durable Subscriptions	956
27.10	Synchronizing of Persistence Service Instances	956
27.11	Enabling RTI Distributed Logger in Persistence Service	957
27.12	Enabling RTI Monitoring Library in Persistence Service	958
27.13	Support for Extensible Types	959
27.13.1	Type Version Discrimination	960
27.14	TCP Transport Support in Persistence Service	960
Chapter 28 Running RTI Persistence Service		
28.1	Starting Persistence Service	962
28.2	Stopping Persistence Service	965
Chapter 29 Administering Persistence Service from a Remote Location		
29.1	Enabling Remote Administration	966
29.2	Remote Commands	967
29.2.1	start	967
29.2.2	stop	967
29.2.3	shutdown	968
29.2.4	status	968
29.3	Accessing Persistence Service from a Connex DDS Application	968
Chapter 30 Advanced Persistence Service Scenarios		
30.1	Scenario: Load-balanced Persistence Services	972
30.2	Scenario: Delegated Reliability	974
30.3	Scenario: Slow Consumer	975

Part 7: RTI CORBA Compatibility Kit	979
Chapter 31 Introduction to RTI CORBA Compatibility Kit	980
Chapter 32 Generating CORBA-Compatible Code	
32.1 Generating C++ Code	983
32.2 Generating Java Code	984
Chapter 33 Supported IDL Types	985
Part 8: RTI TCP Transport	987
Chapter 34 TCP Communication Scenarios	
34.1 Communication Within a Single LAN	988
34.2 Symmetric Communication Across NATs	989
34.3 Asymmetric Communication Across NATs	990
35.1 Configuring the TCP Transport	993
35.1.1 Choosing a Transport Mode	993
35.1.2 Explicitly Instantiating the TCP Transport Plugin	994
35.1.2.1 Additional Header Files and Include Directories	995
35.1.2.2 Additional Libraries and Compiler Flags	995
35.1.3 Configuring the TCP Transport with the Property QoSPolicy	996
35.1.3.1 Configuring the TCP Transport to be Loaded Statically	998
35.1.3.2 Loading TLS Support Libraries Statically	999
35.1.4 Setting the Initial Peers	999
35.1.5 Support for External Hardware Load Balancers in TCP Transport Plugin	1000
35.1.5.1 Session-ID Messages	1002
35.1.6 TCP/TLS Transport Properties	1002
35.1.6.1 Connection Liveliness	1020
Part 9: RTI Monitoring Library	1022
Chapter 36 Using Monitoring Library in Your Application	
36.1 Enabling Monitoring	1024
36.1.1 Method 1—Change the Participant QoS to Automatically Load the Dynamic Monitoring Library	1025
36.1.2 Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library	1025
36.1.2.1 Method 2-A: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in Source Code	1026
36.1.2.2 Method 2-B: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in an Environment Variable	1029
36.2 How does Monitoring Library Work?	1031
36.3 What Monitoring Topics are Published?	1031
36.4 Enabling Support for Large Type-Code (Optional)	1032

36.5 Troubleshooting Monitoring	1033
36.5.1 Buffer Allocation Error	1033
Chapter 37 Configuring Monitoring Library	1034
Part 10: RTI Distributed Logger	1039
Chapter 38 Using Distributed Logger in a Connex DDS Application	
38.1 Using the API Directly	1041
38.2 Examples	1042
38.3 Data Type Resource	1043
38.4 Distributed Logger Topics	1044
38.5 Distributed Logger IDL	1044
38.6 Viewing Log Messages	1045
38.7 Logging Levels	1045
38.8 Distributed Logger Quality of Service Settings	1046
Chapter 39 Enabling Distributed Logger in RTI Services	
39.1 Relationship Between Service Verbosity and Filter Level	1052

About this Document

Paths Mentioned in Documentation

The documentation refers to:

- **<NDDSHOME>**

This refers to the installation directory for Connex DDS. The default installation paths are:

- Mac OS X systems:
/Applications/rti_connex_dds-5.2.3
- UNIX-based systems, non-*root* user:
/home/your user name/rti_connex_dds-5.2.3
- UNIX-based systems, *root* user:
/opt/rti_connex_dds-5.2.3
- Windows systems, user without Administrator privileges:
<your home directory>\rti_connex_dds-5.2.3
- Windows systems, user with Administrator privileges:
C:\Program Files\rti_connex_dds-5.2.3 (64-bit machines)
C:\Program Files (x86)\rti_connex_dds-5.2.3 (32-bit machines)

You may also see \$NDDSHOME or %NDDSHOME%, which refers to an environment variable set to the installation path.

Wherever you see <NDDSHOME> used in a path, replace it with your installation path.

Note for Windows Users: When using a command prompt to enter a command that includes the path **C:\Program Files** (or any directory name that has a space), enclose the path in quotation marks. For example:

```
"C:\Program Files\rti_connex_dds-5.2.3\bin\rtiddsgen"
```

Or if you have defined the NDDSHOME environment variable:

```
"%NDDSHOME%\bin\rtiddsgen"
```

- **<path to examples>**

By default, examples are copied into your home directory the first time you run *RTI Launcher* or any script in `<NDDSHOME>/bin`. This document refers to the location of the copied examples as `<path to examples>`.

Wherever you see `<path to examples>`, replace it with the appropriate path.

Default path to the examples:

- Mac OS X systems: `/Users/your user name/rti_workspace/5.2.3/examples`
- UNIX-based systems: `/home/your user name/rti_workspace/5.2.3/examples`
- Windows systems: `your Windows documents folder\rti_workspace\5.2.3\examples`

Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 7, the folder is `C:\Users\your user name\Documents`; on Windows Server 2003, the folder is `C:\Documents and Settings\your user name\Documents`.

Note: You can specify a different location for `rti_workspace`. You can also specify that you do not want the examples copied to the workspace. For details, see *Controlling Location for RTI Workspace and Copying of Examples* in the *Connex DDS Core Libraries Getting Started Guide*.

Programming Language Conventions

The terminology and example code in this manual assume you are using Traditional C++ without namespace support.

C, Modern C++, C++/CLI, C#, and Java APIs are also available; they are fully described in the API Reference HTML documentation. (Note: the Modern C++ API is not available for all platforms, check the [RTI Connex DDS Core Libraries Platform Notes](#) to see if it is available for your platform.)

Namespace support in Traditional C++, C++/CLI, and C# is also available; see the API Reference HTML documentation (from the **Modules** page, select **Using DDS:: Namespace**) for details. In the Modern C++ API all types, constants and functions are always in namespaces.

Traditional vs. Modern C++

Connex DDS provides two different C++ APIs, which we refer to as the "Traditional C++" and "Modern C++" APIs. They provide substantially different programming paradigms and patterns. The Traditional API could be considered as simply "C with classes," while the Modern API incorporates modern C++ techniques, most notably:

- Generic programming
- Integration with the standard library
- Automatic object lifecycle management, providing full value types and reference types
- C++11 support, such as move operations, initializer lists, and support for range for-loops.

These different programming styles make the Modern C++ API differ significantly with respect to the other language APIs in several aspects; to name a few:

- [Creating and Deleting DDS Entities \(Section 4.1.1 on page 153\)](#)
- [Creating User Data Types with IDL \(Section 3.3 on page 69\)](#)
- [Interacting Dynamically with User Data Types \(Section 3.8 on page 141\)](#)
- [Working with DDS Data Samples \(Section 3.9 on page 145\)](#)
- [Using DataReaders to Access Data \(Read & Take\) \(Section 7.4 on page 491\)](#)
- QoS policies and QoS management
- Naming conventions

This manual points out these kinds of differences whenever they are substantial.

Extensions to the DDS Standard

Connex DDS implements the DDS Standard published by the OMG. It also includes features that are extensions to DDS. These include additional Quality of Service parameters, function calls, structure fields, etc.

Extensions also include product-specific APIs that complement the DDS API. These include APIs to create and use transport plug-ins, and APIs to control the verbosity and logging capabilities. These APIs are prefixed with NDDS, such as **NDDSTransportSupport::register_transport()**.

Environment Variables

Connex DDS documentation refers to path names that have been customized during installation. NDDSHOME refers to the installation directory of Connex DDS.

Names of Supported Platforms

Connex DDS runs on several different target platforms. To support this vast array of platforms, Connex DDS separates the executable, library, and object files for each platform into individual directories.

Each platform name has four parts: hardware architecture, operating system, operating system version and compiler. For example, **i86Linux2.4gcc3.2** is the directory that contains files specific to Linux® version 2.4 for the Intel processor, compiled with gcc version 3.2.

For a full list of supported platforms, see the [RTI Connex DDS Core Libraries Platform Notes](#).

Additional Resources

The details of each API (such as function parameters, return values, etc.) and examples are in the API Reference HTML documentation. In case of discrepancies between the information in this document and the API Reference HTML documentation, the latter should be considered more up-to-date.

Part 1: Welcome to RTI Connex DDS

RTI Connex DDS solutions provide a flexible data distribution infrastructure for integrating data sources of all types. At its core is the world's leading ultra-high performance, distributed networking *DataBus*[™]. It connects data within applications as well as across devices, systems and networks. Connex DDS also delivers large data sets with microsecond performance and granular quality-of-service control. Connex DDS is a standards-based, open architecture that connects devices from deeply embedded real-time platforms to enterprise servers across a variety of networks.

Part 1 introduces the general concepts behind data-centric publish-subscribe communications and provides a brief tour of Connex DDS.

- [Overview \(Section Chapter 1 on page 2\)](#)
- [Data-Centric Publish-Subscribe Communications \(Section Chapter 2 on page 10\)](#)

Chapter 1 Overview

RTI Connex DDS is network middleware for distributed real-time applications. Connex DDS simplifies application development, deployment and maintenance and provides fast, predictable distribution of time-critical data over a variety of transport networks.

Connex DDS solutions provide a flexible data distribution infrastructure for integrating data sources of all types. At its core is the world's leading ultra-high performance, distributed networking *DataBus™*. It connects data within applications as well as across devices, systems and networks. Connex DDS also delivers large data sets with microsecond performance and granular quality-of-service control. Connex DDS is a standards-based, open architecture that connects devices from deeply embedded real-time platforms to enterprise servers across a variety of networks.

With Connex DDS, you can:

- Perform complex one-to-many and many-to-many network communications.
- Customize application operation to meet various real-time, reliability, and quality-of-service goals.
- Provide application-transparent fault tolerance and application robustness.
- Use a variety of transports.

This section introduces basic concepts of middleware and common communication models, and describes how Connex DDS's feature-set addresses the needs of real-time systems.

1.1 What is Connex DDS?

Connex DDS is network middleware for real-time distributed applications. It provides the communications service programmers need to distribute time-critical data between embedded and/or enterprise devices or nodes. Connex DDS uses the publish-subscribe communications model to make data distribution efficient and robust.

Connex DDS implements the Data-Centric Publish-Subscribe (DCPS) API within the OMG's Data Distribution Service (DDS) for Real-Time Systems. DDS is the first standard developed for the needs of real-time systems. DCPS provides an efficient way to transfer data in a distributed system.

With Connex DDS, systems designers and programmers start with a fault-tolerant and flexible communications infrastructure that will work over a wide variety of computer hardware, operating systems, languages, and networking transport protocols. Connex DDS is highly configurable so programmers can adapt it to meet the application's specific communication requirements.

1.2 Network Communications Models

The communications model underlying the network middleware is the most important factor in how applications communicate. The communications model impacts the performance, the ease to accomplish different communication transactions, the nature of detecting errors, and the robustness to different error conditions. Unfortunately, there is no "one size fits all" approach to distributed applications. Different communications models are better suited to handle different classes of application domains.

This section describes three main types of network communications models:

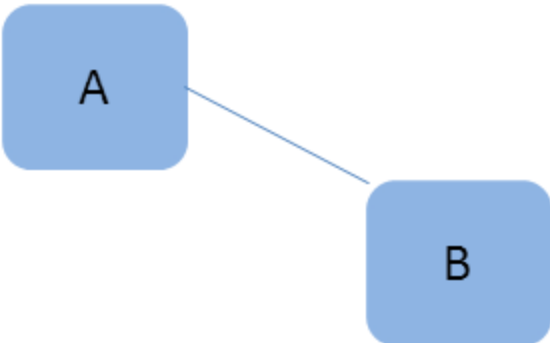
- Point-to-point
- Client-server
- Publish-subscribe

Point-to-point model:

Point-to-point is the simplest form of communication, as illustrated in [Figure 1.1 Point-to-Point on the facing page](#). The telephone is an example of an everyday point-to-point communications device. To use a telephone, you must know the address (phone number) of the other party. Once a connection is established, you can have a reasonably high-bandwidth conversation. However, the telephone does not work as well if you have to talk to many people at the same time. The telephone is essentially one-to-one communication.

TCP is a point-to-point network protocol designed in the 1970s. While it provides reliable, high-bandwidth communication, TCP is cumbersome for systems with many communicating nodes.

Figure 1.1 Point-to-Point

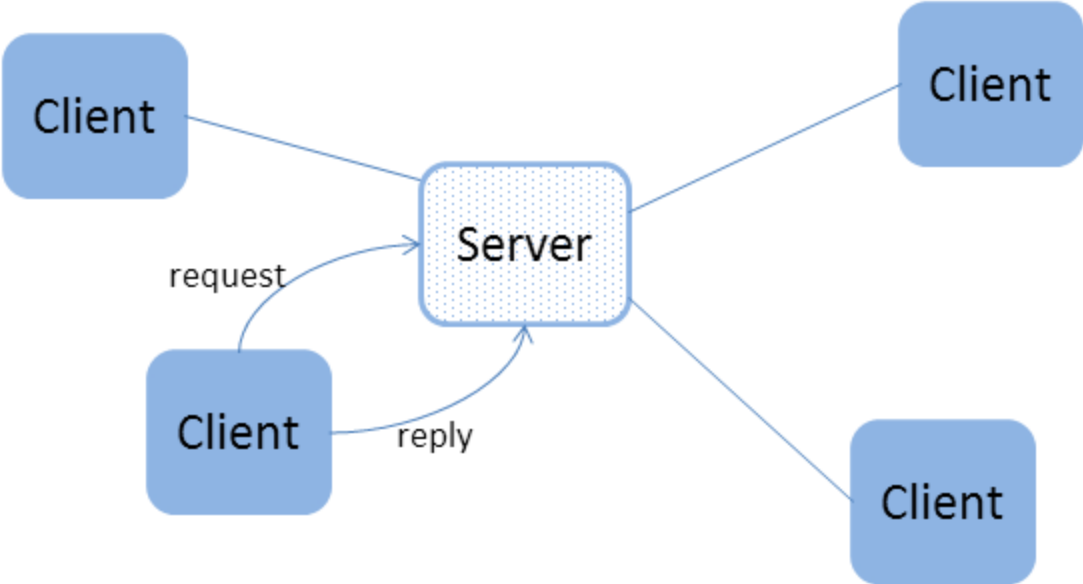


Point-to-point is one-to-one communication.

Client-server model:

To address the scalability issues of the Point-to-Point model, developers turned to the Client-Server model. Client-server networks designate one special server node that connects simultaneously to many client nodes, as illustrated in [Figure 1.2 Client-Server](#) below.

Figure 1.2 Client-Server



Client-server is many-to-one communications.

Client-server is a "many-to-one" architecture. Ordering pizza over the phone is an example of client-server communication. Clients must know the phone number of the pizza parlor to place an order. The parlor can handle many orders without knowing ahead of time where people (clients) are located. After the order (request), the parlor asks the client where the response (pizza) should be sent. In the client-server model, each response is tied to a prior request. As a result, the response can be tailored to each request. In other words, each client makes a request (order) and each reply (pizza) is made for one specific client in mind.

The client-server network architecture works best when information is centralized, such as in databases, transaction processing systems, and file servers. However, if information is being generated at multiple nodes, a client-server architecture requires that all information are sent to the server for later redistribution to the clients. This approach is inefficient and precludes deterministic communications, since the client does not know when new information is available. The time between when the information is available on the server, and when the client asks and receives it adds a variable latency to the system.

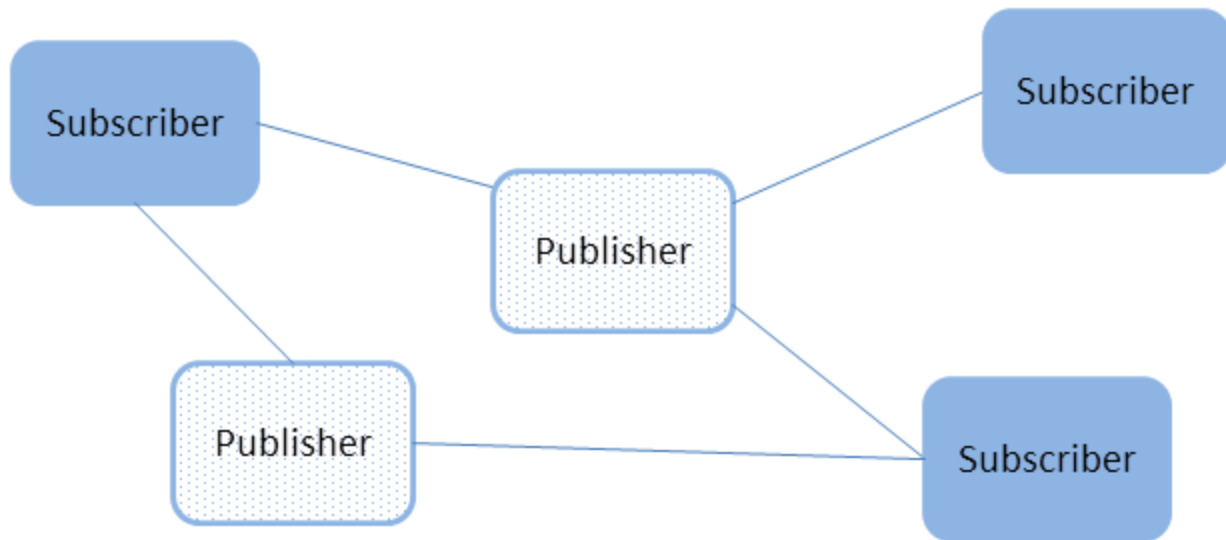
Publish-subscribe model: In the publish-subscribe communications model ([Figure 1.3 Publish-Subscribe on the facing page](#)), computer applications (nodes) “subscribe” to data they need and “publish” data they want to share. Messages pass directly between the publisher and the subscribers, rather than moving into and out of a centralized server. Most time-sensitive information intended to reach many people is sent by a publish-subscribe system. Examples of publish-subscribe systems in everyday life include television, magazines, and newspapers.

Publish-subscribe communication architectures are good for distributing large quantities of time-sensitive information efficiently, even in the presence of unreliable delivery mechanisms. This direct and simultaneous communication among a variety of nodes makes publish-subscribe network architecture the best choice for systems with complex time-critical data flows.

While the publish-subscribe model provides system architects with many advantages, it may not be the best choice for all types of communications, including:

- File-based transfers (alternate solution: FTP)
- Remote Method Invocation (alternate solutions: CORBA, COM, SOAP)
- Connection-based architectures (alternate solution: TCP/IP)
- Synchronous transfers (alternate solution: CORBA)

Figure 1.3 Publish-Subscribe

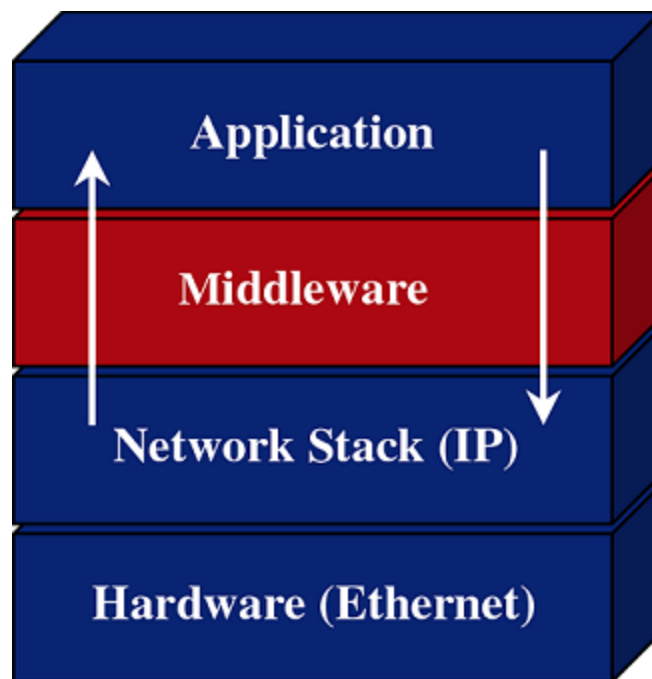


Publish-subscribe is many-to-many communications.

1.3 What is Middleware?

Middleware is a software layer between an application and the operating system. *Network middleware* isolates the application from the details of the underlying computer architecture, operating system and network stack (see [Figure 1.4 Network Middleware on the next page](#)). Network middleware simplifies the development of distributed systems by allowing applications to send and receive information without having to program using lower-level protocols such as sockets and TCP or UDP/IP.

Figure 1.4 Network Middleware



Connex DDS is middleware that insulates applications from the raw operating-system network stack.

Publish-subscribe middleware: Connex DDS is based on a publish-subscribe communications model. Publish-subscribe (PS) middleware provides a simple and intuitive way to distribute data. It decouples the software that creates and sends data—the data *publishers*—from the software that receives and uses the data—the data *subscribers*. Publishers simply declare their intent to send and then publish the data. Subscribers declare their intent to receive, then the data is automatically delivered by the middleware.

Despite the simplicity of the model, PS middleware can handle complex patterns of information flow. The use of PS middleware results in simpler, more modular distributed applications. Perhaps most importantly, PS middleware can automatically handle all network chores, including connections, failures, and network changes, eliminating the need for user applications to program of all those special cases. What experienced network middleware developers know is that handling special cases accounts for over 80% of the effort and code.

1.4 Features of Connex DDS

Connex DDS supports mechanisms that go beyond the basic publish-subscribe model. The key benefit is that applications that use Connex DDS for their communications are entirely decoupled. Very little of their design time has to be spent on how to handle their mutual interactions. In particular, the applications never need information about the other participating applications, including their existence or locations. Connex DDS automatically handles all aspects of message delivery, without requiring any intervention from the user applications, including:

- determining who should receive the messages,
- where recipients are located,
- what happens if messages cannot be delivered.

This is made possible by how Connex DDS allows the user to specify Quality of Service (QoS) parameters as a way to configure automatic-discovery mechanisms and specify the behavior used when sending and receiving messages. The mechanisms are configured up-front and require no further effort on the user's part. By exchanging messages in a completely anonymous manner, Connex DDS greatly simplifies distributed application design and encourages modular, well-structured programs.

Furthermore, Connex DDS includes the following features, which are designed to meet the needs of distributed real-time applications:

- **Data-centric publish-subscribe communications:** Simplifies distributed application programming and provides time-critical data flow with minimal latency.
 - Clear semantics for managing multiple sources of the same data.
 - Efficient data transfer, customizable Quality of Service, and error notification.
 - Guaranteed periodic samples, with maximum rate set by subscriptions.
 - Notification by a callback routine on data arrival to minimize latency.
 - Notification when data does not arrive by an expected deadline.
 - Ability to send the same message to multiple computers efficiently.
- **User-definable data types:** Enables you to tailor the format of the information being sent to each application.
- **Reliable messaging:** Enables subscribing applications to specify reliable delivery of samples.
- **Multiple Communication Networks:** Multiple independent communication networks (DDS *domains*), each using Connex DDS, can be used over the same physical network. Applications are only able to participate in the DDS domains to which they belong. Individual applications can be configured to participate in multiple DDS domains.
- **Symmetric architecture:** Makes your application robust:
 - No central server or privileged nodes, so the system is robust to node failures.
 - Subscriptions and publications can be dynamically added and removed from the system at any time.
- **Pluggable Transports Framework:** Includes the ability to define new transport plug-ins and run over them. Connex DDS comes with a standard UDP/IP pluggable transport and a shared memory transport. It can be configured to operate over a variety of transport mechanisms, including back-planes, switched fabrics, and new networking technologies.

- **Multiple Built-in Transports:** Includes UDP/IP and shared memory transports.
- **Multi-language support:** Includes APIs for the C, C++ (Traditional and Modern APIs), C++/CLI, C#, and Java™ programming languages.
- **Multi-platform support:** Includes support for flavors of UNIX®, real-time operating systems, and Windows®. (Consult the [RTI Connex DDS Core Libraries Platform Notes](#) to see which platforms are supported in this release.)
- **Compliance with Standards:**
 - API complies with the DCPS layer of the OMG's DDS specification.
 - Data types comply with OMG Interface Definition Language™ (IDL).
 - Data packet format complies with the International Engineering Consortium's (IEC's) publicly available specification for the RTPS wire protocol.

Chapter 2 Data-Centric Publish-Subscribe Communications

This section describes the formal communications model used by Connex DDS: the Data-Centric Publish-Subscribe (DCPS) standard. DCPS is a formalization (through a standardized API) and extension of the publish-subscribe communications model presented in [Network Communications Models \(Section 1.2 on page 3\)](#).

This section includes:

2.1 What is DCPS?

DCPS is the portion of the OMG DDS (Data Distribution Service) Standard that addresses data-centric publish-subscribe communications. The DDS standard defines a language-independent model of publish-subscribe communications that has standardized mappings into various implementation languages. Connex DDS offers C, Traditional C++, Modern C++, C++/CLI, C#, and Java versions of the DCPS API.

The publish-subscribe approach to distributed communications is a generic mechanism that can be employed by many different types of applications. The DCPS model described in this chapter extends the publish-subscribe model to address the specific needs of real-time, data-critical applications. As you'll see, it provides several mechanisms that allow application developers to control how communications works and how the middleware handles resource limitations and error conditions.

The “data-centric” portion of the term DCPS describes the fundamental concept supported by the design of the API. In data-centric communications, the focus is on the distribution of *data* between communicating applications. A data-centric system is comprised of data publishers and data subscribers. The communications are based on passing data of known types in named streams from publishers to subscribers.

In contrast, in object-centric communications the fundamental concept is the *interface* between the applications. An interface is comprised of a set of methods of known types (number and types of method arguments). An object-centric system is comprised of interface servers and interface clients, and communications are based on clients invoking methods on named interfaces that are serviced by the corresponding server.

Data and object-centric communications are complementary paradigms in a distributed system. Applications may require both. However, real-time communications often fit a data-centric model more naturally.

2.1.1 DCPS for Real-Time Requirements

DCPS, and specifically the Connex DDS implementation, is well suited for real-time applications. For instance, real-time applications often require the following features:

- **Efficiency**

Real-time systems require efficient data collection and delivery. Only minimal delays should be introduced into the critical data-transfer path. Publish-subscribe is more efficient than client-server in both latency and bandwidth for periodic data exchange.

Publish-subscribe greatly reduces the overhead required to send data over the network compared to a client-server architecture. Occasional subscription requests, at low bandwidth, replace numerous high-bandwidth client requests. Latency is also reduced, since the outgoing request message time is eliminated. As soon as a new DDS sample becomes available, it is sent to the corresponding subscriptions.

- **Determinism**

Real-time applications often care about the determinism of delivering periodic data as well as the latency of delivering event data. Once buffers are introduced into a data stream to support reliable connections, new data may be held undelivered for a unpredictable amount of time while waiting for confirmation that old data was received.

Since publish-subscribe does not inherently require reliable connections, implementations, like Connex DDS, can provide configurable trade-offs between the deterministic delivery of new data and the reliable delivery of all data.

- **Flexible delivery bandwidth**

Typical real-time systems include both real-time and non-real-time nodes. The bandwidth requirements for these nodes—even for the same data—are quite different. For example, an application may be sending DDS samples faster than a non-real-time application is capable of handling.

However, a real-time application may want the same data as fast as it is produced.

DCPS allows subscribers to the same data to set individual limits on how fast data should be delivered to each subscriber. This is similar to how some people get a newspaper every day while others can subscribe to only the Sunday paper.

- Thread awareness

Real-time communications must work without slowing the thread that sends DDS samples. On the receiving side, some data streams should have higher priority so that new data for those streams are processed before lower priority streams.

Connex DDS provides user-level configuration of its internal threads that process incoming data. Users may configure Connex DDS so that different threads are created with different priorities to process received data of different data streams.

- Real-time communications must work without slowing the thread that sends DDS samples. On the receiving side, some data streams should have higher priority so that new data for those streams are processed before lower priority streams.
Connex DDS provides user-level configuration of its internal threads that process incoming data. Users may configure Connex DDS so that different threads are created with different priorities to process received data of different data streams.

- Fault-tolerant operation

Real-time applications are often in control of systems that are required to run in the presence of component failures. Often, those systems are safety critical or carry financial penalties for loss of service. The applications running those systems are usually designed to be fault-tolerant using redundant hardware and software. Backup applications are often “hot” and interconnected to primary systems so that they can take over as soon as a failure is detected.

Publish-subscribe is capable of supporting many-to-many connectivity with redundant *DataWriters* and *DataReaders*. This feature is ideal for constructing fault-tolerant or high-availability applications with redundant nodes and robust fault detection and handling services.

- DCPS, and thus Connex DDS, was designed and implemented specifically to address the requirements above through configuration parameters known as QosPolicies defined by the DCPS standard (see [QosPolicies \(Section 4.2 on page 162\)](#)). [DDS Data Types, Topics, Keys, Instances, and Samples \(Section 2.2 below\)](#) introduces basic DCPS terminology and concepts.

2.2 DDS Data Types, Topics, Keys, Instances, and Samples

In data-centric communications, the applications participating in the communication need to share a common view of the types of data being passed around.

Within different programming languages there are several ‘primitive’ data types that all users of that language naturally share (integers, floating point numbers, characters, booleans, etc.). However, in any non-trivial software system, specialized data types are constructed out of the language primitives. So the data to be shared between applications in the communication system could be structurally simple, using the primitive language types mentioned above, or it could be more complicated, using, for example, C and C++ structs, like this:

```
struct Time {
    long year;
    short day;
```

```

    short hour;
    short minute;
    short second;
};
struct StockPrice {
    float price;
    Time timeStamp;
};

```

Within a set of applications using DCPS, the different applications do not automatically know the structure of the data being sent, nor do they necessarily interpret it in the same way (if, for instance, they use different operating systems, were written with different languages, or were compiled with different compilers). There must be a way to share not only the data, but also information about how the data is structured.

In DCPS, data definitions are shared among applications using OMG IDL, a language-independent means of describing data. For more information on data types and IDL, see [Data Types and DDS Data Samples \(Section Chapter 3 on page 23\)](#).

2.3 Data Topics – What is the Data Called?

Shared knowledge of the data types is a requirement for different applications to communicate with DCPS. The applications must also share a way to identify which data is to be shared. Data (of *any* data type) is uniquely distinguished by using a name called a *Topic*. By definition, a *Topic* corresponds to a single data type. However, several *Topics* may refer to the same data type.

Topics interconnect *DataWriters* and *DataReaders*. A *DataWriter* is an object in an application that tells Connex DDS (and indirectly, other applications) that it has some values of a certain *Topic*. A corresponding *DataReader* is an object in an application that tells Connex DDS that it wants to receive values for the same *Topic*. And the data that is passed from the *DataWriter* to the *DataReader* is of the data type associated with the *Topic*. *DataWriters* and *DataReaders* are described more in [DataWriter-Publishers and DataReaders/Subscribers \(Section 2.4 on page 15\)](#).

For a concrete example, consider a system that distributes stock quotes between applications. The applications could use a data type called *StockPrice*. There could be multiple *Topics* of the *StockPrice* data type, one for each company’s stock, such as IBM, MSFT, GE, etc. Each *Topic* uses the same data type.

Data Type: *StockPrice*

```

struct StockPrice {
    float price;
    Time timeStamp;
};

```

Topic: “IBM”

Topic: “MSFT”

Topic: “GE”

Now, an application that keeps track of the current value of a client’s portfolio would subscribe to all of the topics of the stocks owned by the client. As the value of each stock changes, the new price for the corresponding topic is published and sent to the application.

2.3.1 DDS Samples, Instances, and Keys

The value of data associated with a *Topic* can change over time. The different values of the *Topic* passed between applications are called DDS samples. In our stock-price example, DDS samples show the price of a stock at a certain point in time. So each DDS sample may show a different price.

For a data type, you can select one or more fields within the data type to form a *key*. A *key* is something that can be used to uniquely identify one *instance* of a *Topic* from another *instance* of the same *Topic*. Think of a key as a way to sub-categorize or group related data values for the same *Topic*. Note that not all data types are defined to have keys, and thus, not all topics have keys. For topics without keys, there is only a single instance of that topic.

However, for *Topics* with keys, a unique value for the key identifies a unique *instance* of the *Topic*. DDS samples are then updates to particular instances of a *Topic*. Applications can subscribe to a *Topic* and receive DDS samples for many different instances. Applications can publish DDS samples of one, all, or any number of instances of a *Topic*. Many quality of service parameters actually apply on a *per instance* basis. Keys are also useful for subscribing to a group of related data streams (instances) without pre-knowledge of which data streams (instances) exist at runtime.

For example, let’s change the **StockPrice** data type to include the symbol of the stock. Then instead of having a *Topic* for every stock, which would result in hundreds or thousands of *Topics* and related *DataWriters* and *DataReaders*, each application would only have to publish or subscribe to a single *Topic*, say “StockPrices.” Successive values of a stock would be presented as successive DDS samples of an instance of “StockPrices”, with each instance corresponding to a single stock symbol.

Data Type: StockPrice

```
struct StockPrice {
    float price;
    Time   timeStamp;
    char  *symbol;   //@key
};
```

Instance 1 = (Topic: “StockPrices”) + (Key: “MSFT”)

sample a, price = \$28.00

sample b, price = \$27.88

Instance 2 = (Topic: “StockPrices”) + (Key: “IBM”)

sample a, price = \$74.02

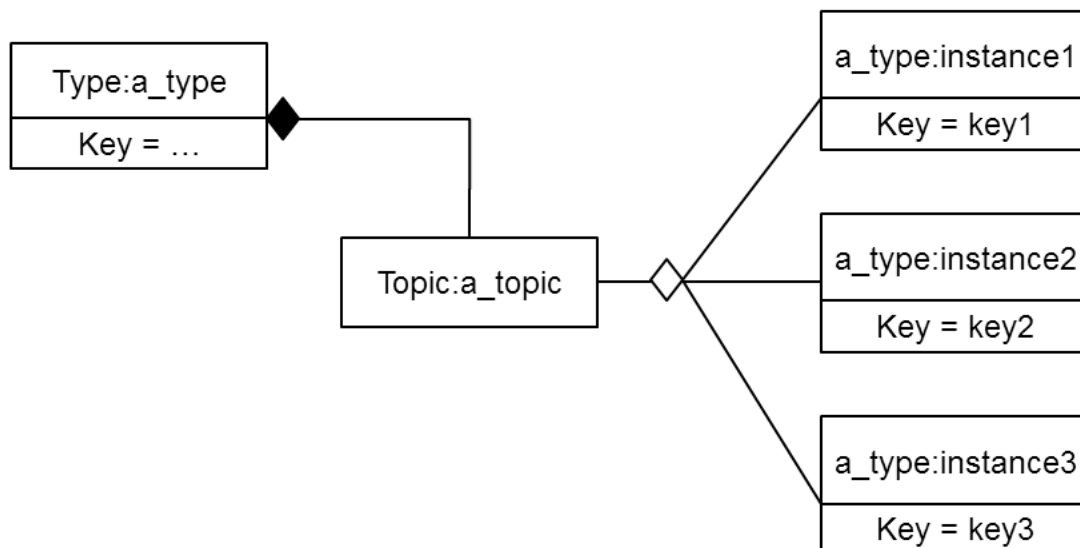
sample b, price = \$73.50

Etc.

Just by subscribing to “StockPrices,” an application can get values for all of the stocks through a single topic. In addition, the application does not have to subscribe explicitly to any particular stock, so that if a new stock is added, the application will immediately start receiving values for that stock as well.

To summarize, the unique values of data being passed using DCPS are called DDS samples. A DDS sample is a combination of a *Topic* (distinguished by a *Topic* name), an *instance* (distinguished by a *key*), and the actual *user data* of a certain data type. As seen in [Figure 2.1 Relationship of Topics, Keys, and Instances below](#), a *Topic* identifies data of a single type, ranging from one single instance to a whole collection of instances of that given topic for keyed data types. For more information, see [Data Types and DDS Data Samples \(Section Chapter 3 on page 23\)](#) and [Topics \(Section Chapter 5 on page 200\)](#).

Figure 2.1 Relationship of Topics, Keys, and Instances



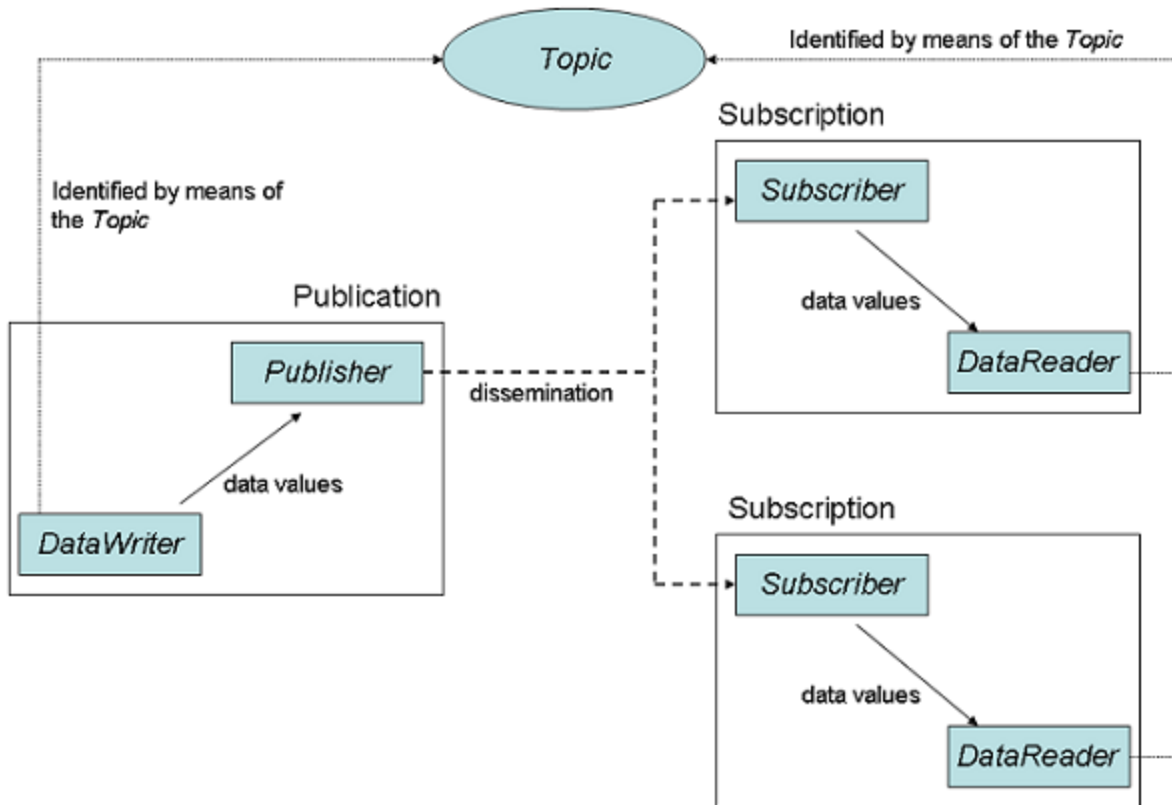
By using keys, a Topic can identify a collection of data-object instances.

2.4 DataWriters/Publishers and DataReaders/Subscribers

In DCPS, applications must use APIs to create entities (objects) in order to establish publish-subscribe communications between each other. The entities and terminology associated with the data itself have been discussed already—*Topics*, keys, instances, DDS samples. This section will introduce the DCPS entities that user code must create to send and receive the data. Note that *Entity* is actually a basic DCPS concept. In object-oriented terms, *Entity* is the base class from which other DCPS classes—*Topic*, *DataWriter*, *DataReader*, *Publisher*, *Subscriber*, *DomainParticipants*—derive. For general information on Entities, see [DDS Entities \(Section Chapter 4 on page 151\)](#).

The sending side uses objects called *Publishers* and *DataWriters*. The receiving side uses objects called *Subscribers* and *DataReaders*. [Figure 2.2 Overview](#) below illustrates the relationship of these objects.

Figure 2.2 Overview



- An application uses *DataWriters* to send data. A *DataWriter* is associated with a single *Topic*. You can have multiple *DataWriters* and *Topics* in a single application. In addition, you can have more than one *DataWriter* for a particular *Topic* in a single application.
- A *Publisher* is the DCPS object responsible for the actual sending of data. *Publishers* own and manage *DataWriters*. A *DataWriter* can only be owned by a single *Publisher* while a *Publisher* can own many *DataWriters*. Thus the same *Publisher* may be sending data for many different *Topics* of different data types. When user code calls the **write()** method on a *DataWriter*, the DDS data sample is passed to the *Publisher* object which does the actual dissemination of data on the network. For more information, see [Sending Data \(Section Chapter 6 on page 242\)](#).
- A *Publisher* is the DCPS object responsible for the actual sending of data. *Publishers* own and manage *DataWriters*. A *DataWriter* can only be owned by a single *Publisher* while a *Publisher* can own many *DataWriters*. Thus the same *Publisher* may be sending data for many different *Topics* of different data types. When user code calls the **write()** method on a *DataWriter*, the DDS data

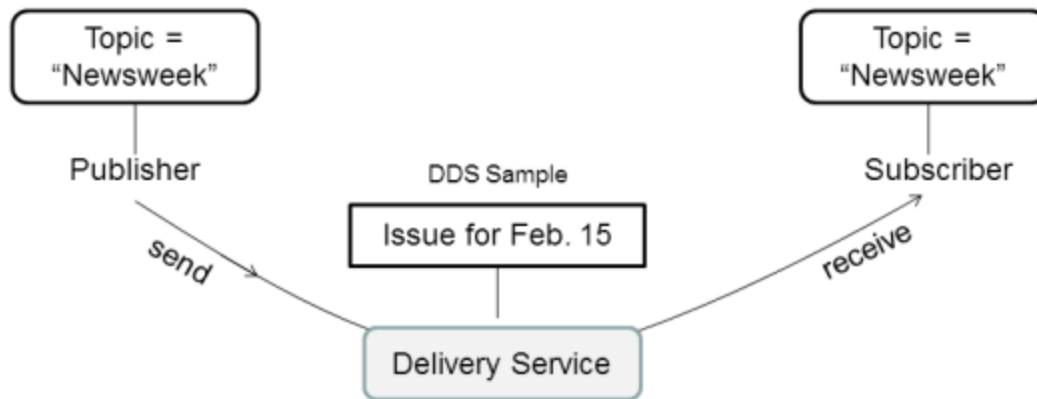
sample is passed to the *Publisher* object which does the actual dissemination of data on the network. For more information, see [Sending Data \(Section Chapter 6 on page 242\)](#).

- The association between a *DataWriter* and a *Publisher* is often referred to as a publication although you never create a DCPS object known as a publication.
- An application uses *DataReaders* to access data received over DCPS. A *DataReader* is associated with a single *Topic*. You can have multiple *DataReaders* and *Topics* in a single application. In addition, you can have more than one *DataReader* for a particular *Topic* in a single application.
- A *Subscriber* is the DCPS object responsible for the actual receipt of published data. *Subscribers* own and manage *DataReaders*. A *DataReader* can only be owned by a single *Subscriber* while a *Subscriber* can own many *DataReaders*. Thus the same *Subscriber* may receive data for many different *Topics* of different data types. When data is sent to an application, it is first processed by a *Subscriber*; the DDS data sample is then stored in the appropriate *DataReader*. User code can either register a *listener* to be called when new data arrives or actively poll the *DataReader* for new data using its **read()** and **take()** methods. For more information, see [Receiving Data \(Section Chapter 7 on page 437\)](#).
- The association between a *DataReader* and a *Subscriber* is often referred to as a subscription although you never create a DCPS object known as a subscription.

Example:

The publish-subscribe communications model is analogous to that of magazine publications and subscriptions. Think of a publication as a weekly periodical such as *Newsweek*®. The *Topic* is the name of the periodical (in this case the string "Newsweek"). The *type* specifies the format of the information, e.g., a printed magazine. The *user data* is the contents (text and graphics) of each DDS sample (weekly issue). The middleware is the distribution service (usually the US Postal service) that delivers the magazine from where it is created (a printing house) to the individual subscribers (people's homes). This analogy is illustrated in [Figure 2.3 An Example of Publish-Subscribe on the facing page](#). Note that by subscribing to a publication, subscribers are requesting current and future DDS samples of that publication (such as once a week in the case of *Newsweek*), so that as new DDS samples are published, they are delivered without having to submit another request for data.

Figure 2.3 An Example of Publish-Subscribe



The publish-subscribe model is analogous to publishing magazines. The Publisher sends DDS samples of a particular Topic to all Subscribers of that Topic. With Newsweek® magazine, the Topic would be "Newsweek." The DDS sample consists of the data (articles and pictures) sent to all Subscribers every week. The middleware (Connex DDS) is the distribution channel: all of the planes, trucks, and people who distribute the weekly issues to the Subscribers.

By default, each DDS sample is propagated individually, independently, and uncorrelated with other DDS samples. However, an application may request that several DDS samples be sent as a coherent set, so that they may be interpreted as such on the receiving side.

2.5 DDS Domains and DomainParticipants

You may have several independent DCPS applications all running on the same set of computers. You may want to isolate one (or more) of those applications so that it isn't affected by the others. To address this issue, DCPS has a concept called *DDS domains*.

DDS domains represent logical, isolated, communication networks. Multiple applications running on the same set of hosts on different DDS domains are completely isolated from each other (even if they are on the same machine). *DataWriters* and *DataReaders* belonging to different DDS domains will never exchange data.

Applications that want to exchange data using DCPS must belong to the same DDS domain. To belong to a DDS domain, DCPS APIs are used to configure and create a *DomainParticipant* with a specific *Domain Index*. DDS domains are differentiated by the *domain index* (an integer value). Applications that have created *DomainParticipants* with the same *domain index* belong to the same DDS domain. *DomainParticipants* own *Topics*, *Publishers*, and *Subscribers*, which in turn owns *DataWriters* and *DataReaders*. Thus all DCPS *Entities* belong to a specific DDS domain.

An application may belong to multiple DDS domains simultaneously by creating multiple *DomainParticipants* with different domain indices. However, *Publishers/DataWriters* and *Subscribers/DataReaders* only belong to the DDS domain in which they were created.

As mentioned before, multiple DDS domains may be used for application isolation, which is useful when you are testing applications using computers on the same network or even the same computers. By assigning each user different domains, one can guarantee that the data produced by one user's application won't accidentally be received by another. In addition, DDS domains may be a way to scale and construct larger systems that are composed of multi-node subsystems. Each subsystem would use an internal DDS domain for intra-system communications and an external DDS domain to connect to other subsystems.

For more information, see [Working with DDS Domains \(Section Chapter 8 on page 536\)](#).

2.6 Quality of Service (QoS)

The publish-subscribe approach to distributed communications is a generic mechanism that can be employed by many different types of systems. The DCPS model described here extends the publish-subscribe model to address the needs of real-time, data-critical applications. It provides standardized mechanisms, known as Quality of Service Policies, that allow application developers to configure how communications occur, to limit resources used by the middleware, to detect system incompatibilities and setup error handling routines.

2.6.1 Controlling Behavior with Quality of Service (QoS) Policies

QoS Policies control many aspects of how and when data is distributed between applications. The overall QoS of the DCPS system is made up of the individual QoS Policies for each DCPS *Entity*. There are QoS Policies for *Topics*, *DataWriters*, *Publishers*, *DataReaders*, *Subscribers*, and *DomainParticipants*.

On the publishing side, the QoS of each *Topic*, the *Topic's DataWriter*, and the *DataWriter's Publisher* all play a part in controlling how and when DDS samples are sent to the middleware. Similarly, the QoS of the *Topic*, the *Topic's DataReader*, and the *DataReader's Subscriber* control behavior on the subscribing side.

Users will employ QoS Policies to control a variety of behaviors. For example, the DEADLINE policy sets up expectations of how often a *DataReader* expects to see DDS samples. The OWNERSHIP and OWNERSHIP_STRENGTH policy are used together to configure and arbitrate whose data is passed to the *DataReader* when there are multiple *DataWriters* for the same instance of a *Topic*. The HISTORY policy specifies whether a *DataWriter* should save old data to send to new subscriptions that join the network later. Many other policies exist and they are presented in [QoS Policies \(Section 4.2 on page 162\)](#).

Some QoS Policies represent “contracts” between publications and subscriptions. For communications to take place properly, the QoS Policies set on the *DataWriter* side must be compatible with corresponding policies set on the *DataReader* side.

For example, the RELIABILITY policy is set by the *DataWriter* to state whether it is configured to send data reliably to *DataReaders*. Because it takes additional resources to send data reliably, some *DataWriters* may only support a best-effort level of reliability. This implies that for those *DataWriters*, Connex DDS will not spend additional effort to make sure that the data sent is received by *DataReaders* or resend any lost data. However, for certain applications, it could be imperative that their *DataReaders* receive every

piece of data with total reliability. Running a system where the *DataWriters* have not been configured to support the *DataReaders* could lead to erratic failures.

To address this issue, and yet keep the publications and subscriptions as decoupled as possible, DCPS provides a way to detect and notify when QoS Policies set by *DataWriters* and *DataReaders* are incompatible. DCPS employs a pattern known as RxO (Requested versus Offered). The *DataReader* sets a “requested” value for a particular QoS Policy. The *DataWriter* sets an “offered” value for that QoS Policy. When Connex DDS matches a *DataReader* to a *DataWriter*, QoS Policies are checked to make sure that all requested values can be supported by the offered values.

Note that not all QoS Policies are constrained by the RxO pattern. For example, it does not make sense to compare policies that affect only the *DataWriter* but not the *DataReader* or vice versa.

If the *DataWriter* cannot satisfy the requested QoS Policies of a *DataReader*, Connex DDS will not connect the two DDS entities and will notify the applications on each side of the incompatibility if so configured.

For example, a *DataReader* sets its DEADLINE QoS to 4 seconds—that is, the *DataReader* is *requesting* that it receive new data at least every 4 seconds.

In one application, the *DataWriter* sets its DEADLINE QoS to 2 seconds—that is, the *DataWriter* is committing to sending data at least every 2 seconds. This writer can satisfy the request of the reader, and thus, Connex DDS will pass the data sent from the writer to the reader.

In another application, the *DataWriter* sets its DEADLINE QoS to 5 seconds. It only commits to sending data at 5 second intervals. This will not satisfy the request of the *DataReader*. Connex DDS will flag this incompatibility by calling user-installed listeners in both *DataWriter* and *DataReader* applications and not pass data from the writer to the reader.

For a summary of the QoS Policies supported by Connex DDS, see [QoS Policies \(Section 4.2 on page 162\)](#).

2.7 Application Discovery

The DCPS model provides anonymous, transparent, many-to-many communications. Each time an application sends a DDS sample of a particular *Topic*, the middleware distributes the DDS sample to all the applications that want that *Topic*. The publishing application does not need to specify how many applications receive the *Topic*, nor where those applications are located. Similarly, *subscribing applications* do not specify the location of the publications. In addition, new publications and *subscriptions* of the *Topic* can appear at any time, and the middleware will automatically interconnect them.

So how is this all done? Ultimately, in each application for each publication, Connex DDS must keep a list of applications that have subscribed to the same *Topic*, nodes on which they are located, and some additional QoS parameters that control how the data is sent. Also, Connex DDS must keep a list of applications and publications for each of the *Topics* to which the application has subscribed.

This propagation of this information (the existence of publications and subscriptions and associated QoS) between applications by Connex DDS is known as the *discovery* process. While the DDS (DCPS) standard does not specify how discovery occurs, Connex DDS uses a standard protocol RTPS for both discovery and formatting on-the-wire packets.

When a *DomainParticipant* is created, Connex DDS sends out packets on the network to announce its existence. When an application finds out that another application belongs to the same DDS domain, then it will exchange information about its existing publications and subscriptions and associated QoS with the other application. As new *DataWriters* and *DataReaders* are created, this information is sent to known applications.

The *Discovery* process is entirely configurable by the user and is discussed extensively in [Discovery \(Section Chapter 14 on page 709\)](#).

Part 2: Core Concepts

This section includes:

- [Data Types and DDS Data Samples \(Section Chapter 3 on page 23\)](#)
- [DDS Entities \(Section Chapter 4 on page 151\)](#)
- [Topics \(Section Chapter 5 on page 200\)](#)
- [Sending Data \(Section Chapter 6 on page 242\)](#)
- [Receiving Data \(Section Chapter 7 on page 437\)](#)
- [Working with DDS Domains \(Section Chapter 8 on page 536\)](#)
- [Building Applications \(Section Chapter 9 on page 622\)](#)

Chapter 3 Data Types and DDS Data Samples

How data is stored or laid out in memory can vary from language to language, compiler to compiler, operating system to operating system, and processor to processor. This combination of language/compiler/operating system/processor is called a *platform*. Any modern middleware must be able to take data from one specific platform (say *C/gcc.3.2.2/Solaris/Sparc*) and transparently deliver it to another (for example, *Java/JDK 1.6/Windows/Pentium*). This process is commonly called serialization/deserialization, or marshalling/demarshalling.

Messaging products have typically taken one of two approaches to this problem:

1. **Do nothing.** Messages consist only of opaque streams of bytes. The JMS *BytesMessage* is an example of this approach.
2. **Send everything, every time.** Self-describing messages are at the opposite extreme, embedding full reflective information, including data types and field names, with each message. The JMS *MapMessage* and the messages in TIBCO Rendezvous are examples of this approach.

The “do nothing” approach is lightweight on its surface but forces you, the user of the middleware API, to consider all data encoding, alignment, and padding issues. The “send everything” alternative results in large amounts of redundant information being sent with every packet, impacting performance.

Connext DDS takes an intermediate approach. Just as objects in your application program belong to some data type, DDS data samples sent on the same Connext DDS topic share a data type. This type defines the fields that exist in the DDS data samples and what their constituent types are. The middleware stores and propagates this meta-information separately from the individual DDS data samples, allowing it to propagate DDS samples efficiently while handling byte ordering and alignment issues for you.

To publish and/or subscribe to data with Connext DDS, you will carry out the following steps:

1. Select a type to describe your data.

You have a number of choices. You can choose one of these options, or you can mix and match them.

- Use a built-in type provided by the middleware.

This option may be sufficient if your data typing needs are very simple. If your data is highly structured, or you need to be able to examine fields within that data for filtering or other purposes, this option may not be appropriate. The built-in types are described in [Built-in Data Types \(Section 3.2 on page 30\)](#).

- Use the *RTI Code Generator* to define a type at compile-time using a language-independent description language.

Code generation offers two strong benefits not available with dynamic type definition: (1) it allows you to share type definitions across programming languages, and (2) because the structure of the type is known at compile time, it provides rigorous static type safety.

The *RTI Code Generator* accepts input the following formats:

- **OMG IDL.** This format is a standard component of both the DDS and CORBA specifications. It describes data types with a C++-like syntax. This format is described in [Creating User Data Types with IDL \(Section 3.3 on page 69\)](#).
- **XML in a DDS-specific format.** This XML format is terser, and therefore easier to read and write by hand, than an XSD file. It offers the general benefits of XML-extensibility and ease of integration, while fully supporting DDS-specific data types and concepts. This format is described in [Creating User Data Types with Extensible Markup Language \(XML\) \(Section 3.4 on page 121\)](#).
- Define a type programmatically at run time.

This method may be appropriate for applications with dynamic data description needs: applications for which types change frequently or cannot be known ahead of time. It is described in [Defining New Types \(Section 3.8.2 on page 141\)](#).

2. Register your type with a logical name.

If you've chosen to use a built-in type instead of defining your own, you can omit this step; the middleware pre-registers the built-in types for you.

This step is described in the [Defining New Types \(Section 3.8.2 on page 141\)](#).

3. Create a *Topic* using the type name you previously registered.

If you've chosen to use a built-in type instead of defining your own, you will use the API constant corresponding to that type's name.

Creating and working with *Topics* is discussed in [Topics \(Section Chapter 5 on page 200\)](#).

4. Create one or more *DataWriters* to publish your data and one or more *DataReaders* to subscribe to it.

The concrete types of these objects depend on the concrete data type you've selected, in order to provide you with a measure of type safety.

Creating and working with *DataWriters* and *DataReaders* are described in [Sending Data \(Section Chapter 6 on page 242\)](#) and [Receiving Data \(Section Chapter 7 on page 437\)](#), respectively.

Whether publishing or subscribing to data, you will need to know how to create and delete DDS data samples and how to get and set their fields. These tasks are described in [Working with DDS Data Samples \(Section 3.9 on page 145\)](#).

This section describes:

3.1 Introduction to the Type System

A *user data type* is any custom type that your application defines for use with Connex DDS. It may be a structure, a union, a value type, an enumeration, or a typedef (or language equivalents).

Your application can have any number of user data types. They can be composed of any of the primitive data types listed below or of other user data types.

Only structures, unions, and value types may be read and written directly by Connex DDS; enums, typedefs, and primitive types must be contained within a structure, union, or value type. In order for a *DataReader* and *DataWriter* to communicate with each other, the data types associated with their respective Topic definitions must be identical.

- octet, char, wchar
- short, unsigned short
- long, unsigned long
- long long, unsigned long long
- float
- double, long double
- boolean
- enum (with or without explicit values)
- bounded and unbounded string and wstring

The following type-building constructs are also supported:

- module (also called a *package* or *namespace*)
- pointer
- array of primitive or user type elements
- bounded/unbounded sequence of elements¹—a *sequence* is a variable-length ordered collection, such as a vector or list
- typedef
- bitfield²
- union
- struct
- value type, a complex type that supports inheritance and other object-oriented features

To use a data type with Connex DDS, you must define that type in a way the middleware understands and then register the type with the middleware. These steps allow Connex DDS to serialize, deserialize, and otherwise operate on specific types. They will be described in detail in the following sections.

3.1.1 Sequences

A sequence contains an ordered collection of elements that are all of the same type. The operations supported in the sequence are documented in the API Reference HTML documentation, which is available for all supported programming languages (select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **Sequence Support**).

Java sequences implement the `java.util.List` interface from the standard Collections framework.

In the Modern C++ API a sequences of type `T` maps to the type `dds::core::vector<T>`. This type is similar to `std::vector<T>`.

Elements in a sequence are accessed with their index, just like elements in an array. Indices start at zero in all APIs except Ada. In Ada, indices start at 1. Unlike arrays, however, sequences can grow in size. A sequence has two sizes associated with it: a physical size (the "maximum") and a logical size (the "length"). The physical size indicates how many elements are currently allocated by the sequence to hold; the logical size indicates how many valid elements the sequence actually holds. The length can vary from zero up to the maximum. Elements cannot be accessed at indices beyond the current length.

¹Sequences of sequences are not supported directly. To work around this constraint, typedef the inner sequence and form a sequence of that new type.

²Data types containing bitfield members are not supported by DynamicData. [RTI Bug # 12638]

A sequence may be declared as bounded or unbounded. A sequence's "bound" is the maximum number of elements that the sequence can contain at any one time. A finite bound is very important because it allows Connex DDS to preallocate buffers to hold serialized and deserialized samples of your types; these buffers are used when communicating with other nodes in your distributed system. If a sequence has no bound, Connex DDS will not know how large to allocate its buffers and will therefore have to allocate them on the fly as individual samples are read and written—impacting the latency and determinism of your application.

By default, any unbounded sequences found in an IDL file will be given a default bound of 100 elements. This default value can be overwritten using the *RTI Code Generator's* **-sequenceSize** command-line argument (see the *RTI Code Generator User's Manual*).

When using C, C++, or .NET, you can change the default behavior and used truly unbounded sequences by using *RTI Code Generator's* **-unboundedSupport** command-line argument. When using this option, the generated code will deserialize incoming samples by dynamically allocating and deallocating memory to accommodate the actual size of the sequences.

Unbounded built-in types are only supported in the C, C++, Java and .NET APIs

To configure unbounded support for code generated with `rtiddsgen -unboundedSupport`:

1. Use these threshold QoS properties:
 - **dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size** on the *DataWriter*
 - **dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size** on the *DataReader* (only if keyed)
2. Set the QoS value **reader_resource_limits.dynamically_allocate_fragmented_samples** on the *DataReader* to true.
3. For the Java API, also set these properties accordingly for the Java serialization buffer:
 - **dds.data_writer.history.memory_manager.java_stream.min_size**
 - **dds.data_writer.history.memory_manager.java_stream.trim_to_size**
 - **dds.data_reader.history.memory_manager.java_stream.min_size**
 - **dds.data_reader.history.memory_manager.java_stream.trim_to_size**

See also:

- [Unbounded Built-in Types \(Section 3.2.7.2 on page 67\)](#)
- [Writer-Side Memory Management when Using Java \(Section 20.1.3 on page 851\)](#)
- [Reader-Side Memory Management when Using Java \(Section 20.2.2 on page 856\)](#)

3.1.2 Strings and Wide Strings

Connex DDS supports both strings consisting of single-byte characters (the IDL string type) and strings consisting of wide characters (IDL wstring). The wide characters supported by Connex DDS are four bytes long, large enough to store not only two-byte Unicode/UTF16 characters but also UTF32 characters.

Like sequences, strings may be bounded or unbounded. A string's "bound" is its maximum length (not counting the trailing NULL character in C and C++).

In the Modern C++ API strings map to the type **dds::core::string**, similar to **std::string**.

By default, any unbounded string found in an IDL file will be given a default bound of 255 elements. This default value can be overwritten using the *RTI Code Generator's* **-stringSize** command-line argument (see the *RTI Code Generator User's Manual*).

In C, C++, and .NET, you can change the default behavior and used truly unbounded string by using *RTI Code Generator's* **-unboundedSupport** command-line argument. When using this option, the generated code will deserialize incoming samples by dynamically allocating and deallocating memory to accommodate the actual size of the strings.

Unbounded built-in types are only supported in the C, C++, Java and .NET APIs

To configure unbounded support for built-in types:

1. Set the properties **dds.builtin_type.*.max_size** and **dds.builtin_type.*.alloc_size** to 2,147,483,647.
2. Use these threshold QoS properties:
 - **dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size** on the *DataWriter*
 - **dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size** on the *DataReader* (only if keyed)
3. Set the QoS value **reader_resource_limits.dynamically_allocate_fragmented_samples** on the *DataReader* to true.
4. For the Java API, also set these properties accordingly for the Java serialization buffer:
 - **dds.data_writer.history.memory_manager.java_stream.min_size**
 - **dds.data_writer.history.memory_manager.java_stream.trim_to_size**
 - **dds.data_reader.history.memory_manager.java_stream.min_size**
 - **dds.data_reader.history.memory_manager.java_stream.trim_to_size**

See also:

- [Unbounded Built-in Types \(Section 3.2.7.2 on page 67\)](#)
- [Writer-Side Memory Management when Using Java \(Section 20.1.3 on page 851\)](#)
- [Reader-Side Memory Management when Using Java \(Section 20.2.2 on page 856\)](#)

3.1.3 Introduction to TypeCode

Type schemas—the names and definitions of a type and its fields—are represented by TypeCode objects (known as DynamicType in the Modern C++ API). A type code value consists of a type code kind (see the **TCKind** enumeration below) and a list of members. For compound types like structs and arrays, this list will recursively include one or more type code values.

```
enum TCKind {
    TK_NULL,
    TK_SHORT,
    TK_LONG,
    TK_USHORT,
    TK_ULONG,
    TK_FLOAT,
    TK_DOUBLE,
    TK_BOOLEAN,
    TK_CHAR,
    TK_OCTET,
    TK_STRUCT,
    TK_UNION,
    TK_ENUM,
    TK_STRING,
    TK_SEQUENCE,
    TK_ARRAY,
    TK_ALIAS,
    TK_LONGLONG,
    TK_ULONGLONG,
    TK_LONGDOUBLE,
    TK_WCHAR,
    TK_WSTRING,
    TK_VALUE
}
```

Type codes unambiguously match type representations and provide a more reliable test than comparing the string type names.

The **TypeCode** class, modeled after the corresponding CORBA API, provides access to type-code information. For details on the available operations for the **TypeCode** class, see the API Reference HTML documentation, which is available for all supported programming languages (select **Modules, RTI Connex DDS API Reference, Topic Module, Type Code Support** or, for the Modern C++ API select **Modules, RTI Connex DDS API Reference, Infrastructure Module, DynamicType and DynamicData**).

Note: Type-code support must be enabled if you are going to use [ContentFilteredTopics \(Section 5.4 on page 212\)](#) with the default SQL filter. You may disable type codes and use a custom filter, as described in [Creating ContentFilteredTopics \(Section 5.4.3 on page 214\)](#).

3.1.3.1 Sending TypeCodes on the Network

In addition to being used locally, serialized type codes are typically published automatically during discovery as part of the built-in topics for publications and subscriptions. See [Built-in DataReaders \(Section 16.2 on page 773\)](#). This allows applications to publish or subscribe to topics of arbitrary types. This functionality is useful for generic system monitoring tools like the *rtiddsspy* debug tool (see the API Reference HTML documentation).

Note: In the C, Traditional C++, Java and .NET APIs Type codes are not cached by Connex DDS upon receipt and are therefore not available from the built-in data returned by the *DataWriter's* `get_matched_subscription_data()` operation or the *DataReader's* `get_matched_publication_data()` operation; in the Modern C++ API they are available.

If your data type has an especially complex type code, you may need to increase the value of the `type_code_max_serialized_length` field in the *DomainParticipant's* [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#). Or, to prevent the propagation of type codes altogether, you can set this value to zero (0). Be aware that some features of monitoring tools, as well as some features of the middleware itself (such as *ContentFilteredTopics*) will not work correctly if you disable *TypeCode* propagation.

3.2 Built-in Data Types

Connex DDS provides a set of standard types that are built into the middleware. These types can be used immediately; they do not require you to write IDL, use *RTI Code Generator (rtiddsgen)* (see [Using RTI Code Generator \(rtiddsgen\) \(Section 3.6 on page 138\)](#)), or use the dynamic type API (see [Managing Memory for Built-in Types \(Section 3.2.7 on page 62\)](#)).

The supported built-in types are **String**, **KeyedString**, **Octets**, and **KeyedOctets**. (The latter two types are called **Bytes** and **KeyedBytes**, respectively, on Java and .NET platforms.)

The built-in type API is located under the DDS namespace in Traditional C++ and .NET. For Java, the API is contained inside the package `com.rti.dds.type.builtin`. In the Modern C++ API they are located in the `dds::core` namespace.

Built-in data types are discussed in the following sections:

3.2.1 Registering Built-in Types

By default, the built-in types are automatically registered when a *DomainParticipant* is created. You can change this behavior by setting the *DomainParticipant's* `dds.builtin_type.auto_register` property to 0

(false) using the [PROPERTY QosPolicy \(DDS Extension\)](#) (Section 6.5.17 on page 394).

3.2.2 Creating Topics for Built-in Types

To create a topic for a built-in type, just use the standard *DomainParticipant* operations, `create_topic()` or `create_topic_with_profile()` (see [Creating Topics \(Section 5.1.1 on page 202\)](#)); for the `type_name` parameter, use the value returned by the `get_type_name()` operation, listed below for each API.

Note: In the following examples, you will see the sentinel "`<BuiltinType>`."

For C and Traditional C++: `<BuiltinType>` = String, KeyedString, Octets or KeyedOctets

For Java and .NET¹: `<BuiltinType>` = String, KeyedString, Bytes or KeyedBytes

C API:

```
const char* DDS_<BuiltinType>TypeSupport_get_type_name();
```

Traditional C++ API with namespace:

```
const char* DDS::<BuiltinType>TypeSupport::get_type_name();
```

Traditional C++ API without namespace:

```
const char* DDS<BuiltinType>TypeSupport::get_type_name();
```

C++/CLI API:

```
System::String^ DDS_<BuiltinType>TypeSupport::get_type_name();
```

C# API:

```
System.String DDS.<BuiltinType>TypeSupport.get_type_name();
```

Java API:

```
String
com.rti.dds.type.builtin.<BuiltinType>TypeSupport.get_type_name();
```

(This step is not required in the Modern C++ API)

3.2.2.1 Topic Creation Examples

For simplicity, error handling is not shown in the following examples.

C Example:

```
DDS_Topic * topic = NULL;
/* Create a builtin type Topic */
```

¹RTI Connext DDS .NET language binding is currently supported for C# and C++/CLI.

```
topic = DDS_DomainParticipant_create_topic(
    participant, "StringTopic",
    DDS_StringTypeSupport_get_type_name(),
    &DDS_TOPIC_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);
```

Traditional C++ Example with namespaces:¹

```
using namespace DDS;
...
/* Create a String builtin type Topic */
Topic * topic = participant->create_topic(
    "StringTopic", StringTypeSupport::get_type_name(),
    DDS_TOPIC_QOS_DEFAULT, NULL, DDS_STATUS_MASK_NONE);
```

Modern C++ Example:

```
dds::topic::Topic<dds::core::StringTopicType> topic(participant, "StringTopic");
```

C++/CLI Example:

```
using namespace DDS;
...
/* Create a builtin type Topic */
Topic^ topic = participant->create_topic(
    "StringTopic", StringTypeSupport::get_type_name(),
    DomainParticipant::TOPIC_QOS_DEFAULT,
    nullptr, StatusMask::STATUS_MASK_NONE);
```

C# Example:

```
using namespace DDS;
... /*
Create a builtin type Topic */
Topic topic = participant.create_topic(
    "StringTopic", StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusMask.STATUS_MASK_NONE);
```

Java Example:

```
import com.rti.dds.type.builtin.*;
...
/* Create a builtin type Topic */
Topic topic = participant.create_topic(
    "StringTopic", StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusKind.STATUS_MASK_NONE);
```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

3.2.3 String Built-in Type

The String built-in type is represented by a NULL-terminated character array (`char *`) in C and C++ and an immutable String object in Java and .NET¹. This type can be used to publish and subscribe to a single string.

3.2.3.1 Creating and Deleting Strings

In C and C++, Connex DDS provides a set of operations to create (**`DDS::String_alloc()`**), destroy (**`DDS::String_free()`**), and clone strings (**`DDS::String_dup()`**). Select **Modules, RTI Connex DDS API Reference, Infrastructure Module, String support** in the API Reference HTML documentation, which is available for all supported programming languages.

Memory Considerations in Copy Operations:

When the read/take operations that take a sequence of strings as a parameter are used in copy mode, Connex DDS allocates the memory for the string elements in the sequence if they are initialized to NULL.

If the elements are not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory associated with the elements is reallocated with every DDS sample, because strings are immutable objects.
- In C and C++, the memory associated with the elements must be large enough to hold the received data. Insufficient memory may result in crashes.

When **`take_next_sample()`** and **`read_next_sample()`** are called in C and C++, you must make sure that the input string has enough memory to hold the received data. Insufficient memory may result in crashes.

3.2.3.2 String DataWriter

The string *DataWriter* API matches the standard *DataWriter* API (see [Using a Type-Specific DataWriter \(FooDataWriter\) \(Section 6.3.7 on page 281\)](#)). There are no extensions.

The following examples show how to write simple strings with a string built-in type *DataWriter*. For simplicity, error handling is not shown.

C Example:

```
DDS_StringDataWriter * stringWriter = ... ;
```

¹RTI Connex DDS .NET language binding is currently supported for C# and C++/CLI.

```

DDS_ReturnCode_t retCode; char * str = NULL;
/* Write some data */
retCode = DDS_StringDataWriter_write(
    stringWriter, "Hello World!", &DDS_HANDLE_NIL);
str = DDS_String_dup("Hello World!");
retCode = DDS_StringDataWriter_write(
    stringWriter, str, &DDS_HANDLE_NIL);
DDS_String_free(str);

```

Traditional C++ Example with namespaces:¹

```

#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
StringDataWriter * stringWriter = ... ;
/* Write some data */
ReturnCode_t retCode = stringWriter->write(
    "Hello World!", HANDLE_NIL);
char * str = DDS::String_dup("Hello World!");
retCode = stringWriter->write(str, HANDLE_NIL);
DDS::String_free(str);

```

Modern C++ Example:

```

dds::pub::DataWriter<dds::core::StringTopicType> string_writer(
    participant, string_topic);
string_writer.write("Hello World!");
dds::core::string str = "Hello World!";
string_writer.write(str);

```

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
StringDataWriter^ stringWriter = ... ;
/* Write some data */
stringWriter->write(
    "Hello World!", InstanceHandle_t::HANDLE_NIL);
String^ str = "Hello World!";
stringWriter->write(
    str, InstanceHandle_t::HANDLE_NIL);

```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

C# Example:

```

using System;
using DDS;
...
StringDataWriter stringWriter = ... ;
/* Write some data */
stringWriter.write(
    "Hello World!", InstanceHandle_t.HANDLE_NIL);
String str = "Hello World!";
stringWriter.write(
    str, InstanceHandle_t.HANDLE_NIL);

```

Java Example:

```

import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
StringDataWriter stringWriter = ... ;
/* Write some data */
stringWriter.write(
    "Hello World!", InstanceHandle_t.HANDLE_NIL);
String str = "Hello World!";
stringWriter.write(
    str, InstanceHandle_t.HANDLE_NIL);

```

3.2.3.3 String DataReader

The string *DataReader* API matches the standard *DataReader* API (see [Using a Type-Specific DataReader \(FooDataReader\)](#) (Section 7.4.1 on page 491)). There are no extensions.

The following examples show how to read simple strings with a string built-in type *DataReader*. For simplicity, error handling is not shown.

C Example:

```

struct DDS_StringSeq dataSeq =
    DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq =
    DDS_SEQUENCE_INITIALIZER;
DDS_StringDataReader * stringReader = ... ;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_StringDataReader_take(
    stringReader, &dataSeq,
    &infoSeq, DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE,

```



```

        DDS_ANY_VIEW_STATE,
        DDS_ANY_INSTANCE_STATE);
for (i = 0; i < DDS_StringSeq_get_length(&data_seq);
    ++i) {
    if (DDS_SampleInfoSeq_get_reference(
        &info_seq, i)->valid_data) {
        DDS_StringTypeSupport_print_data(
            DDS_StringSeq_get(&data_seq, i));
    }
}
/* Return loan */
retCode = DDS_StringDataReader_return_loan(
    stringReader, &data_seq, &info_seq);

```

Traditional C++ Example with namespaces:¹

```

#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
StringSeq dataSeq;
SampleInfoSeq infoSeq;
StringDataReader * stringReader = ... ;
/* Take a print the data */
ReturnCode_t retCode = stringReader->take(
    dataSeq, infoSeq,
    LENGTH_UNLIMITED,
    ANY_SAMPLE_STATE,
    ANY_VIEW_STATE,
    ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        StringTypeSupport::print_data(dataSeq[i]);
    }
}
/* Return loan */
retCode = stringReader->return_loan(
    dataSeq, infoSeq);

```

Modern C++ Example:

```

using namespace dds::core;
using namespace dds::sub;
DataReader<StringTopicType> string_reader(
    participant, string_topic);

```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

```

LoanedSamples<StringTopicType> samples =
    string_reader.take();
for (auto sample : samples) {
    if (sample.info().valid()) {
        std::cout << sample.data() << std::endl;
    }
}

```

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
StringSeq^ dataSeq = gcnew StringSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
StringDataReader^ stringReader = ... ;
/* Take and print the data */
stringReader->take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq->get_at(i)->valid_data) {
        StringTypeSupport::print_data(
            dataSeq->get_at(i));
    }
}
/* Return loan */
stringReader->return_loan(dataSeq, infoSeq);

```

C# Example:

```

using System;
using DDS;
...
StringSeq dataSeq = new StringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
StringDataReader stringReader = ... ;
/* Take and print the data */
stringReader.take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq.get_at(i).valid_data) {

```

```

        StringTypeSupport.print_data(
            dataSeq.get_at(i));
    }
}

```

Java Example:

```

import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
StringSeq dataSeq = new StringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
StringDataReader stringReader = ... ;
/* Take and print the data */
stringReader.take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println(
            (String)dataSeq.get(i));
    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);

```

3.2.4 KeyedString Built-in Type

The Keyed String built-in type is represented by a (key, value) pair, where key and value are strings. This type can be used to publish and subscribe to keyed strings. The language specific representations of the type are as follows:

C/Traditional C++ Representation (without namespaces):

```

struct DDS_KeyedString {
    char * key;
    char * value;
};

```

Modern C++ Representation:

```

class dds::core::KeyedStringTopicType {
public:

```

```

dds::core::string& key();
dds::core::string& value();
// ... see API documentation for full definition
};

```

C++/CLI Representation:

```

namespace DDS {
    public ref struct KeyedString: {
        public:
            System::String^ key;
            System::String^ value;
            ...
    };
};

```

C# Representation:

```

namespace DDS {
    public class KeyedString {
        public System.String key;
        public System.String value;
    };
};

```

Java Representation:

```

namespace DDS {
    public class KeyedString {
        public System.String key;
        public System.String value;
    };
};

```

3.2.4.1 Creating and Deleting Keyed Strings

Connex DDS provides a set of constructors/destructors to create/destroy Keyed Strings. For details, see the API Reference HTML documentation, which is available for all supported programming languages (select **Modules**, **RTI Connex DDS API Reference**, **Topic Module**, **Built-in Types**).

If you want to manipulate the memory of the fields 'value' and 'key' in the KeyedString struct in C/C++, use the operations **DDS::String_alloc()**, **DDS::String_dup()**, and **DDS::String_free()**, as described in the API Reference HTML documentation (select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **String Support**).

3.2.4.2 Keyed String DataWriter

The keyed string *DataWriter* API is extended with the following methods (in addition to the standard methods described in [Using a Type-Specific DataWriter \(FooDataWriter\) \(Section 6.3.7 on page 281\)](#)):

```

DDS::ReturnCode_t
DDS::KeyedStringDataWriter::dispose (
    const char* key,
    const DDS::InstanceHandle_t* instance_handle);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::dispose_w_timestamp (
    const char* key,
    const DDS::InstanceHandle_t* instance_handle,
    const struct DDS::Time_t* source_timestamp);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::get_key_value (
    char * key,
    const DDS::InstanceHandle_t* handle);
DDS::InstanceHandle_t
DDS::KeyedStringDataWriter::lookup_instance (
    const char * key);
DDS::InstanceHandle_t
DDS::KeyedStringDataWriter::register_instance (
    const char* key);
DDS::InstanceHandle_t
DDS_KeyedStringDataWriter::register_instance_w_timestamp (
    const char * key,
    const struct DDS_Time_t* source_timestamp);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::unregister_instance (
    const char * key,
    const DDS::InstanceHandle_t* handle);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::unregister_instance_w_timestamp (
    const char* key,
    const DDS::InstanceHandle_t* handle,
    const struct DDS::Time_t* source_timestamp);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::write (
    const char * key,
    const char * str,
    const DDS::InstanceHandle_t* handle);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::write_w_timestamp (
    const char * key,
    const char * str,
    const DDS::InstanceHandle_t* handle,
    const struct DDS::Time_t* source_timestamp);

```

These operations are introduced to provide maximum flexibility in the format of the input parameters for the write and instance management operations. For additional information and a complete description of the operations, see the API Reference HTML documentation, which is available for all supported programming languages.

The following examples show how to write keyed strings using a keyed string built-in type *DataWriter* and some of the extended APIs. For simplicity, error handling is not shown.

C Example:

```
DDS_KeyedStringDataWriter * stringWriter = ... ;
DDS_ReturnCode_t retCode;
struct DDS_KeyedString * keyedStr = NULL;
char * str = NULL;
/* Write some data using the KeyedString structure */
keyedStr = DDS_KeyedString_new(255, 255);
strcpy(keyedStr->key, "Key 1");
strcpy(keyedStr->value, "Value 1");
retCode = DDS_KeyedStringDataWriter_write_string_w_key(
    stringWriter, keyedStr,
    &DDS_HANDLE_NIL);
DDS_KeyedString_delete(keyedStr);
/* Write some data using individual strings */
retCode = DDS_KeyedStringDataWriter_write_string_w_key(
    stringWriter, "Key 1",
    "Value 1", &DDS_HANDLE_NIL);
str = DDS_String_dup("Value 2");
retCode = DDS_KeyedStringDataWriter_write_string_w_key(
    stringWriter, "Key 1",
    str, &DDS_HANDLE_NIL);
DDS_String_free(str);
```

C++ Example with Namespaces:¹

```
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedStringDataWriter * stringWriter = ... ;
/* Write some data using the KeyedString */
KeyedString * keyedStr = new KeyedString(255, 255);
strcpy(keyedStr->key, "Key 1");
strcpy(keyedStr->value, "Value 1");
ReturnCode_t retCode = stringWriter->write(
    keyedStr, HANDLE_NIL);
delete keyedStr;
```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
KeyedStringDataWriter^ stringWriter = ... ;
/* Write some data using the KeyedString */
KeyedString^ keyedStr = gcnew KeyedString();
keyedStr->key = "Key 1";
keyedStr->value = "Value 1";
stringWriter->write(
    keyedStr, InstanceHandle_t::HANDLE_NIL);
/* Write some data using individual strings */
stringWriter->write
    "Key 1", "Value 1",
    InstanceHandle_t::HANDLE_NIL);
String^ str = "Value 2";
stringWriter->write(
    "Key 1", str,
    InstanceHandle_t::HANDLE_NIL);

```

C# Example:

```

using System;
using DDS;
...
KeyedStringDataWriter stringWriter = ... ;
/* Write some data using the KeyedString */
KeyedString keyedStr = new KeyedString();
keyedStr.key = "Key 1";
keyedStr.value = "Value 1";
stringWriter.write(
    keyedStr, InstanceHandle_t.HANDLE_NIL);
/* Write some data using individual strings */
stringWriter.write(
    "Key 1", "Value 1",
    InstanceHandle_t.HANDLE_NIL);
String str = "Value 2";
stringWriter.write(
    "Key 1", str,
    InstanceHandle_t.HANDLE_NIL);

```

Java Example:

```

import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
KeyedStringDataWriter stringWriter = ... ;

```

```

/* Write some data using the KeyedString */
KeyedString keyedStr = new KeyedString();
keyedStr.key = "Key 1";
keyedStr.value = "Value 1";
stringWriter.write(
    keyedStr, InstanceHandle_t.HANDLE_NIL);
/* Write some data using individual strings */
stringWriter.write(
    "Key 1", "Value 1",
    InstanceHandle_t.HANDLE_NIL);
String str = "Value 2";
stringWriter.write(
    "Key 1", str,
    InstanceHandle_t.HANDLE_NIL);

```

3.2.4.3 Keyed String DataReader

The `KeyedString DataReader` API is extended with the following operations (in addition to the standard methods described in [Using a Type-Specific DataReader \(FooDataReader\)](#) (Section 7.4.1 on page 491)):

```

DDS::ReturnCode_t
DDS::KeyedStringDataReader::get_key_value(
    char * key,
    const DDS::InstanceHandle_t* handle);
DDS::InstanceHandle_t
DDS::KeyedStringDataReader::lookup_instance(
    const char * key);

```

For additional information and a complete description of these operations in all supported languages, see the API Reference HTML documentation, which is available for all supported programming languages.

Memory considerations in copy operations:

For read/take operations with copy semantics, such as `read_next_sample()` and `take_next_sample()`, Connex DDS allocates memory for the fields `'value'` and `'key'` if they are initialized to NULL.

If the fields are not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory associated to the fields `'value'` and `'key'` will be reallocated with every DDS sample.
- In C and C++, the memory associated with the fields `'value'` and `'key'` must be large enough to hold the received data. Insufficient memory may result in crashes.

The following examples show how to read keyed strings with a keyed string built-in type `DataReader`. For simplicity, error handling is not shown.

C Example:

```

struct DDS_KeyedStringSeq dataSeq =
    DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq =
    DDS_SEQUENCE_INITIALIZER;
DDS_KeyedKeyedStringDataReader * stringReader = ... ;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_KeyedStringDataReader_take(
    stringReader, &dataSeq,
    &infoSeq,
    DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);

for (i = 0;
    i < DDS_KeyedStringSeq_get_length(&data_seq);
    ++i) {
    if (DDS_SampleInfoSeq_get_reference(
        &info_seq, i)->valid_data) {
        DDS_KeyedStringTypeSupport_print_data(
            DDS_KeyedStringSeq_get_reference(&data_seq, i));
    }
}
/* Return loan */
retCode = DDS_KeyedStringDataReader_return_loan(
    stringReader, &data_seq, &info_seq);

```

C++ Example with Namespaces:¹

```

#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedStringSeq dataSeq;
SampleInfoSeq infoSeq;
KeyedStringDataReader * stringReader = ... ;
/* Take a print the data */
ReturnCode_t retCode = stringReader->take(
    dataSeq, infoSeq,
    LENGTH_UNLIMITED,
    ANY_SAMPLE_STATE,
    ANY_VIEW_STATE,
    ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {

```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

```

    if (infoSeq[i].valid_data) {
        KeyedStringTypeSupport::print_data(&dataSeq[i]);
    }
}
/* Return loan */
retCode = stringReader->return_loan(dataSeq, infoSeq);

```

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
KeyedStringSeq^ dataSeq = gcnew KeyedStringSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
KeyedStringDataReader^ stringReader = ... ;
/* Take and print the data */
stringReader->take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq->get_at(i)->valid_data) {
        KeyedStringTypeSupport::print_data(
            dataSeq->get_at(i));
    }
}
/* Return loan */
stringReader->return_loan(dataSeq, infoSeq);

```

C# Example:

```

using System;
using DDS;
...
KeyedStringSeq dataSeq = new KeyedStringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedStringDataReader stringReader = ... ;
/* Take and print the data */
stringReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq.get_at(i).valid_data) {
        KeyedStringTypeSupport.print_data(
            dataSeq.get_at(i));
    }
}

```

```

    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);

```

Java Example:

```

import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
KeyedStringSeq dataSeq = new KeyedStringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedStringDataReader stringReader = ... ;
/* Take and print the data */
stringReader.take(dataSeq, infoSeq,
                 ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
                 SampleStateKind.ANY_SAMPLE_STATE,
                 ViewStateKind.ANY_VIEW_STATE,
                 InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println((
            (KeyedString)dataSeq.get(i)).toString());
    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);

```

3.2.5 Octets Built-in Type

The octets built-in type is used to send sequences of octets. The language-specific representations are as follows:

C/Traditional C++ Representation (without Namespaces):

```

struct DDS_Octets {
    int length;
    unsigned char * value;
};

```

Modern C++ Representation:

```

class dds::core::BytesTopicType {
public:
    uint8_t& operator [] (uint32_t index);
    // ... see API documentation for full definition

```

```
};
```

C++/CLI Representation:

```
namespace DDS {
    public ref struct Bytes: {
        public:
            System::Int32 length;
            System::Int32 offset;
            array<System::Byte>^ value;
            ...
    };
};
```

C# Representation:

```
namespace DDS {
    public class Bytes {
        public System.Int32 length;
        public System.Int32 offset;
        public System.Byte[] value;
        ...
    };
};
```

Java Representation:

```
package com.rti.dds.type.builtin;
public class Bytes implements Copyable {
    public int length;
    public int offset;
    public byte[] value;
    ...
};
```

3.2.5.1 Creating and Deleting Octets

Connex DDS provides a set of constructors/destructors to create and destroy Octet objects. For details, see the API Reference HTML documentation, which is available for all supported programming languages (select **Modules**, **RTI Connex DDS API Reference**, **Topic Module**, **Built-in Types**).

If you want to manipulate the memory of the value field inside the Octets struct in C/Traditional C++, use the operations **DDS::OctetBuffer_alloc()**, **DDS::OctetBuffer_dup()**, and **DDS::OctetBuffer_free()**, described in the API Reference HTML documentation (select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **Octet Buffer Support**).

3.2.5.2 Octets DataWriter

(Note: for Modern C++ API, refer to the API documentation)

In addition to the standard methods (see [Using a Type-Specific DataWriter \(FooDataWriter\)](#) (Section 6.3.7 on page 281)), the octets *DataWriter* API is extended with the following methods:

```
DDS::ReturnCode_t DDS::OctetsDataWriter::write(
    const DDS::OctetSeq & octets,
    const DDS::InstanceHandle_t & handle);
DDS::ReturnCode_t DDS::OctetsDataWriter::write(
    const unsigned char * octets,
    int length,
    const DDS::InstanceHandle_t& handle);
DDS::ReturnCode_t DDS::OctetsDataWriter::write_w_timestamp(
    const DDS::OctetSeq & octets,
    const DDS::InstanceHandle_t & handle,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t DDS::OctetsDataWriter::write_w_timestamp(
    const unsigned char * octets,
    int length,
    const DDS::InstanceHandle_t& handle,
    const DDS::Time_t& source_timestamp);
```

These methods are introduced to provide maximum flexibility in the format of the input parameters for the write operations. For additional information and a complete description of these operations in all supported languages, see the API Reference HTML documentation.

The following examples show how to write an array of octets using an octets built-in type *DataWriter* and some of the extended APIs. For simplicity, error handling is not shown.

C Example:

```
DDS_OctetsDataWriter * octetsWriter = ... ;
DDS_ReturnCode_t retCode;
struct DDS_Octets * octets = NULL;
char * octetArray = NULL;
/* Write some data using the Octets structure */
octets = DDS_Octets_new_w_size(1024);
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
retCode = DDS_OctetsDataWriter_write(
    octetsWriter, octets, &DDS_HANDLE_NIL);
DDS_Octets_delete(octets);
/* Write some data using an octets array */
octetArray = (unsigned char *)malloc(1024);
octetArray[0] = 46;
octetArray[1] = 47;
```

```
retCode = DDS_OctetsDataWriter_write_octets (
    octetsWriter, octetArray, 2,
    &DDS_HANDLE_NIL);
free(octetArray);
```

C++ Example with Namespaces:¹

```
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
OctetsDataWriter * octetsWriter = ... ;
/* Write some data using the Octets structure */
Octets * octets = new Octets(1024);
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
ReturnCode_t retCode = octetsWriter->write(octets, HANDLE_NIL);
delete octets;
/* Write some data using an octet array */
unsigned char * octetArray = new unsigned char[1024];
octetArray[0] = 46;
octetArray[1] = 47;
retCode = octetsWriter->write(octetArray, 2, HANDLE_NIL);
delete []octetArray;
```

C++/CLI Example:

```
using namespace System;
using namespace DDS;
...
BytesDataWriter^ octetsWriter = ...;
/* Write some data using Bytes */
Bytes^ octets = gcnew Bytes(1024);
octets->value[0] =46;
octets->value[1] =47;
octets.length = 2;
octets.offset = 0;
octetWriter->write(octets, InstanceHandle_t::HANDLE_NIL);
/* Write some data using individual strings */
array<Byte>^ octetArray = gcnew array<Byte>(1024);
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter->write(octetArray, 0, 2, InstanceHandle_t::HANDLE_NIL);
```

C# Example:

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

```

using System;
using DDS;
...
BytesDataWriter stringWriter = ...;
/* Write some data using the Bytes */
Bytes octets = new Bytes(1024);
octets.value[0] = 46;
octets.value[1] = 47;
octets.length = 2;
octets.offset = 0;
octetWriter.write(octets, InstanceHandle_t.HANDLE_NIL);
/* Write some data using individual strings */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter.write(octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);

```

Java Example:

```

import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
BytesDataWriter octetsWriter = ... ;
/* Write some data using the Bytes class*/
Bytes octets = new Bytes(1024);
octets.length = 2;
octets.offset = 0;
octets.value[0] = 46;
octets.value[1] = 47;
octetsWriter.write(octets, InstanceHandle_t.HANDLE_NIL);
/* Write some data using a byte array */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter.write(octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);

```

3.2.5.3 Octets DataReader

(Note: for the Modern C++ API, refer to the API Reference HTML documentation)

The octets *DataReader* API matches the standard *DataReader* API (see [Using a Type-Specific DataReader \(FooDataReader\) \(Section 7.4.1 on page 491\)](#)). There are no extensions.

Memory considerations in copy operations:

For read/take operations with copy semantics, such as `read_next_sample()` and `take_next_sample()`, Connex DDS allocates memory for the field 'value' if it is initialized to NULL.

If the field 'value' is not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory for the field 'value' will be reallocated if the current size is not large enough to hold the received data.
- In C and C++, the memory associated with the field 'value' must be big enough to hold the received data. Insufficient memory may result in crashes.

The following examples show how to read octets with an octets built-in type *DataReader*. For simplicity, error handling is not shown.

C Example:

```
struct DDS_OctetsSeq dataSeq = DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq = DDS_SEQUENCE_INITIALIZER;
DDS_OctetsDataReader * octetsReader = ... ;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_OctetsDataReader_take(
    octetsReader, &dataSeq,
    &infoSeq, DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
for (i = 0; i < DDS_OctetsSeq_get_length(&dataSeq); ++i) {
    if (DDS_SampleInfoSeq_get_reference(
        &infoSeq, i)->valid_data) {
        DDS_OctetsTypeSupport_print_data(
            DDS_OctetsSeq_get_reference(&dataSeq, i));
    }
}
/* Return loan */
retCode = DDS_OctetsDataReader_return_loan(
    octetsReader, &dataSeq, &infoSeq);
```

C++ Example with Namespaces:¹

```
#include "ndds/ndds_namespace_cpp.h"
```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, `DDS::StringDataWriter` becomes `DDSStringDataWriter`.


```

using namespace DDS;
...
OctetsSeq dataSeq;
SampleInfoSeq infoSeq;
OctetsDataReader * octetsReader = ... ;
/* Take a print the data */
ReturnCode_t retCode = octetsReader->take(
    dataSeq, infoSeq,
    LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
    ANY_VIEW_STATE, ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        OctetsTypeSupport::print_data(&dataSeq[i]);
    }
}
/* Return loan */
retCode = octetsReader->return_loan(dataSeq, infoSeq);

```

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
BytesSeq^ dataSeq = gcnew BytesSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
BytesDataReader^ octetsReader = ... ;
/* Take and print the data */
octetsReader->take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq->get_at(i)->valid_data) {
        BytesTypeSupport::print_data(dataSeq->get_at(i));
    }
}
/* Return loan */
octetsReader->return_loan(dataSeq, infoSeq);

```

C# Example:

```

using System;
using DDS;
...
BytesSeq dataSeq = new BytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
BytesDataReader octetsReader = ... ;

```

```

/* Take and print the data */
octetsReader.take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq.get_at(i)).valid_data) {
        ByteTypeSupport.print_data(dataSeq.get_at(i));
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);

```

Java Example:

```

import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
BytesSeq dataSeq = new BytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
BytesDataReader octetsReader = ... ;
/* Take and print the data */
octetsReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println(((Bytes)dataSeq.get(i)).toString());
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);

```

3.2.6 KeyedOctets Built-in Type

The keyed octets built-in type is used to send sequences of octets with a key. The language-specific representations of the type are as follows:

C/Traditional C++ Representation (without Namespaces):

```

struct DDS_KeyedOctets {
    char * key;
    int length;
    unsigned char * value;
}

```

```
};
```

Modern C++ Representation:

```
class dds::core::KeyedStringTopicType {
public:
    dds::core::string& key();
    uint8_t& operator [] (uint32_t index);
    // ... see API documentation for full definition
};
```

C++/CLI Representation:

```
namespace DDS {
    public ref struct KeyedBytes {
    public:
        System::String^ key;
        System::Int32 length;
        System::Int32 offset;
        array<System::Byte>^ value;
        ...
    };
};
```

C# Representation:

```
namespace DDS {
    public class KeyedBytes {
    public System.String key;
    public System.Int32 length;
    public System.Int32 offset;
    public System.Byte[] value;
    ...
    };
};
```

Java Representation:

```
package com.rti.dds.type.builtin;
public class KeyedBytes {
    public String key;
    public int length;
    public int offset;
    public byte[] value;
    ...
};
```

3.2.6.1 Creating and Deleting KeyedOctets

Connex DDS provides a set of constructors/destructors to create/destroy KeyedOctets objects. For details, see the API Reference HTML documentation, which is available for all supported programming languages (select **Modules**, **RTI Connex DDS API Reference**, **Topic Module**, **Built-in Types**).

To manipulate the memory of the **value** field in the KeyedOctets struct in C/C++: use **DDS::OctetBuffer_alloc()**, **DDS::OctetBuffer_dup()**, and **DDS::OctetBuffer_free()**. See the API Reference HTML documentation (select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **Octet Buffer Support**).

To manipulate the memory of the **key** field in the KeyedOctets struct in C/C++: use **DDS::String_alloc()**, **DDS::String_dup()**, and **DDS::String_free()**. See the API Reference HTML documentation (select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **String Support**).

3.2.6.2 Keyed Octets DataWriter

In addition to the standard methods (see [Using a Type-Specific DataWriter \(FooDataWriter\) \(Section 6.3.7 on page 281\)](#)), the keyed octets *DataWriter* API is extended with the following methods:

```
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::dispose(
    const char* key,
    const DDS::InstanceHandle_t & instance_handle);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::dispose_w_timestamp(
    const char* key,
    const DDS::InstanceHandle_t & instance_handle,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::get_key_value(
    char * key,
    const DDS::InstanceHandle_t& handle);
DDS::InstanceHandle_t
DDS::KeyedOctetsDataWriter::lookup_instance(
    const char * key);
DDS::InstanceHandle_t
DDS::KeyedOctetsDataWriter::register_instance(
    const char* key);
DDS::InstanceHandle_t
DDS::KeyedOctetsDataWriter::
    register_instance_w_timestamp(
    const char * key,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::unregister_instance(
    const char * key,
    const DDS::InstanceHandle_t & handle);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::
```

```

unregister_instance_w_timestamp(
    const char* key,
    const DDS::InstanceHandle_t & handle,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::write(
    const char * key,
    const unsigned char * octets,
    int length,
    const DDS::InstanceHandle_t& handle);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::write(
    const char * key,
    const DDS::OctetSeq & octets,
    const DDS::InstanceHandle_t & handle);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::write_w_timestamp(
    const char * key,
    const unsigned char * octets,
    int length,
    const DDS::InstanceHandle_t& handle,
    const DDS::Time_t& source_timestamp);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::write_w_timestamp(
    const char * key,
    const DDS::OctetSeq & octets,
    const DDS::InstanceHandle_t & handle,
    const DDS::Time_t & source_timestamp);

```

These methods are introduced to provide maximum flexibility in the format of the input parameters for the write and instance management operations. For more information and a complete description of these operations in all supported languages, see the API Reference HTML documentation.

The following examples show how to write keyed octets using a keyed octets built-in type *DataWriter* and some of the extended APIs. For simplicity, error handling is not shown.

C Example:

```

DDS_KeyedOctetsDataWriter * octetsWriter = ... ;
DDS_ReturnCode_t retCode;
struct DDS_KeyedOctets * octets = NULL;
char * octetArray = NULL;
/* Write some data using KeyedOctets structure */
octets = DDS_KeyedOctets_new_w_size(128,1024);
strcpy(octets->key, "Key 1");
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
retCode = DDS_KeyedOctetsDataWriter_write(
    octetsWriter, octets, &DDS_HANDLE_NIL);

```

```

DDS_KeyedOctets_delete(octets);
/* Write some data using an octets array */
octetArray = (unsigned char *)malloc(1024);
octetArray[0] = 46;
octetArray[1] = 47;
retCode =
DDS_KeyedOctetsDataWriter_write_octets_w_key (
    octetsWriter, "Key 1",
    octetArray, 2, &DDS_HANDLE_NIL);
free(octetArray);

```

C++ Example with Namespaces:¹

```

#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedOctetsDataWriter * octetsWriter = ...;
/* Write some data using KeyedOctets */
KeyedOctets * octets = new KeyedOctets(128,1024);
strcpy(octets->key, "Key 1");
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
ReturnCode_t retCode =
    octetsWriter->write(octets, HANDLE_NIL);
delete octets;
/* Write some data using an octet array */
unsigned char * octetArray = new unsigned char[1024];
octetArray[0] = 46;
octetArray[1] = 47;
retCode = octetsWriter->write(
    "Key 1", octetArray, 2, HANDLE_NIL);
delete []octetArray;

```

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
KeyedOctetsDataWriter^ octetsWriter = ... ;
/* Write some data using KeyedBytes */
KeyedBytes^ octets = gcnew KeyedBytes(1024);
octets->key = "Key 1";
octets->value[0] =46;

```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

```

octets->value[1] =47;
octets.length = 2;
octets.offset = 0;
octetWriter->write(
    octets, InstanceHandle_t::HANDLE_NIL);
/* Write some data using individual strings */
array<Byte>^ octetArray = gcnew array<Byte>(1024);
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter->write(
    "Key 1", octetArray,
    0, 2, InstanceHandle_t::HANDLE_NIL);

```

C# Example:

```

using System;
using DDS;
...
KeyedBytesDataWriter stringWriter = ... ;
/* Write some data using the KeyedBytes */
KeyedBytes octets = new KeyedBytes(1024);
octets.key = "Key 1";
octets.value[0] = 46;
octets.value[1] = 47;
octets.length = 2;
octets.offset = 0;
octetWriter.write(octets,
    InstanceHandle_t.HANDLE_NIL);
/* Write some data using individual strings */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter.write(
    "Key 1", octetArray,
    0, 2, InstanceHandle_t.HANDLE_NIL);

```

Java Example:

```

import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
KeyedBytesDataWriter octetsWriter = ... ;
/* Write some data using KeyedBytes class */
KeyedBytes octets = new KeyedBytes(1024);
octets.key = "Key 1";
octets.length = 2;
octets.offset = 0;
octets.value[0] = 46;

```

```

octets.value[1] = 47;
octetsWriter.write(octets,
    InstanceHandle_t.HANDLE_NIL);
/* Write some data using a byte array */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter.write(
    "Key 1", octetArray,
0, 2, InstanceHandle_t.HANDLE_NIL);

```

3.2.6.3 Keyed Octets DataReader

The KeyedOctets *DataReader* API is extended with the following methods (in addition to the standard methods described in [Using a Type-Specific DataReader \(FooDataReader\) \(Section 7.4.1 on page 491\)](#)):

```

DDS::ReturnCode_t
DDS::KeyedOctetsDataReader::get_key_value(
    char * key,
    const DDS::InstanceHandle_t* handle);

DDS::InstanceHandle_t
DDS::KeyedOctetsDataReader::lookup_instance(
    const char * key);

```

For more information and a complete description of these operations in all supported languages, see the [API Reference HTML documentation](#).

Memory considerations in copy operations:

For read/take operations with copy semantics, such as **read_next_sample()** and **take_next_sample()**, Connex DDS allocates memory for the fields '**value**' and '**key**' if they are initialized to NULL.

If the fields are not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory of the field '**value**' will be reallocated if the current size is not large enough to hold the received data. The memory associated with the field '**key**' will be reallocated with every DDS sample (the key is an immutable object).
- In C and C++, the memory associated with the fields '**value**' and '**key**' must be large enough to hold the received data. Insufficient memory may result in crashes.

The following examples show how to read keyed octets with a keyed octets built-in type *DataReader*. For simplicity, error handling is not shown.

C Example:


```

struct DDS_KeyedOctetsSeq dataSeq =
    DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq =
    DDS_SEQUENCE_INITIALIZER;
DDS_KeyedOctetsDataReader * octetsReader = ... ;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_KeyedOctetsDataReader_take(
    octetsReader,
    &dataSeq, &infoSeq, DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
for (i = 0;
    i < DDS_KeyedOctetsSeq_get_length(&data_seq);
    ++i) {
    if (DDS_SampleInfoSeq_get_reference(
        &info_seq, i)->valid_data) {
        DDS_KeyedOctetsTypeSupport_print_data(
            DDS_KeyedOctetsSeq_get_reference(
                &data_seq, i));
    }
}
/* Return loan */
retCode = DDS_KeyedOctetsDataReader_return_loan(
    octetsReader, &data_seq, &info_seq);

```

C++ Example with Namespaces:¹

```

#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedOctetsSeq dataSeq;
SampleInfoSeq infoSeq;
KeyedOctetsDataReader * octetsReader = ... ;
/* Take and print the data */
ReturnCode_t retCode = octetsReader->take(
    dataSeq, infoSeq, LENGTH_UNLIMITED,
    ANY_SAMPLE_STATE, ANY_VIEW_STATE,
    ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        KeyedOctetsTypeSupport::print_data(
            &dataSeq[i]);
    }
}

```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

```

}
/* Return loan */
retCode = octetsReader->return_loan(
    dataSeq, infoSeq);

```

C++/CLI Example:

```

using namespace System;
using namespace DDS;
...
KeyedBytesSeq^ dataSeq = gcnew KeyedBytesSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
KeyedBytesDataReader^ octetsReader = ... ;
/* Take and print the data */
octetsReader->take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i){
    if (infoSeq->get_at(i)->valid_data){
        KeyedBytesTypeSupport::print_data(
            dataSeq->get_at(i));
    }
}
/* Return loan */
octetsReader->return_loan(dataSeq, infoSeq);

```

C# Example:

```

using System;
using DDS;
...
KeyedBytesSeq dataSeq = new KeyedButesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedBytesDataReader octetsReader = ... ;
/* Take and print the data */
octetsReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq.get_at(i).valid_data) {
        KeyedBytesTypeSupport.print_data(
            dataSeq.get_at(i));
    }
}
/* Return loan */

```

```
octetsReader.return_loan(dataSeq, infoSeq);
```

Java Example:

```
import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
KeyedBytesSeq dataSeq = new KeyedBytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedBytesDataReader octetsReader = ... ;
/* Take and print the data */
octetsReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);

for (int i = 0; i < data_seq.length(); ++i) {
    if (((SampleInfo) infoSeq.get(i)).valid_data) {
        System.out.println(
            ((KeyedBytes) dataSeq.get(i)).toString());
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);
```

3.2.7 Managing Memory for Built-in Types

When a DDS sample is written, the *DataWriter* serializes it and stores the result in a buffer obtained from a pool of preallocated buffers. In the same way, when a DDS sample is received, the *DataReader* deserializes it and stores the result in a DDS sample coming from a pool of preallocated DDS samples.

By default, the buffers on the *DataWriter* and the samples on the *DataReader* are preallocated with their maximum size. For example:

```
struct MyString
Unknown macro: { string<128> value; }
```

This IDL-defined type has a maximum serialized size of 133 bytes (4 bytes for length + 128 characters + 1 NULL terminating character). So the serialization buffers will have a size of 133 bytes. The buffer can hold samples with 128 characters strings. Consequently, the preallocated samples will be sized to keep this length.

However, for built-in types, the maximum size of the buffers/DDS samples is unknown and depends on the nature of the application using the built-in type.

For example, a video surveillance application that is using the keyed octets built-in type to publish a stream of images will require bigger buffers than a market-data application that uses the same built-in type to publish market-data values.

To accommodate both kinds of applications and optimize memory usage, you can configure the maximum size of the built-in types on a per-*DataWriter* or per-*DataReader* basis using the [PROPERTY QoS Policy \(DDS Extension\)](#) (Section 6.5.17 on page 394). [Table 3.1 Properties for Allocating Size of Built-in Types, per DataWriter and DataReader](#) lists the supported built-in type properties. When the properties are defined in the *DomainParticipant*, they are applicable to all *DataWriters* and *DataReaders* belonging to the *DomainParticipant*, unless they are overwritten in the *DataWriters* and *DataReaders*.

These properties must be set consistently with respect to the corresponding ***.max_size** properties in the *DomainParticipant* (see [Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant](#)). The value of the **alloc_size** property must be less than or equal to the **max_size** property with the same name prefix in the *DomainParticipant*.

[Examples—Setting the Maximum Size for a String Programmatically](#) (Section 3.2.7.1 on the next page) includes examples of how to set the maximum size of a string built-in type for a *DataWriter* programmatically, for each API. You can also set the maximum size of the built-in types using XML QoS Profiles. For example, the following XML shows how to set the maximum size of a string built-in type for a *DataWriter*.

```
<dds>
<qos_library name="BuiltinExampleLibrary">
  <qos_profile name="BuiltinExampleProfile">
    <datawriter_qos>
      <property>
        <value>
          <element>
            <name>dds.builtin_type.string.alloc_size</name>
            <value>2048</value>
          </element>
        </value>
      </property>
    </datawriter_qos>
    <datareader_qos>
      <property>
        <value>
          <element>
            <name>dds.builtin_type.string.alloc_size</name>
            <value>2048</value>
          </element>
        </value>
      </property>
    </datareader_qos>
  </qos_profile>
</qos_library>
```

```
</qos_library>
</dds>
```

Table 3.1 Properties for Allocating Size of Built-in Types, per DataWriter and DataReader

Built-in Type	Property	Description
string	dds.builtin_type.string.alloc_size	Maximum size of the strings published by the DataWriter or received by the DataReader (includes the NULL-terminated character). Default: dds.builtin_type.string.max_size if defined (see Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.
keyedstring	dds.builtin_type.keyed_string.alloc_key_size	Maximum size of the keys used by the DataWriter or DataReader (includes the NULL-terminated character). Default: dds.builtin_type.keyed_string.max_key_size if defined (see Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.
	dds.builtin_type.keyed_string.alloc_size	Maximum size of the strings published by the DataWriter or received by the DataReader (includes the NULL-terminated character). Default: dds.builtin_type.keyed_string.max_size if defined (see Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.
octets	dds.builtin_type.octets.alloc_size	Maximum size of the octet sequences published by the DataWriter or DataReader. Default: dds.builtin_type.octets.max_size if defined (see Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 2048.
keyed-octets	dds.builtin_type.keyed_octets.alloc_key_size	Maximum size of the key published by the DataWriter or received by the DataReader (includes the NULL-terminated character). Default: dds.builtin_type.keyed_octets.max_key_size if defined (see Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.
	dds.builtin_type.keyed_octets.alloc_size	Maximum size of the octet sequences published by the DataWriter or DataReader. Default: dds.builtin_type.keyed_octets.max_size if defined (see Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 2048.

3.2.7.1 Examples—Setting the Maximum Size for a String Programmatically

For simplicity, error handling is not shown in the following examples.

C Example:

```
DDS_DataWriter * writer = NULL;
DDS_StringDataWriter * stringWriter = NULL;
DDS_Publisher * publisher = ... ;
DDS_Topic * stringTopic = ... ;
struct DDS_DataWriterQos writerQos =
```

```

    DDS_DataWriterQos_INITIALIZER;
DDS_ReturnCode_t retCode;
retCode = DDS_DomainParticipant_get_default_datawriter_qos (
    participant, &writerQos);
retCode = DDS_PropertyQosPolicyHelper_add_property (
    &writerQos.property,
    "dds.builtin_type.string.alloc_size", "1000",
    DDS_BOOLEAN_FALSE);
writer = DDS_Publisher_create_datawriter(
    publisher, stringTopic, &writerQos,
    NULL, DDS_STATUS_MASK_NONE);
stringWriter = DDS_StringDataWriter_narrow(writer);
DDS_DataWriterQos_finalize(&writerQos);

```

Traditional C++ Example with Namespaces: ¹

```

#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
Publisher * publisher = ... ;
Topic * stringTopic = ... ;
DataWriterQos writerQos;
ReturnCode_t retCode =
    participant->get_default_datawriter_qos(writerQos);
retCode = PropertyQosPolicyHelper::add_property (
    &writerQos.property,
    "dds.builtin_type.string.alloc_size",
    "1000", BOOLEAN_FALSE);
DataWriter * writer = publisher->create_datawriter(
    stringTopic, writerQos,
    NULL, STATUS_MASK_NONE);
StringDataWriter * stringWriter =
    StringDataWriter::narrow(writer);

```

Modern C++ Example:

```

dds::pub::qos::DataWriterQos writer_qos =
    participant.default_datawriter_qos();
writer_qos.policy<rti::core::policy::Property>().set({
    "dds.builtin_type.string.alloc_size", "1000"});
dds::pub::DataWriter<dds::core::StringTopicType> writer(
    publisher, string_topic, writer_qos);

```

C++/CLI Example:

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.

```

using namespace DDS;
...
Topic^ stringTopic = ... ;
Publisher^ publisher = ... ;
DataWriterQos^ writerQos = gcnew DataWriterQos();
participant->get_default_datawriter_qos(writerQos);
PropertyQosPolicyHelper::add_property(
    writerQos->property_qos,
    "dds.builtin_type.string.alloc_size",
    "1000", false);
DataWriter^ writer = publisher->create_datawriter(
    stringTopic, writerQos,
    nullptr, StatusMask::STATUS_MASK_NONE);
StringDataWriter^ stringWriter =
    safe_cast<StringDataWriter^>(writer);

```

C# Example:

```

using DDS;
...
Topic stringTopic = ... ;
Publisher publisher = ... ;
DataWriterQos writerQos = new DataWriterQos();
participant.get_default_datawriter_qos(writerQos);
PropertyQosPolicyHelper.add_property (
    writerQos.property_qos,
    "dds.builtin_type.string.alloc_size",
    "1000", false);
StringDataWriter stringWriter =
    (StringDataWriter) publisher.create_datawriter(
        stringTopic, writerQos, null,
        StatusMask.STATUS_MASK_NONE);

```

Java Example:

```

import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
Topic stringTopic = ... ;
Publisher publisher = ... ;
DataWriterQos writerQos = new DataWriterQos();
participant.get_default_datawriter_qos(writerQos);
PropertyQosPolicyHelper.add_property (
    writerQos.property,
    "dds.builtin_type.string.alloc_size",
    "1000", false);
StringDataWriter stringWriter =
    (StringDataWriter) publisher.create_datawriter(

```

```
stringTopic, writerQos,
null, StatusKind.STATUS_MASK_NONE);
```

3.2.7.2 Unbounded Built-in Types

In some scenarios, the maximum size of a built-in type is not known in advance and there is no a reasonable maximum size. For example, this could occur in a file transfer application using the built-in type Octets. Setting a large value for the **dds.builtin_type.*.alloc_size** property would involve high memory usage.

For the above use case, you can configure the built-in type to be unbounded by setting the property **dds.builtin_type.*.alloc_size** to the maximum value of a 32-bit signed integer: 2,147,483,647. Then the middleware will not preallocate the *DataReader* queue's samples to their maximum size. Instead, it will deserialize incoming samples by dynamically allocating and deallocating memory to accommodate the actual size of the sample value.

To configure unbounded support for built-in types:

1. Set the properties **dds.builtin_type.*.max_size** and **dds.builtin_type.*.alloc_size** to 2,147,483,647.
2. Use these threshold QoS properties:
 - **dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size** on the *DataWriter*
 - **dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size** on the *DataReader* (only if keyed)
3. Set the QoS value **reader_resource_limits.dynamically_allocate_fragmented_samples** on the *DataReader* to true.
4. For the Java API, also set these properties accordingly for the Java serialization buffer:
 - **dds.data_writer.history.memory_manager.java_stream.min_size**
 - **dds.data_writer.history.memory_manager.java_stream.trim_to_size**
 - **dds.data_reader.history.memory_manager.java_stream.min_size**
 - **dds.data_reader.history.memory_manager.java_stream.trim_to_size**

See these sections in the *RTI Connext DDS Core Libraries User's Manual*:

- Section 20.1.3, Writer-Side Memory Management when Using Java
- Section 20.2.2, Reader-Side Memory Management when Using Java

Unbounded built-in types are only supported in the C, C++, .NET, and Java APIs.

3.2.8 Type Codes for Built-in Types

The type codes associated with the built-in types are generated from the following IDL type definitions:

```

module DDS {
    /* String */
    struct String {
        string<max_size> value;
    };
    /* KeyedString */
    struct KeyedString {
        string<max_size> key; //@key
        string<max_size> value;
    };
    /* Octets */
    struct Octets {
        sequence<octet, max_size> value;
    };
    /* KeyedOctets */
    struct KeyedOctets {
        string<max_size> key; //@key
        sequence<octet, max_size> value;
    };
};

```

The maximum size (**max_size**) of the strings and sequences that will be included in the type code definitions can be configured on a per-*DomainParticipant*-basis by using the properties in [Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant](#).

Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant

Built-in Type	Property	Description
String	dds.builtin_type.string.max_size	Maximum size of the strings published by the <i>DataWriters</i> and received by the <i>DataReaders</i> belonging to a <i>DomainParticipant</i> (includes the NULL-terminated character). Default: 1024
KeyedString	dds.builtin_type.keyed_string.max_key_size	Maximum size of the keys used by the <i>DataWriters</i> and <i>DataReaders</i> belonging to a <i>DomainParticipant</i> (includes the NULL-terminated character). Default: 1024
	dds.builtin_type.keyed_string.max_size	Maximum size of the strings published by the <i>DataWriters</i> and received by the <i>DataReaders</i> belonging to a <i>DomainParticipant</i> using the built-in type (includes the NULL-terminated character). Default: 1024

Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant

Built-in Type	Property	Description
Octets	dds.builtin_type.octets.max_size	Maximum size of the octet sequences published by the <i>DataWriters</i> and <i>DataReaders</i> belonging to a <i>DomainParticipant</i> . Default: 2048
Keyed-Octets	dds.builtin_type.keyed_octets.max_key_size	Maximum size of the key published by the <i>DataWriter</i> and received by the <i>DataReaders</i> belonging to the <i>DomainParticipant</i> (includes the NULL-terminated character). Default: 1024.
	dds.builtin_type.keyed_octets.max_size	Maximum size of the octet sequences published by the <i>DataWriters</i> and <i>DataReaders</i> belonging to a <i>DomainParticipant</i> . Default: 2048

3.3 Creating User Data Types with IDL

You can create user data types in a text file using IDL (Interface Description Language). IDL is programming-language independent, so the same file can be used to generate code in C, Traditional C++, Modern C++, C++/CLI, and Java (the languages supported by *RTI Code Generator* (*rtiddsgen*)). *RTI Code Generator* parses the IDL file and automatically generates all the necessary routines and wrapper functions to bind the types for use by Connex DDS at run time. You will end up with a set of required routines and structures that your application and Connex DDS will use to manipulate the data.

Connex DDS only uses a subset of the IDL syntax. IDL was originally defined by the OMG for the use of CORBA client/server applications in an enterprise setting. Not all of the constructs that can be described by the language are as useful in the context of high-performance data-centric embedded applications. These include the constructs that define method and function prototypes like “interface.”

RTI Code Generator will parse any file that follows version 3.0.3 of the IDL specification. It will quietly ignore all syntax that is not recognized by Connex DDS. In addition, even though “anonymous sequences” (sequences of sequences with no intervening typedef) are currently legal in IDL, they have been deprecated by the specification; thus *RTI Code Generator* does not support them.

Certain keywords are considered reserved by the IDL specification; see [Table 3.3 Reserved IDL Keywords](#).

Table 3.3 Reserved IDL Keywords

abstract	emits	local	pseudo	typeid
alias	enum	long	public	typename

Table 3.3 Reserved IDL Keywords

any	eventtype	mirrorport	publishes	typeprefix
attribute	exception	module	raises	union
boolean	factory	multiple	readonly	unsigned
case	FALSE	native	sequence	uses
char	finder	object	setraises	valuebase
component	fixed	octet	short	valuetype
connector	float	oneway	string	void
const	getraises	out	struct	wchar
consumes	home	port	supports	wstring
context	import	porttype	switch	
custom	in	primarykey	TRUE	
default	inout	private	truncatable	
double	interface	provides	typedef	

The IDL constructs supported by *RTI Code Generator* are described in [Table 3.5 Specifying Data Types in IDL for C](#) through [Table 3.9 Specifying Data Types in IDL for Java](#). Use these tables to map primitive types to their equivalent IDL syntax, and vice versa.

For C and Traditional C++, *RTI Code Generator* uses typedefs instead of the language keywords for primitive types. For example, `DDS_Long` instead of `long` or `DDS_Double` instead of `double`. This ensures that the types are of the same size regardless of the platform.¹

The remainder of this section includes:

3.3.1 Variable-Length Types

When *RTI Code Generator* generates code for data structures with variable-length types—strings and sequences—it includes functions that create, initialize and finalize (destroy) those objects. These support

¹The number of bytes sent on the wire for each data type is determined by the Common Data Representation (CDR) standard. For details on CDR, please see the Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 2: CORBA Interoperability, Section 9.3, CDR Transfer Syntax (<http://www.omg.org/spec/CORBA/3.3/>).

functions will properly initialize pointers and allocate and deallocate the memory used for variable-length types. All Connex DDS APIs assume that the data structures passed to them are properly initialized.

For variable-length types, the actual length (instead of the maximum length) of data is transmitted on the wire when the DDS sample is written (regardless of whether the type has hard-coded bounds).

3.3.1.1 Sequences

C, Traditional C++, C++/CLI, and C# users can allocate memory from a number of sources: from the heap, the stack, or from a custom allocator of some kind. In those languages, sequences provide the concept of memory "ownership." A sequence may own the memory allocated to it or be loaned memory from another source. If a sequence owns its memory, it will manage its underlying memory storage buffer itself. When a sequence's maximum size is changed, the sequence will free and reallocate its buffer as needed. However, if a sequence was created with loaned memory by user code, then its memory is not its own to free or reallocate. Therefore, you cannot set the maximum size of a sequence whose memory is loaned. See the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI Connex DDS API Reference, Infrastructure Module, Sequence Support) for more information about how to loan and unloan memory for sequence.

In IDL, as described above, a sequence may be declared as bounded or unbounded. A sequence's "bound" is the greatest value its maximum may take. If you use the initializer functions *RTI Code Generator* provides for your types, all sequences will have their maximums set to their declared bounds. However, the amount of data transmitted on the wire when the DDS sample is written will vary.

In the Modern C++ API, sequences (`dds::core::vector`) always own the memory.

3.3.1.2 Strings and Wide Strings

(Note: this section doesn't apply to the Modern C++ API, where `dds::core::string` behaves similarly to `std::string`)

The initialization functions that *RTI Code Generator* provides for your types will allocate all of the memory for strings in a type to their declared bounds. Take care—if you assign a string pointer (`char *`) in a data structure allocated or initialized by a Connex DDS-generated function, you should release (free) the memory originally allocated for the string, otherwise the memory will be leaked.

To Java and .NET users, an IDL string is a `String` object: it is immutable and knows its own length. C and C++ users must take care, however, as there is no way to determine how much memory is allocated to a character pointer "string"; all that can be determined is the string's current logical length. In some cases, Connex DDS may need to copy a string into a structure that user code has provided. Connex DDS does not free the memory of the string provided to it, as it cannot know from where that memory was allocated.

In the C and C++ APIs, Connex DDS therefore uses the following conventions:

- A string's memory is "owned" by the structure that contains that string. Calling the finalization function provided for a type will free all recursively contained strings. If you have allocated a contained string in a special way, you must be careful to clean up your own memory and assign the pointer to NULL *before* calling the type's **finalize()** method, so that Connex DDS will skip over that string.
- You must provide a non-NULL string pointer for Connex DDS to copy into. Otherwise, Connex DDS will log an error.
- When you provide a non-NULL string pointer in your data structure, Connex DDS will copy into the provided memory without performing any additional memory allocations. Be careful—if you provide Connex DDS with an uninitialized pointer or allocate a string that is too short, you may corrupt the memory or cause a program crash. Connex DDS will never try to copy a string that is longer than the bound of the destination string. However, your application must insure that any string that it allocates is long enough.

Connex DDS provides a small set of C functions for dealing with strings. These functions simplify common tasks, avoid some platform-specific issues (such as the lack of a **strdup()** function on some platforms), and provide facilities for dealing with wide strings, for which no standard C library exists. Connex DDS always uses these functions internally for managing string memory; you are recommended—but not required—to use them as well. See the API Reference HTML documentation, which is available for all supported programming languages (select **Modules**, **RTI DDS API Reference**, **Infrastructure Module**, **String Support**) for more information about strings.

3.3.2 Value Types

A value type is like a structure, but with support for additional object-oriented features such as inheritance. It is similar to what is sometimes referred to in Java as a *POJO*—a Plain Old Java Object.

Readers familiar with value types in the context of CORBA should consult [Table 3.4 Value Type Support](#) to see which value type-related IDL keywords are supported and what their behavior is in the context of Connex DDS.

Table 3.4 Value Type Support

Aspect	Level of Support in RTI Code Generator
Inheritance	Single inheritance from other value types
Public state members	Supported
Private state members	Become public when code is generated
Custom keyword	Ignored (the value type is parsed without the keyword and code is generated to work with it)
Abstract value types	No code generated (the value type is parsed, but no code is generated)

Table 3.4 Value Type Support

Aspect	Level of Support in RTI Code Generator
Operations	No code generated (the value type is parsed, but no code is generated)
Truncatable keyword	Ignored (the value type is parsed without the keyword and code is generated to work with it)

3.3.3 Type Codes

Type codes are enabled by default when you run *RTI Code Generator*. The **-notypecode** option disables generation of type code information. Type-code support does increase the amount of memory used, so if you need to save on memory, you may consider disabling type codes. (The **-notypecode** option is described in the *RTI Code Generator User's Manual*.)

Locally, your application can access the type code for a generated type "Foo" by calling the **FooTypeSupport::get_typecode()** (Traditional C++ Notation) operation in the code for the type generated by *RTI Code Generator* (unless type-code support is disabled with the **-notypecode** option).

Note: Type-code support must be enabled if you are going to use [ContentFilteredTopics \(Section 5.4 on page 212\)](#) with the default SQL filter. You may disable type codes and use a custom filter, as described in [Creating ContentFilteredTopics \(Section 5.4.3 on page 214\)](#).

3.3.4 Translations for IDL Types

This section describes how to specify your data types in an IDL file. *RTI Code Generator* supports all the types listed in the following tables:

- [Table 3.5 Specifying Data Types in IDL for C](#)
- [Table 3.6 Specifying Data Types in IDL for Traditional C++](#)
- [Table 3.8 Specifying Data Types in IDL for the Modern C++ API](#)
- [Table 3.7 Specifying Data Types in IDL for C++/CLI](#)
- [Table 3.9 Specifying Data Types in IDL for Java](#)
- [Table 3.10 Specifying Data Types in IDL for Ada](#)

In each table, the middle column shows the IDL syntax for a data type in an IDL file. The rightmost column shows the corresponding language mapping created by *RTI Code Generator*.

Table 3.5 Specifying Data Types in IDL for C

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
char (see Note: 1 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Char char_member; } PrimitiveStruct;</pre>
wchar	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Wchar wchar_member; } PrimitiveStruct;</pre>
octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Octet octet_member; } PrimitiveStruct;</pre>
short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Short short_member; } PrimitiveStruct;</pre>
unsigned short	<pre>struct PrimitiveStruct { unsigned short unsigned_short_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_UnsignedShort unsigned_short_member; } PrimitiveStruct;</pre>
long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Long long_member; } PrimitiveStruct;</pre>
unsigned long	<pre>struct PrimitiveStruct { unsigned long unsigned_long_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_UnsignedLong unsigned_long_member; } PrimitiveStruct;</pre>
long long	<pre>struct PrimitiveStruct { long long long_long_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_LongLong long_long_member; } PrimitiveStruct;</pre>

Table 3.5 Specifying Data Types in IDL for C

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unsigned long long	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_UnsignedLongLong unsigned_long_long_member; } PrimitiveStruct;</pre>
float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Float float_member; } PrimitiveStruct;</pre>
double	<pre>struct PrimitiveStruct { double double_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Double double_member; } PrimitiveStruct;</pre>
long double (see Note: 2 below)	<pre>struct PrimitiveStruct { long double long_double_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_LongDouble long_double_member; } PrimitiveStruct;</pre>
pointer (see Note: 9 below)	<pre>struct MyStruct { long * member; };</pre>	<pre>typedef struct MyStruct { DDS_Long * member; } MyStruct;</pre>
boolean	<pre>struct PrimitiveStruct { boolean boolean_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Boolean boolean_member; } PrimitiveStruct;</pre>
enum	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 }; enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre>typedef enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 } PrimitiveEnum; typedef enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 } PrimitiveEnum;</pre>

Table 3.5 Specifying Data Types in IDL for C

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
constant	<pre>const short SIZE = 5;</pre>	<pre>#define SIZE 5</pre>
struct (see Note: 10 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>typedef struct PrimitiveStruct { char char_member; } PrimitiveStruct;</pre>
union (see Note: 3 and Note: 10 below)	<pre>union PrimitiveUnion switch (long){ case 1: short short_member; default: long long_member; };</pre>	<pre>typedef struct PrimitiveUnion { DDS_Long _d; struct { DDS_Short short_member; DDS_Long long_member; } _u; } PrimitiveUnion;</pre>
typedef	<pre>typedef short TypedefShort;</pre>	<pre>typedef DDS_Short TypedefShort;</pre>
array of above types	<pre>struct OneDArrayStruct { short short_array[2]; }; struct TwoDArrayStruct { short short_array[1][2]; };</pre>	<pre>typedef struct OneDArrayStruct { DDS_Short short_array[2]; } OneDArrayStruct; typedef struct TwoDArrayStruct { DDS_Short short_array[1][2]; } TwoDArrayStruct;</pre>
bounded sequence of above types (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short,4> short_sequence; };</pre>	<pre>typedef struct SequenceStruct { DDSShortSeq short_sequence; } SequenceStruct;</pre> <p>Note: Sequences of primitive types have been predefined by Connex DDS.</p>

Table 3.5 Specifying Data Types in IDL for C

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unbounded sequence of above types (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short> short_sequence; };</pre>	<pre>typedef struct SequenceStruct { DDSShortSeq short_sequence; } SequenceStruct;</pre> <p>See Note: 12 below.</p>
array of sequences	<pre>struct ArraysOfSequences{ sequence<short,4> sequences_array[2]; };</pre>	<pre>typedef struct ArraysOfSequences { DDS_ShortSeq sequences_array[2]; } ArraysOfSequences;</pre>
sequence of arrays (see Note: 11 below)	<pre>typedef short ShortArray[2]; struct SequenceOfArrays { sequence<ShortArray,2> arrays_sequence; };</pre>	<pre>typedef DDS_Short ShortArray[2]; DDS_SEQUENCE_NO_GET(ShortArraySeq, ShortArray); typedef struct SequenceOfArrays { ShortArraySeq arrays_sequence; } SequenceOfArrays;</pre> <p>DDS_SEQUENCE_NO_GET is a Connex DDS macro that defines a new sequence type for a user data type. In this case, the user data type is ShortArray.</p>
sequence of sequences (see Note: 4 and Note: 11 below)	<pre>typedef sequence<short,4> ShortSequence; struct SequencesOfSequences{ sequence<ShortSequence,2> sequences_sequence; };</pre>	<pre>typedef DDS_ShortSeq ShortSequence; DDS_SEQUENCE(ShortSequenceSeq, ShortSequence); typedef struct SequencesOfSequences{ ShortSequenceSeq sequences_sequence; } SequencesOfSequences;</pre>
bounded string	<pre>struct PrimitiveStruct { string<20> string_member; };</pre>	<pre>typedef struct PrimitiveStruct { char* string_member; /* maximum length = (20) */ } PrimitiveStruct;</pre>

Table 3.5 Specifying Data Types in IDL for C

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unbounded string	<pre>struct PrimitiveStruct { string string_member; };</pre>	<pre>typedef struct PrimitiveStruct { char* string_member; /* maximum length = (255) */ } PrimitiveStruct;</pre> <p>See Note: 12 below.</p>
bounded wstring	<pre>struct PrimitiveStruct { wstring<20> wstring_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Wchar * wstring_member; /* maximum length = (20) */ } PrimitiveStruct;</pre>
unbounded wstring	<pre>struct PrimitiveStruct { wstring wstring_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Wchar * wstring_member; /* maximum length = (255) */ } PrimitiveStruct;</pre> <p>See Note: 12 below.</p>
module	<pre>module PackageName { struct Foo { long field; }; };</pre>	<p>With the -namespace option (only available for C++):</p> <pre>namespace PackageName{ typedef struct Foo { DDS_Long field; } Foo; };</pre> <p>Without the -namespace option:</p> <pre>typedef struct PackageName_Foo { DDS_Long field; } PackageName_Foo;</pre>

Table 3.5 Specifying Data Types in IDL for C

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
valuetype (see Note: 9 and Note: 10 below)	<pre>valuetype MyValueType { public MyValueType2 * member; }; valuetype MyValueType { public MyValueType2 member; }; valuetype MyValueType: MyBaseValueType { public MyValueType2 * member; };</pre>	<pre>typedef struct MyValueType { MyValueType2 * member; } MyValueType; typedef struct MyValueType { MyValueType2 member; } MyValueType; typedef struct MyValueType { MyBaseValueType parent; MyValueType2 * member; } MyValueType;</pre>

Table 3.6 Specifying Data Types in IDL for Traditional C++

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
char (see Note: 1 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>class PrimitiveStruct { DDS_Char char_member; } PrimitiveStruct;</pre>
wchar	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre>class PrimitiveStruct { DDS_Wchar wchar_member; } PrimitiveStruct;</pre>
octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre>class PrimitiveStruct { DDS_Octet octect_member; } PrimitiveStruct;</pre>
short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre>class PrimitiveStruct { DDS_Short short_member; } PrimitiveStruct;</pre>

Table 3.6 Specifying Data Types in IDL for Traditional C++

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unsigned short	<pre>struct PrimitiveStruct { unsigned short unsigned_short_member; };</pre>	<pre>class PrimitiveStruct { DDS_UnsignedShort unsigned_short_member; } PrimitiveStruct;</pre>
long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre>class PrimitiveStruct { DDS_Long long_member; } PrimitiveStruct;</pre>
unsigned long	<pre>struct PrimitiveStruct { unsigned long unsigned_long_member; };</pre>	<pre>class PrimitiveStruct { DDS_UnsignedLong unsigned_long_member; } PrimitiveStruct;</pre>
long long	<pre>struct PrimitiveStruct { long long long_long_member; };</pre>	<pre>class PrimitiveStruct { DDS_LongLong long_long_member; } PrimitiveStruct;</pre>
unsigned long long	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_member; };</pre>	<pre>class PrimitiveStruct { DDS_UnsignedLongLong unsigned_long_long_member; } PrimitiveStruct;</pre>
float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre>typedef struct PrimitiveStruct { DDS_Float float_member; } PrimitiveStruct;</pre>
double	<pre>struct PrimitiveStruct { double double_member; };</pre>	<pre>class PrimitiveStruct { DDS_Double double_member; } PrimitiveStruct;</pre>
long double (see Note: 2 below)	<pre>struct PrimitiveStruct { long double long_double_member; };</pre>	<pre>class PrimitiveStruct { DDS_LongDouble long_double_member; } PrimitiveStruct;</pre>

Table 3.6 Specifying Data Types in IDL for Traditional C++

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
pointer (see Note: 9 below)	<pre>struct MyStruct { long * member; };</pre>	<pre>class MyStruct { DDS_Long * member; } MyStruct;</pre>
boolean	<pre>struct PrimitiveStruct { boolean boolean_member; };</pre>	<pre>class PrimitiveStruct { DDS_Boolean boolean_member; } PrimitiveStruct;</pre>
enum	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 }; enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre>typedef enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 } PrimitiveEnum; typedef enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 } PrimitiveEnum;</pre>
constant	<pre>const short SIZE = 5;</pre>	<pre>static const DDS_Short size = 5;</pre>
struct (see Note: 10 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>typedef struct PrimitiveStruct { char char_member; } PrimitiveStruct;</pre>
union (see Note: 3 and Note: 10 below)	<pre>union PrimitiveUnion switch (long){ case 1: short short_member; default: long long_member; };</pre>	<pre>class PrimitiveUnion { DDS_Long _d; class{ DDS_Short short_member; DDS_Long long_member; } _u; } PrimitiveUnion;</pre>
typedef	<pre>typedef short TypedefShort;</pre>	<pre>typedef DDS_Short TypedefShort;</pre>

Table 3.6 Specifying Data Types in IDL for Traditional C++

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
array of above types	<pre> struct OneDArrayStruct { short short_array[2]; }; struct TwoDArrayStruct { short short_array[1][2]; }; </pre>	<pre> class OneDArrayStruct { DDS_Short short_array[2]; } OneDArrayStruct; class TwoDArrayStruct { DDS_Short short_array[1][2]; } TwoDArrayStruct; </pre>
bounded sequence of above types (see Note: 11 and Note: 15 below)	<pre> struct SequenceStruct { sequence<short, 4> short_sequence; }; </pre>	<pre> class SequenceStruct { DDSShortSeq short_sequence; } SequenceStruct; </pre> <p>Note: Sequences of primitive types have been predefined by Connex DDS.</p>
unbounded sequence of above types (see Note: 11 and Note: 15 below)	<pre> struct SequenceStruct { sequence<short> short_sequence; }; </pre>	<pre> typedef struct SequenceStruct { DDSShortSeq short_sequence; } SequenceStruct; </pre> <p>See Note: 12 below.</p>
array of sequences	<pre> struct ArraysOfSequences{ sequence<short, 4> sequences_array[2]; }; </pre>	<pre> class ArraysOfSequences { DDS_ShortSeq sequences_array[2]; } ArraysOfSequences; </pre>

Table 3.6 Specifying Data Types in IDL for Traditional C++

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
sequence of arrays (see Note: 11 below)	<pre>typedef short ShortArray[2]; struct SequenceofArrays { sequence<ShortArray,2> arrays_sequence; };</pre>	<pre>typedef DDS_Short ShortArray[2]; DDS_SEQUENCE_NO_GET(ShortArraySeq, ShortArray); class SequenceOfArrays { ShortArraySeq arrays_sequence; } SequenceOfArrays;</pre> <p>DDS_SEQUENCE_NO_GET is a Connex DDS macro that defines a new sequence type for a user data type. In this case, the user data type is ShortArray.</p>
sequence of sequences (see Note: 4 and Note: 11 below)	<pre>typedef sequence<short,4> ShortSequence; struct SequencesOfSequences{ sequence<ShortSequence,2> sequences_sequence; };</pre>	<pre>typedef DDS_ShortSeq ShortSequence; DDS_SEQUENCE(ShortSequenceSeq, ShortSequence); class SequencesOfSequences{ ShortSequenceSeq sequences_sequence; } SequencesOfSequences;</pre>
bounded string	<pre>struct PrimitiveStruct { string<20> string_member; };</pre>	<pre>class PrimitiveStruct { char* string_member; /* maximum length = (20) */ } PrimitiveStruct;</pre>
unbounded string	<pre>struct PrimitiveStruct { string string_member; };</pre>	<pre>class PrimitiveStruct { char* string_member; /* maximum length = (255) */ } PrimitiveStruct;</pre> <p>See Note: 12 below.</p>
bounded wstring	<pre>struct PrimitiveStruct { wstring<20> wstring_member; };</pre>	<pre>class PrimitiveStruct { DDS_Wchar * wstring_member; /* maximum length = (20) */ } PrimitiveStruct;</pre>

Table 3.6 Specifying Data Types in IDL for Traditional C++

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unbounded wstring	<pre>struct PrimitiveStruct { wstring wstring_member; };</pre>	<pre>class PrimitiveStruct { DDS_Wchar * wstring_member; /* maximum length = (255) */ } PrimitiveStruct;</pre> <p>See Note: 12 below.</p>
module	<pre>module PackageName { struct Foo { long field; }; };</pre>	<p>With the -namespace option (only available for C++):</p> <pre>namespace PackageName{ typedef struct Foo { DDS_Long field; } Foo; };</pre> <p>Without the -namespace option:</p> <pre>class PackageName_Foo { DDS_Long field; } PackageName_Foo;</pre>
valuetype (see Note: 9 and Note: 10 below)	<pre>valuetype MyValueType { public MyValueType2 * member; }; valuetype MyValueType { public MyValueType2 member; }; valuetype MyValueType: MyBaseValueType { public MyValueType2 * member; };</pre>	<pre>class MyValueType { public: MyValueType2 * member; }; class MyValueType { public: MyValueType2 member; }; class MyValueType : public MyBaseValueType { public: MyValueType2 * member; };</pre>

Table 3.7 Specifying Data Types in IDL for C++/CLI

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
char (see Note: 1 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Char char_member; };</pre>
wchar	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Char wchar_member; };</pre>
octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Byte octet_member; };</pre>
short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Int16 short_member; };</pre>
unsigned short	<pre>struct PrimitiveStruct { unsigned short unsigned_short_member; };</pre>	<pre>public ref class PrimitiveStruct { System::UInt16 unsigned_short_member; };</pre>
long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Int32 long_member; };</pre>
unsigned long	<pre>struct PrimitiveStruct { unsigned long unsigned_long_member; };</pre>	<pre>public ref class PrimitiveStruct { System::UInt32 unsigned_long_member; };</pre>
long long	<pre>struct PrimitiveStruct { long long long_ long_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Int64 long_long_member; };</pre>
unsigned long long	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_member; };</pre>	<pre>public ref class PrimitiveStruct { System::UInt64 unsigned_long_long_member; };</pre>

Table 3.7 Specifying Data Types in IDL for C++/CLI

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Single float_member; };</pre>
double	<pre>struct PrimitiveStruct { double double_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Double double_member; } PrimitiveStruct;</pre>
long double (see Note: 2 below)	<pre>struct PrimitiveStruct { long double long_double_member; };</pre>	<pre>public ref class PrimitiveStruct { DDS::LongDouble long_double_member; } PrimitiveStruct;</pre>
boolean	<pre>struct PrimitiveStruct { boolean boolean_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Boolean boolean_member; };</pre>
enum	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 }; enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre>public enum class PrimitiveEnum : System::Int32 { ENUM1, ENUM2, ENUM3 }; public enum class PrimitiveEnum : System::Int32 { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>
constant	<pre>const short SIZE = 5;</pre>	<pre>public ref class SIZE { public: static System::Int16 VALUE = 5; };</pre>
struct (see Note: 10 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>public ref class PrimitiveStruct { System::Char char_member; };</pre>

Table 3.7 Specifying Data Types in IDL for C++/CLI

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
union (see Note: 3 and Note: 10 below)	<pre>union PrimitiveUnion switch (long) { case 1: short short_member; default: long long_member; };</pre>	<pre>public ref class PrimitiveUnion { System::Int32 _d; struct PrimitiveUnion_u { System::Int16 short_member; System::Int32 long_member; } _u; };</pre>
array of above types	<pre>struct OneDArrayStruct { short short_array[2]; };</pre>	<pre>public ref class OneDArrayStruct { array<System::Int16>^ short_array; /*length == 2*/ };</pre>
bounded sequence of above types (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short,4> short_sequence; };</pre>	<pre>public ref class SequenceStruct { ShortSeq^ short_sequence; /*max = 4*/ };</pre> <p>Note: Sequences of primitive types have been predefined by Connext DDS</p>
unbounded sequence of above types (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short> short_sequence; };</pre>	<pre>public ref class SequenceStruct { ShortSeq^ short_sequence; /*max = <default bound>*/ };</pre> <p>See Note: 12 below.</p>
array of sequences	<pre>struct ArraysOfSequences{ sequence<short,4> sequences_array[2]; };</pre>	<pre>public ref class ArraysOfSequences { array<DDS::ShortSeq^>^ sequences_array; // maximum length = (2) };</pre>
bounded string	<pre>struct PrimitiveStruct { string<20> string_member; };</pre>	<pre>public ref class PrimitiveStruct { System::String^ string_member; // maximum length = (20) };</pre>

Table 3.7 Specifying Data Types in IDL for C++/CLI

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unbounded string	<pre>struct PrimitiveStruct { string string_member; };</pre>	<pre>public ref class PrimitiveStruct { System::String^ string_member; // maximum length = (255) };</pre> <p>See Note: 12 below.</p>
bounded wstring	<pre>struct PrimitiveStruct { wstring<20> wstring_member; };</pre>	<pre>public ref class PrimitiveStruct { System::String^ string_member; // maximum length = (20) };</pre>
unbounded wstring	<pre>struct PrimitiveStruct { wstring wstring_member; };</pre>	<pre>public ref class PrimitiveStruct { System::String^ string_member; // maximum length = (255) };</pre> <p>See Note: 12 below.</p>
module	<pre>module PackageName { struct Foo { long field; }; };</pre>	<pre>namespace PackageName { public ref class Foo { System::Int32 field; }; };</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
char (see Note: 1 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>class PrimitiveStruct { public: char char_member() const OMG_NOEXCEPT; void char_member(char value); };</pre>
wchar	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre>class PrimitiveStruct { public: DDS_Wchar wchar_member() const OMG_NOEXCEPT; void wchar_member(DDS_Wchar value); };</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre>class PrimitiveStruct { public: uint8_t octet_member() const OMG_NOEXCEPT; void octet_member(uint8_t value); };</pre>
short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre>class PrimitiveStruct { public: int16_t short_member() const OMG_NOEXCEPT; void short_member(int16_t value); };</pre>
unsigned short	<pre>struct PrimitiveStruct { unsigned short unsigned_short_ member; };</pre>	<pre>class PrimitiveStruct { public: uint16_t unsigned_short_member() const OMG_NOEXCEPT; void unsigned_short_member(uint16_t value); };</pre>
long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre>class PrimitiveStruct { public: int32_t long_member() const OMG_NOEXCEPT; void long_member(int32_t value); };</pre>
unsigned long	<pre>struct PrimitiveStruct { unsigned long unsigned_long_ member; };</pre>	<pre>class PrimitiveStruct { public: uint32_t long_member() const OMG_NOEXCEPT; void unsigned_long_member(uint32_t value); };</pre>
long long	<pre>struct PrimitiveStruct { long long long_long_member; };</pre>	<pre>class PrimitiveStruct { public: rti::core::int64 long_long_member() const OMG_NOEXCEPT; void long_long_member(rti::core::int64 value); };</pre>
unsigned long long	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_ member; };</pre>	<pre>class PrimitiveStruct { public: rti::core::uint64 unsigned_long_long_member; rti::core::uint64 unsigned_long_long_member() const OMG_NOEXCEPT; };</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre>class PrimitiveStruct { public: float float_member() const OMG_NOEXCEPT; void float_member(float value); };</pre>
double	<pre>struct PrimitiveStruct { double double_ member; };</pre>	<pre>class PrimitiveStruct { public: double double_member() const OMG_NOEXCEPT; void double_member(double value); };</pre>
long double (see Note: 2 below)	<pre>struct PrimitiveStruct { long double long_ double_member; };</pre>	<pre>class PrimitiveStruct { public: rti::core::LongDouble& long_double_member() OMG_NOEXCEPT; const rti::core::LongDouble& long_double_member() const OMG_ NOEXCEPT; void long_double_member(const rti::core::LongDouble& value); }</pre>
pointer (see Note: 9 below)	<pre>struct MyStruct { long * member; };</pre>	<pre>class PrimitiveStruct { public: int32_t * member() const OMG_NOEXCEPT; void member(int32_t * value); };</pre>
boolean	<pre>struct PrimitiveStruct { boolean boolean_ member; };</pre>	<pre>class PrimitiveStruct { public: bool boolean_member() const OMG_NOEXCEPT; void boolean_member(bool value); };</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
enum	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 }; enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre>struct PrimitiveEnum_def { enum type { ENUM1, ENUM2, ENUM3 }; }; typedef dds::core::safe_enum<PrimitiveEnum_def> PrimitiveEnum; struct PrimitiveEnum_def { enum type { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 }; }; typedef dds::core::safe_enum<PrimitiveEnum_def> PrimitiveEnum;</pre>
constant	<pre>const short SIZE = 5;</pre>	<pre>static const int16_t SIZE = 5;</pre>
struct (see Note: 10 and Note: 14 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>class PrimitiveStruct { public: char char_member() const OMG_NOEXCEPT; void char_member(char value); }</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
union (see Note: 3 and Note: 10 below)	<pre>union PrimitiveUnion switch (long){ case 1: short short_ member; default: long long_ member; };</pre>	<pre>class PrimitiveUnion { public: int32_t _d() const ; void _d(int32_t value); int16_t short_member() const ; void short_member(int16_t value); int32_t long_member() const ; void long_member(int32_t value); static int32_t default_discriminator(); private: int32_t m_d; struct Union_ { int16_t m_short_member_; int32_t m_long_member_; Union_(); Union_(int16_t short_member, int32_t long_member); }; Union_ m_u; };</pre>
typedef	<pre>typedef short TypedefShort;</pre>	<pre>typedef int16_t TypedefShort; struct TypedefShort_AliasTag_t {};</pre>
array of above types	<pre>struct OneDArrayStruct { short short_array[2]; }; struct TwoDArrayStruct { short short_array[1] [2]; };</pre>	<pre>class OneDArrayStruct { public: dds::core::array<int16_t, 2>& short_array() OMG_NOEXCEPT; const dds::core::array<int16_t, 2>& short_array() const OMG_NOEXCEPT; void short_array(const dds::core::array<int16_t, 2>& value); }; class TwoDArrayStruct { public: dds::core::array<dds::core::array<int16_t, 2>, 1>& short_array() OMG_NOEXCEPT; const dds::core::array<dds::core::array<int16_t, 2>, 1>& short_ array() const OMG_NOEXCEPT; void short_array(const dds::core::array<dds::core::array<int16_t, 2>, 1>& value); };</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
bounded sequence of above types (see Note: 11 below)	<pre>struct SequenceStruct { sequence<short, 4> short_sequence; };</pre>	<pre>class SequenceStruct { public: dds::core::vector<int16_t>& short_sequence() OMG_NOEXCEPT; const dds::core::vector<int16_t>& short_sequence() const OMG_NOEXCEPT; void short_sequence(const dds::core::vector<int16_t>& value); };</pre>
unbounded sequence of above types (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short> short_sequence; };</pre>	<pre>class SequenceStruct { public: dds::core::vector<int16_t>& short_sequence() OMG_NOEXCEPT; const dds::core::vector<int16_t>& short_sequence() const OMG_NOEXCEPT; void short_sequence(const dds::core::vector<int16_t>& value); };</pre> <p>See Note: 12 below.</p>
array of sequences	<pre>struct ArraysOfSequences { sequence<short, 4> sequences_array [2]; };</pre>	<pre>class ArraysOfSequences { public: dds::core::array<dds::core::vector<int16_t>, 2>& sequences_array() OMG_NOEXCEPT; const dds::core::array<dds::core::vector<int16_t>, 2>& sequences_ array() const OMG_NOEXCEPT; void sequences_array(const dds::core::array<dds::core::vector<int16_t>, 2>& value); };</pre>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
sequence of arrays (see Note: 1 and Note: 15 below)	<pre>typedef short ShortArray [2]; struct SequenceofArrays { sequence<ShortArray,2> arrays_sequence; };</pre>	<pre>typedef dds::core::array<int16_t, 2> ShortArray; class SequenceofArrays { public: dds::core::vector<ShortArray>& arrays_sequence() OMG_NOEXCEPT; const dds::core::vector<ShortArray>& arrays_sequence() const OMG_NOEXCEPT; void arrays_sequence(const dds::core::vector<ShortArray>& value); };</pre>
sequence of sequences (see Note: 4 and Note: 11 below)	<pre>typedef sequence<short,4> ShortSequence; struct SequencesOfSequences{ sequence<ShortSequence, 2> sequences_ sequence; };</pre>	<pre>typedef dds::core::vector<int16_t> ShortSequence; class SequencesOfSequences { public: dds::core::vector<ShortSequence>& sequences_sequence() OMG_NOEXCEPT; const dds::core::vector<ShortSequence>& sequences_sequence() const OMG_NOEXCEPT; void sequences_sequence(const dds::core::vector<ShortSequence>& value); };</pre>
bounded string	<pre>struct PrimitiveStruct { string<20> string_ member; };</pre>	<pre>class PrimitiveStruct { public: dds::core::string& string_member() OMG_NOEXCEPT; const dds::core::string& string_member() const OMG_NOEXCEPT; void string_member(const dds::core::string& value); };</pre>
unbounded string	<pre>struct PrimitiveStruct { string string_ member; };</pre>	<pre>class PrimitiveStruct { public: dds::core::string& string_member() OMG_NOEXCEPT; const dds::core::string& string_member() const OMG_NOEXCEPT; void string_member(const dds::core::string& value); };</pre> <p>See Note: 12 below.</p>

Table 3.8 Specifying Data Types in IDL for the Modern C++ API

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
bounded wstring	<pre>struct PrimitiveStruct { wstring<20> wstring_ member; };</pre>	<pre>class PrimitiveStruct { public: dds::core::wstring& wstring_member() OMG_NOEXCEPT; const dds::core::wstring& wstring_member() const OMG_NOEXCEPT; void wstring_member(const dds::core::wstring& value); };</pre>
unbounded wstring	<pre>struct PrimitiveStruct { wstring wstring_ member; };</pre>	<pre>class PrimitiveStruct { public: dds::core::wstring& wstring_member() OMG_NOEXCEPT; const dds::core::wstring& wstring_member() const OMG_NOEXCEPT; void wstring_member(const dds::core::wstring& value); };</pre> <p>See Note: 12 below.</p>
module	<pre>module PackageName { struct Foo { long field; }; };</pre>	<pre>namespace PackageName { class Foo { public: int32_t field() const OMG_NOEXCEPT; void field(int32_t value); }; };</pre>
value type (see Note: 9 and Note: 10 below)	<pre>valuetype MyBaseValueType { public long member; }; valuetype MyValueType: MyBaseValueType { public short * member2; };</pre>	<pre>class MyBaseValueType { public: int32_t member() const OMG_NOEXCEPT; void member(int32_t value); }; class MyValueType : public MyBaseValueType { public: int16_t * member2() const OMG_NOEXCEPT; void member2(int16_t * value); };</pre>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
char (see Note: 5 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>public class PrimitiveStruct { public char char_member; ... }</pre>
wchar (see Note: 5 below)	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre>public class PrimitiveStruct { public char wchar_member; ... }</pre>
octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre>public class PrimitiveStruct { public byte byte_member; ... }</pre>
short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre>public class PrimitiveStruct { public short short_member; ... }</pre>
unsigned short (see Note: 6 below)	<pre>struct PrimitiveStruct { unsigned short unsigned_short_member; };</pre>	<pre>public class PrimitiveStruct { public short unsigned_short_member; ... }</pre>
long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre>public class PrimitiveStruct { public int long_member; ... }</pre>
unsigned long (see Note: 6 below)	<pre>struct PrimitiveStruct { unsigned long unsigned_long_member; };</pre>	<pre>public class PrimitiveStruct { public int unsigned_long_member; ... }</pre>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
long long	<pre>struct PrimitiveStruct { long long long_long_member; };</pre>	<pre>public class PrimitiveStruct { public long long_long_member; ... }</pre>
unsigned long long (see Note: 7 below)	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_member; };</pre>	<pre>public class PrimitiveStruct { public long unsigned_long_long_member; ... }</pre>
float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre>public class PrimitiveStruct { public float float_member; ... }</pre>
double	<pre>struct PrimitiveStruct { double double_member; };</pre>	<pre>public class PrimitiveStruct { public double double_member; ... }</pre>
long double (see Note: 7 below)	<pre>struct PrimitiveStruct { long double long_double_member; };</pre>	<pre>public class PrimitiveStruct { public double long_double_member; ... }</pre>
pointer (see Note: 9 below)	<pre>struct MyStruct { long * member; };</pre>	<pre>public class MyStruct { public int member; ... };</pre>
boolean	<pre>struct PrimitiveStruct { boolean boolean_member; };</pre>	<pre>public class PrimitiveStruct { public boolean boolean_member; ... }</pre>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
enum	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 };</pre>	<pre>public class PrimitiveEnum extends Enum { public static PrimitiveEnum ENUM1 = new PrimitiveEnum ("ENUM1", 0); public static PrimitiveEnum ENUM2 = new PrimitiveEnum ("ENUM2", 1); public static PrimitiveEnum ENUM3 = new PrimitiveEnum ("ENUM3", 2); public static PrimitiveEnum valueOf(int ordinal); ... }</pre>
	<pre>enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre>public class PrimitiveEnum extends Enum { public static PrimitiveEnum ENUM1 = new PrimitiveEnum ("ENUM1", 10); public static PrimitiveEnum ENUM2 = new PrimitiveEnum ("ENUM2", 10); public static PrimitiveEnum ENUM3 = new PrimitiveEnum ("ENUM3", 20); public static PrimitiveEnum valueOf(int ordinal); ... }</pre>
constant	<pre>const short SIZE = 5;</pre>	<pre>public class SIZE { public static final short VALUE = 5; }</pre>
struct (see Note: 10 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>public class PrimitiveStruct { public char char_member; }</pre>
union (see Note: 10 below)	<pre>union PrimitiveUnion switch (long){ case 1: short short_member; default: long long_member; };</pre>	<pre>public class PrimitiveUnion { public int _d; public short short_member; public int long_member; ... }</pre>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
typedef of primitives, enums, strings (see Note: 8 below)	<pre>typedef short ShortType; struct PrimitiveStruct { ShortType short_member; };</pre>	<pre>/* typedefs are unwounded to the original type when used */ public class PrimitiveStruct { public short short_member; ... }</pre>
typedef of sequences or arrays (see Note: 8 below)	<pre>typedef short ShortArray[2];</pre>	<pre>/* Wrapper class */ public class ShortArray { public short[] userData = new short[2]; ... }</pre>
array	<pre>struct OneDArrayStruct { short short_array[2]; };</pre>	<pre>public class OneDArrayStruct { public short[] short_array = new short[2]; ... }</pre>
	<pre>struct TwoDArrayStruct { short short_array[1][2]; };</pre>	<pre>public class TwoDArrayStruct { public short[][] short_array = new short[1][2]; ... }</pre>
bounded sequence (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short,4> short_sequence; };</pre>	<pre>public class SequenceStruct { public ShortSeq short_sequence = new ShortSeq((4)); ... }</pre> <p>Note: Sequences of primitive types have been predefined by Connex DDS.</p>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
unbounded sequence (see Note: 11 and Note: 15 below)	<pre>struct SequenceStruct { sequence<short> short_sequence; };</pre>	<pre>public class SequenceStruct { public ShortSeq short_sequence = new ShortSeq((100)); ... }</pre> <p>See Note: 12 below.</p>
array of sequences	<pre>struct ArraysOfSequences{ sequence<short,4> sequences_array[2]; };</pre>	<pre>public class ArraysOfSequences { public ShortSeq[] sequences_array = new ShortSeq[2]; ... }</pre>
sequence of arrays (see Note: 11 below)	<pre>typedef short ShortArray[2]; struct SequenceOfArrays{ sequence<ShortArray,2> arrays_sequence; };</pre>	<pre>/* Wrapper class */ public class ShortArray { public short[] userData = new short[2]; ... } /* Sequence of wrapper class objects */ public final class ShortArraySeq extends ArraySequence { ... } public class SequenceOfArrays { public ShortArraySeq arrays_sequence = new ShortArraySeq((2)); ... }</pre>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
sequence of sequences (see Note: 4 and Note: 11 below)	<pre>typedef sequence<short,4> ShortSequence; struct SequencesOfSequences{ sequence<ShortSequence,2> sequences_sequence; };</pre>	<pre>/* Wrapper class */ public class ShortSequence { public ShortSeq userData = new ShortSeq((4)); ... } /* Sequence of wrapper class objects */ public final class ShortSequenceSeq extends ArraySequence { ... } public class SequencesOfSequences { public ShortSequenceSeq sequences_sequence = new ShortSequenceSeq((2)); ... }</pre>
bounded string	<pre>struct PrimitiveStruct { string<20> string_member; };</pre>	<pre>public class PrimitiveStruct { public String string_member = new String(); /* maximum length = (20) */ ... }</pre>
unbounded string	<pre>struct PrimitiveStruct { string string_member; };</pre>	<pre>public class PrimitiveStruct { public String string_member = new String(); * maximum length = (255) */ ... }</pre> <p>See Note: 12 below.</p>
bounded wstring	<pre>struct PrimitiveStruct { wstring<20> wstring_member; };</pre>	<pre>public class PrimitiveStruct { public String wstring_member = new String(); /* maximum length = (20) */ ... }</pre>

Table 3.9 Specifying Data Types in IDL for Java

IDL Type	Example Entry in IDL file	Example Java Output Generated by RTI Code Generator (rtiddsgen)
unbounded wstring	<pre>struct PrimitiveStruct { wstring wstring_member; };</pre>	<pre>public class PrimitiveStruct { public String wstring_member = new String(); /* maximum length = (255) */ ... }</pre> <p>See Note: 12 below.</p>
module	<pre>module PackageName { struct Foo { long field; }; };</pre>	<pre>package PackageName; public class Foo { public int field; ... }</pre>
valuetype (see Note: 9 and Note: 10 below)	<pre>valuetype MyValueType { public MyValueType2 * member; }; valuetype MyValueType { public MyValueType2 member; }; valuetype MyValueType: MyBaseValueType { public MyValueType2 * member; };</pre>	<pre>public class MyValueType { public MyValueType2 member; ... }; public class MyValueType { public MyValueType2 member; ... }; public class MyValueType extends MyBaseValueType { public MyValueType2 member; ... }</pre>

Table 3.10 Specifying Data Types in IDL for Ada

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
char (see Note: 13 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>type PrimitiveStruct is record char_member : aliased Standard.DDS.Char; end record;</pre>
wchar	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre>type PrimitiveStruct is record wchar_member : aliased Standard.DDS.Wchar; end record;</pre>
octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre>type PrimitiveStruct is record octet_member: aliased Standard.DDS.Octet; end record;</pre>
short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre>type PrimitiveStruct is record short_member: aliased Standard.DDS.Short; end record;</pre>
unsigned short	<pre>struct PrimitiveStruct { unsigned short unsigned_short_member; };</pre>	<pre>type PrimitiveStruct is record unsigned_short_member: aliased Standard.DDS.Unsigned_Short; end record;</pre>
long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre>type PrimitiveStruct is record long_member: aliased Standard.DDS.Long; end record;</pre>
unsigned long	<pre>struct PrimitiveStruct { unsigned long unsigned_long_member; };</pre>	<pre>type PrimitiveStruct is record unsigned_long_member: aliased Standard.DDS.Unsigned_Long; end record;</pre>

Table 3.10 Specifying Data Types in IDL for Ada

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
long long	<pre>struct PrimitiveStruct { long long long_long_ member; };</pre>	<pre>type PrimitiveStruct is record long_long_member: aliased Standard.DDS.Long_Long; end record;</pre>
unsigned long long	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_ long_member; };</pre>	<pre>type PrimitiveStruct is record unsigned_long_long_member: aliased Standard.DDS.Unsigned_Long_Long; end record;</pre>
float	<pre>struct PrimitiveStruct { float float_ member; };</pre>	<pre>type PrimitiveStruct is record float_member: aliased Standard.DDS.Float; end record;</pre>
double	<pre>struct PrimitiveStruct { double double_ member; };</pre>	<pre>type PrimitiveStruct is record double_member: aliased Standard.DDS.Double; end record;</pre>
long double (see Note: 2 below)	<pre>struct PrimitiveStruct { long double long_double_ member; };</pre>	<pre>type PrimitiveStruct is record long_double_member: aliased Standard.DDS.Long_Double; end record;</pre>
pointer (see Note: 9 below)	<pre>struct MyStruct { long * member; };</pre>	<pre>type MyStruct is record member : access Standard.DDS.Long; end record;</pre>
boolean	<pre>struct PrimitiveStruct { boolean boolean_ member; };</pre>	<pre>type PrimitiveStruct is record boolean_member: aliased Standard.DDS.Boolean; end record;</pre>

Table 3.10 Specifying Data Types in IDL for Ada

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
enum	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 }; enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre>type PrimitiveEnum is (ENUM1, ENUM2, ENUM3); type PrimitiveEnum is (ENUM1, ENUM2, ENUM3); ... for PrimitiveEnum use (ENUM1 => 10 , ENUM2 => 20 , ENUM3 => 30);</pre>
constant	<pre>const short SIZE = 5;</pre>	<pre>SIZE : constant Standard.DDS.Short := 5;</pre>
struct (see Note: 10 below)	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre>type PrimitiveStruct is record char_member : aliased Standard.DDS.Char; end record;</pre>
union (see Note: 3 and Note: 10 below)	<pre>union PrimitiveUnion switch (long){ case 1: short short_member; default: long long_member; };</pre>	<pre>type U_PrimitiveUnion is record short_member : aliased Standard.DDS.Short; long_member : aliased Standard.DDS.Long; end record; type PrimitiveUnion is record d : Standard.DDS.Long; u : U_PrimitiveUnion; end record;</pre>
typedef	<pre>typedef short TypedefShort;</pre>	<pre>type TypedefShort is new Standard.DDS.Short;</pre>
array of above types	<pre>struct OneDArrayStruct { short short_array [2]; }; struct TwoDArrayStruct { short short_array [1][2]; };</pre>	<pre>type OneDArrayStruct is record short_array : aliased Standard.DDS.Short_Array(1..2); end record; type TwoDArrayStruct_short_array_Array is array (1..1, 1..2) of aliased Standard.DDS.Short; type TwoDArrayStruct is record short_array : aliased TwoDArrayStruct_short_array_Array; end record;</pre>

Table 3.10 Specifying Data Types in IDL for Ada

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
bounded sequence of above types (see Note: 11 and Note: 15 below)	<pre> struct SequenceStruct { sequence<short,4> short_sequence; }; </pre>	<pre> type SequenceStruct is record short_sequence : aliased Standard.DDS.Short_Seq.Sequence; end record; </pre>
unbounded sequence of above types (see Note: 11 and Note: 15 below)	<pre> struct SequenceStruct { sequence<short> short_sequence; }; </pre>	<pre> type SequenceStruct is record short_sequence : aliased Standard.DDS.Short_Seq.Sequence; end record; </pre> <p>See Note: 13 below.</p>
array of sequences	<pre> struct ArrayOfSequences{ sequence<short,4> sequences_array [2]; }; </pre>	<pre> type ArraysOfSequences_sequences_array_Array is array (1..2) of aliased Standard.DDS.Short_Seq.Sequence; type ArraysOfSequences is record sequences_array : aliased ArraysOfSequences_sequences_array_Array; end record; </pre>

Table 3.10 Specifying Data Types in IDL for Ada

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
sequence of arrays (see Note: 1 below)	<pre>typedef short ShortArray[2]; struct SequenceofArrays { sequence<ShortArray,2> arrays_ sequence; };</pre>	<pre>type ShortArray is array (1..2) of Standard.DDS.Short; ... type SequenceofArrays is record arrays_sequence : aliased ADA_IDL_File.ShortArray_Seq.Sequence; end record;</pre> <p>Note: ADA_IDL_File.ShortArray_Seq.Sequence is an instantiation of Standard.DDS.Sequences_Generic for the user's data type</p>
sequence of sequences (see Note: 4 and Note: 1 below)	<pre>typedef sequence<short,4> ShortSequence; struct SequencesOfSequences{ sequence<ShortSequence ,2> sequences_ sequence; };</pre>	<pre>type ShortSequence is new Standard.DDS.Short_Seq.Sequence; ... type SequencesOfSequences is record sequences_sequence : aliased ADA_IDL_File.ShortSequence_Seq.Sequence; end record;</pre> <p>Note: ADA_IDL_File.ShortSequence_Seq.Sequence is an instantiation of Standard.DDS.Sequences_Generic for the user's data type</p>
bounded string	<pre>struct PrimitiveStruct { string<20> string_ member; };</pre>	<pre>type PrimitiveStruct is record string_member : aliased Standard.DDS.String; -- maximum length = (20) end record;</pre>
unbounded string	<pre>struct PrimitiveStruct { string string_ member; };</pre>	<pre>type PrimitiveStruct is record string_member : aliased Standard.DDS.String; -- maximum length = (255) end record;</pre>
bounded wstring	<pre>struct PrimitiveStruct { wstring<20> wstring_member; };</pre>	<pre>type PrimitiveStruct is record wstring_member : aliased Standard.DDS.Wide_String; -- maximum length = (20) end record;</pre>

Table 3.10 Specifying Data Types in IDL for Ada

IDL Type	Example Entry in IDL File	Example Output Generated by RTI Code Generator (rtiddsgen)
unbounded wstring	<pre>struct PrimitiveStruct { wstring wstring_ member; };</pre>	<pre>type PrimitiveStruct is record wstring_member : aliased Standard.DDS.Wide_String; -- maximum length = (255) end record;</pre>
module	<pre>module PackageName { struct Foo { long field; }; };</pre>	<pre>package PackageName is type Foo is record field : aliased Standard.DDS.Long; end record; end PackageName;</pre>
value type (see Note: 9 and Note: 10 below)	<pre>valuetype MyBaseValueType { valuetype MyBaseValueType { public long member; }; }; valuetype MyValueType: MyBaseValueType { public short * member2; };</pre>	<pre>type MyBaseValueType is record member : aliased Standard.DDS.Long; end record; type MyValueType is record parent : ADA_IDL_File.MyBaseValueType; member2 : access Standard.DDS.Short; end record;</pre>

Notes for [Table 3.5 Specifying Data Types in IDL for C](#) through [Table 3.9 Specifying Data Types in IDL for Java](#):

Note: 1: In C and C++, primitive types are not represented as native language types (e.g. long, char, etc.) but as custom types in the DDS namespace (DDS_Long, DDS_Char, etc.). These typedefs are used to ensure that a field's size is the same across platforms.

Note: 2: Some platforms do not support long double or have different sizes for that type than defined by IDL (16 bytes). On such platforms, DDS_LongDouble (as well as the unsigned version) is mapped to a character array that matches the expected size of that type by default. If you are using a platform whose native mapping has exactly the expected size, you can instruct Connex DDS to use the native type instead. That is, if `sizeof(long double) == 16`, you can tell Connex DDS to map DDS_LongDouble to long double by defining the following macro either in code or on the compile line:

```
-DRTI_CDR_SIZEOF_LONG_DOUBLE=16
```

Note: 3: Unions in IDL are mapped to structs in C, C++ and records in ADA, so that Connex DDS will not have to dynamically allocate memory for unions containing variable-length fields such as strings or sequences. To be efficient, the entire struct (or class in C++/CLI) is not sent when the union is published. Instead, Connex DDS uses the discriminator field of the struct to decide what field in the struct is actually sent on the wire.

Note: 4: So-called "anonymous sequences" —sequences of sequences in which the sequence element has no type name of its own—are not supported. Such sequences are deprecated in CORBA and may be removed from future versions of IDL. For example, this is *not* supported:

```
sequence<sequence<short,4>,4> MySequence;
```

Sequences of typedef'ed types, where the typedef is really a sequence, are supported. For example, this is supported:

```
typedef sequence<short,4> MyShortSequence;
sequence<MyShortSequence,4> MySequence;
```

- Note: 5:** IDL wchar and char are mapped to Java char, 16-bit unsigned quantities representing Unicode characters as specified in the standard OMG IDL to Java mapping. In C++/CLI, char and wchar are mapped to System::Char.
- Note: 6:** The unsigned version for integer types is mapped to its signed version as specified in the standard OMG IDL to Java mapping.
- Note: 7:** There is no current support in Java for the IDL long double type. This type is mapped to double as specified in the standard OMG IDL to Java mapping.
- Note: 8:** Java does not have a typedef construct, nor does C++/CLI. Typedefs for types that are neither arrays nor sequences (struct, unions, strings, wstrings, primitive types and enums) are "unwound" to their original type until a simple IDL type or user-defined IDL type (of the non-typedef variety) is encountered. For typedefs of sequences or arrays, *RTI Code Generator* will generate wrapper classes if **-corba** is not used; no wrapper classes are generated if **-corba** is used.
- Note: 9:** In C, C++ and ADA, all the members in a value type, structure or union that are declared with the pointer symbol ("*") will be mapped to references (pointers). In C++/CLI and Java, the pointer symbol is ignored because the members are always mapped as references.
- Note: 10:** In-line nested types are not supported inside structures, unions or valuetypes. For example, this is *not* supported:

```

struct Outer {
    short outer_short;
    struct Inner {
        char inner_char;
        short inner_short;
    } outer_nested_inner;
};

```

Note: 11: The sequence <Type>Seq is implicitly declared in the IDL file and therefore it cannot be declared explicitly by the user. For example, this is not supported:

```
typedef sequence<Foo> FooSeq; //error
```

Note: 12: *RTI Code Generator* will supply a default bound for sequences and strings. You can specify that bound with the **-sequenceSize** or **-stringSize** command-line option, respectively. See the *RTI Code Generator User's Manual*.

Note: 13: In ADA, primitive types are not represented as native language types (e.g. , Character, etc.) but as custom types in the DDS namespace (Standard.DDS.Long, Standard.DDS.Char, etc.). These typedefs are used to ensure that a field's size is the same across platforms.

Note: 14: Every type provides a default constructor, a copy constructor, a move constructor (C++11), a constructor with parameters to set all the type's members, a destructor, a copy-assignment operator, and a move-assignment operator (C++11). Types also include equality operators, the operator << and a namespace-level swap function.

```

PrimitiveStruct();
explicit PrimitiveStruct(char char_member);
PrimitiveStruct(PrimitiveStruct&& other_) OMG_NOEXCEPT;
PrimitiveStruct& operator=(PrimitiveStruct&& other_) OMG_NOEXCEPT;

bool operator == (const PrimitiveStruct& other_) const;
bool operator != (const PrimitiveStruct& other_) const;
void swap(PrimitiveStruct& other_) OMG_NOEXCEPT ;
std::ostream& operator << (std::ostream& o,const PrimitiveStruct&
sample);

```

Note: 15: Sequences of pointers are not supported. For example, this is NOT supported:

```
sequence<long*, 100>;
```

Sequences of typedefed types, where the typedef is really a pointer, are supported. For example, this is supported:

```
typedef long* pointerToLong;
sequence<pointerToLong, 100>;
```

3.3.5 Escaped Identifiers

To use an IDL keyword as an identifier, the keyword must be “escaped” by prepending an underscore, ‘_’. In addition, you must run *RTI Code Generator* with the **-enableEscapeChar** option. For example:

```
struct MyStruct {
    octet _octet; // octet is a keyword. To use the type
                // as a member name we add '_'
};
```

The use of ‘_’ is a purely lexical convention that turns off keyword checking. The generated code will not contain ‘_’. For example, the mapping to C would be as follows:

```
struct MyStruct {
    unsigned char octet;
};
```

Note: If you generate code from an IDL file to a language ‘X’ (for example, C++), the keywords of this language cannot be used as IDL identifiers, even if they are escaped. For example:

```
struct MyStruct {
    long int; // error
    long _int; // error
};
```

3.3.6 Namespaces In IDL Files

In IDL, the **module** keyword is used to create namespaces for the declaration of types defined within the file.

Here is an example IDL definition:

```
module PackageName {
    struct Foo {
        long field;
    };
};
```

C Mapping:

The name of the module is concatenated to the name of the structure to create the namespace. The resulting code looks like this:

```
typedef struct PackageName_Foo {
    DDS_Long field;
} PackageName_Foo;
```

C++ Mapping:

In the Traditional C++ API, when using the **-namespace** command-line option, *RTI Code Generator* generates a namespace, such as the following:

```
namespace PackageName{
    class Foo {
    public:
        DDS_Long field;
    }
}
```

Without the **-namespace** option, the mapping adds the module to the name of the class:

```
class PackageName_Foo {
    public:
        DDS_Long field;
}
```

In the Modern C++ API, namespaces are always used.

C++/CLI Mapping:

Independently of the usage of the **-namespace** command-line option, *RTI Code Generator* generates a namespace, such as the following:

```
namespace PackageName{
    public ref struct Foo: public DDS::ICopyable<Foo^> {
    public:
        System::Int32 field;
    };
}
```

Java Mapping:

A **Foo.java** file will be created in a directory called **PackageName** to use the equivalent concept as defined by Java. The file **PackageName/Foo.java** will contain a declaration of Foo class:

```
package PackageName;
    public class Foo {
        public int field;
    };
```

In a more complex example, consider the following IDL definition:

```
module PackageName {
    struct Bar {
        long field;
    };
    struct Foo {
        Bar barField;
    };
};
```

When *RTI Code Generator* generates code for the above definition, it will resolve the **Bar** type to be within the scope of the **PackageName** module and automatically generate fully qualified type names.

C Mapping:

```
typedef struct PackageName_Bar {
    DDS_Long field;
} PackageName_Bar;
typedef struct PackageName_Foo {
    PackageName_Bar barField;
} PackageName_Foo;
```

C++ Mapping:

With **-namespace**:

```
namespace PackageName {
    class Bar {
    public:
        DDS_Long field;
    };
    class Foo {
    public:
        PackageName::Bar barField;
    };
};
```

Without **-namespace**:

```
class PackageName_Bar {
    public:
        DDS_Long field;
};
class PackageName_Foo {
    public:
        PackageName_Bar barField;
};
```

C++/CLI Mapping:

```
namespace PackageName{
    public ref struct Bar: public DDS::ICopyable<Bar^> {
        public:
            System::Int32 field;
    };
    public ref struct Foo: public DDS::ICopyable<Foo^> {
        public:
            PackageName::Bar^ barField;
    };
};
```

Java Mapping:

PackageName/Bar.java and **PackageName/Foo.java** would be created with the following code, respectively:

```
package PackageName;
public class Bar {
    public
        int field;
};

package PackageName;
public class Foo {
    public
        PackageName.Bar barField = PackageName.Bar.create();
};
```

3.3.7 Referring to Other IDL Files

IDL files may refer to other IDL files using a syntax borrowed from C, C+, and C+/CLI preprocessors:

```
#include "Bar.idl"
```

If *RTI Code Generator* encounters such a statement in an IDL file **Foo.idl** and runs with the preprocessor enabled (default), it will look in **Bar.idl** to resolve the types referenced in **Foo.idl**. For example:

```

Bar.idl
    struct Bar {
    };

Foo.idl
    struct Foo {
        Bar m1;
    };

```

The parsing of Foo in the previous scenario will be successful as Bar can be found in **Bar.idl**. If Bar was not declared in **Bar.idl**, **RTI Code Generator** will report an error indicating that the symbol could not be found.

If the preprocessor is not enabled when running *RTI Code Generator* (see command-line option **-ppDisable**), the parsing of the previous IDL file will fail because RTI Code Generator will not be able to find a reference to Bar within **Bar.idl**.

To prevent RTI Code Generator from resolving a type, use the **//@resolve-name** directive (see [The @resolve-name Directive \(Section 3.3.9.3 on page 119\)](#)).

3.3.8 Preprocessor Directives

RTI Code Generator supports the standard preprocessor directives defined by the IDL specification, such as **#if**, **#endif**, **#include**, and **#define**.

To support these directives, *RTI Code Generator* calls an external C preprocessor before parsing the IDL file. On Windows systems, the preprocessor is **cl.exe**. On other architectures, the preprocessor is **cpp**. You can change the default preprocessor with the **-ppPath** option. If you do not want to run the preprocessor, use the **-ppDisable** option (see the *RTI Code Generator User's Manual*).

3.3.9 Using Custom Directives

The following *RTI Code Generator*-specific directives can be used in your IDL file:

```

//@key (see The @key Directive \(Section 3.3.9.1 on the next page\))
//@copy (see The @copy and Related Directives \(Section 3.3.9.2 on page 117\))
//@copy-c
//@copy-cppcli
//@copy-java
//@copy-java-begin
//@copy-declaration
//@copy-c-declaration
//@copy-cppcli-declaration
//@copy-java-declaration
//@copy-java-declaration-begin
//@resolve-name [true | false] (see The @resolve-name Directive \(Section 3.3.9.3 on page 119\))
//@top-level [true | false] (see The @top-level Directive \(Section 3.3.9.4 on page 120\))

```


Notes:

- To apply multiple directives to the same member or structure in an IDL file, put each additional directive on a new line, as shown below:

```
struct A {
    long a; //@key
           //@ID 20
    long b;
}; //@Extensibility FINAL_EXTENSIBILITY
   //@top-level false
```

- Custom directives start with “//@”. Do not put a space between the slashes and the @, or the directive will not be recognized by *RTI Code Generator*.

The directives are case-sensitive. For instance, you must use //@key (not //@Key).

3.3.9.1 The @key Directive

To declare a key for your data type, insert the @key directive in the IDL file after one or more fields of the data type.

With each key, Connex DDS associates an internal 16-byte representation, called a *key-hash*.

If the maximum size of the serialized key is greater than 16 bytes, to generate the key-hash, Connex DDS computes the MD5 key-hash of the serialized key in network-byte order. Otherwise (if the maximum size of the serialized key is <= 16 bytes), the key-hash is the serialized key in network-byte order.

Only **struct** definitions in IDL may have key fields. When *RTI Code Generator* encounters //@key, it considers the previously declared field in the enclosing structure to be part of the key. [Table 3.11 Example Keys](#) shows some examples of keys.

Table 3.11 Example Keys

Type	Key Fields
<pre>struct NoKey { long member1; long member2; }</pre>	
<pre>struct SimpleKey { long member1; //@key long member2; }</pre>	member1

Table 3.11 Example Keys

Type	Key Fields
<pre>struct NestedNoKey { SimpleKey member1; long member2; }</pre>	
<pre>struct NestedKey { SimpleKey member1; //@key long member2; }</pre>	member1.member1
<pre>struct NestedKey2 { NoKey member1; //@key long member2; }</pre>	member1.member1 member1.member2
<pre>valuetype BaseValueKey { public long member1; //@key }</pre>	member1
<pre>valuetype DerivedValueKey :BaseValueKey { public long member2; //@key }</pre>	member1 member2
<pre>valuetype DerivedValue : BaseValueKey { public long member2; }</pre>	member1
<pre>struct ArrayKey { long member1[3]; //@key }</pre>	member1[0] member1[1] member1[2]

3.3.9.2 The @copy and Related Directives

To copy a line of text verbatim into the generated code files, use the **@copy** directive in the IDL file. This feature is particularly useful when you want your generated code to contain text that is valid in the target programming language but is not valid IDL. It is often used to add user comments or headers or pre-processor commands into the generated code.

```
//@copy // Modification History
//@copy // -----
//@copy // 17Jul05aaa, Created.
//@copy
//@copy // #include "MyTypes.h"
```

These variations allow you to use the same IDL file for multiple languages:

@copy-c	Copies code if the language is C or C++
@copy-cppcli	Copies code if the language is C++/CLI
@copy-java	Copies code if the language is Java.
@copy-ada	Copies code if the language is Ada.

For example, to add import statements to generated Java code:

```
//@copy-java import java.util.*;
```

The above line would be ignored if the same IDL file was used to generate non-Java code.

In C, C++, and C++/CLI, the lines are copied into all of the **foo*.h, c, cxx, cpp** files generated from **foo.idl**. For Java, the lines are copied into all of the ***.java** files that were generated from the original “.idl” file. The lines will not be copied into any additional files that are generated using the **-example** command line option.

@copy-java-begin copies a line of text at the beginning of all the Java files generated for a type. The directive only applies to the first type that is immediately below in the IDL file. A similar directive for Ada files is also available, **@copy-ada-begin**.

If you want *RTI Code Generator* to copy lines only into the files that declare the data types—**foo.h** for C, C++, and C++/CLI, **foo.java** for Java—use the **//@copy*declaration** forms of this directive.

Note that the first whitespace character to follow **//@copy** is considered a delimiter and will not be copied into generated files. All subsequent text found on the line, including any leading whitespaces will be copied.

//@copy-declaration	Copies the text into the file where the type is declared (<type>.h for C and C++, or <type>.java for Java)
//@copy-c-declaration	Same as //@copy-declaration, but for C and C++ code
//@copy-cppcli-declaration	Same as //@copy-declaration, but for C++/CLI code
//@copy-java-declaration	Same as //@copy-declaration, but for Java-only code
//@copy-ada-declaration	Same as //@copy-declaration, but for Ada-only code
//@copy-java-declaration-begin	Same as //@copy-java-declaration, but only copies the text into the file where the type is declared
//@copy-ada-declaration-begin	Same as //@copy-java-declaration-begin, but only for Ada-only code

3.3.9.3 The @resolve-name Directive

By default, the RTI Code Generator tries to resolve all the references to types and constants in an IDL file. For example:

```
module PackageName {
    struct Foo {
        Bar barField;
    };
};
```

The compilation of the previous IDL file will report an error like the following:

```
ERROR com.rti.ndds.nddsgen.Main Foo.idl line x:x member type 'Bar' not found
```

In most cases, this is the expected behavior. However, in some cases, you may want to skip the resolution step. For example, assume that the Bar type is defined in a separate IDL file and that you are running *RTI Code Generator* without an external preprocessor by using the command-line option **-ppDisable** (maybe because the preprocessor is not available in their host platform, see [Preprocessor Directives \(Section 3.3.8 on page 115\)](#)):

Bar.idl

```
module PackageName {
    struct Bar {
        long field;
    };
};
```

Foo.idl

```
#include "Bar.idl"
module PackageName {
    struct Foo {
        Bar barField;
    };
};
```

In this case, compiling **Foo.idl** would generate the 'not found' error. However, Bar is defined in Bar.idl. To specify that *RTI Code Generator* should not resolve a type reference, use the **//@resolve-name false** directive. For example:

```
#include "Bar.idl"
module PackageName {
    struct Foo {
```

```

        Bar barField; //@resolve-name false
    };
};

```

When this directive is used, then for the field preceding the directive, RTI Code Generator will assume that the type is a unkeyed 'structure' and it will use the type name unmodified in the generated code.

Java mapping:

```

package PackageName;
public class Foo {
    public Bar barField = Bar.create();
};

```

C++ mapping:

```

namespace PackageName {
class Foo {
    public:
        Bar barField;
};
};

```

It is up to you to include the correct header files (or if using Java, to import the correct packages) so that the compiler resolves the 'Bar' type correctly. If needed, this can be done using the copy directives (see [The @copy and Related Directives \(Section 3.3.9.2 on page 117\)](#)).

When used at the end of the declaration of a structure in IDL, then the directive applies to all types within the structure, including the base type if defined. For example:

```

struct MyStructure: MyBaseStructure
{
    Foo member1;
    Bar member2;
};    //@resolve-name false

```

3.3.9.4 The @top-level Directive

By default, *RTI Code Generator* generates user-level type-specific methods for all structures/unions found in an IDL file. These methods include the methods used by *DataWriters* and *DataReaders* to send and receive data of a given type. General methods for writing and reading that take a void pointer are not offered by Connex DDS because they are not type safe. Instead, type-specific methods must be created to support a particular data type.

We use the term 'top-level type' to refer to the data type for which you intend to create a DCPS *Topic* that can be published or subscribed to. For top-level types, *RTI Code Generator* must create all of the type-

specific methods previously described in addition to the code to serialize/deserialize those types. However, some of structures/unions defined in the IDL file are only embedded within higher-level structures and are not meant to be published or subscribed to individually. For non-top-level types, the *DataWriters* and *DataReaders* methods to send or receive data of those types are superfluous and do not need to be created. Although the existence of these methods is not a problem in and of itself, code space can be saved if these methods are not generated in the first place.

You can mark non-top-level types in an IDL file with the directive `'//@top-level false'` to tell *RTI Code Generator* not to generate type-specific methods. Code will still be generated to serialize and deserialize those types, since they may be embedded in top-level types.

In this example, *RTI Code Generator* will generate *DataWriter/DataReader* code for `TopLevelStruct` only:

```
struct EmbeddedStruct{
    short member;
}; //@top-level false
struct TopLevelStruct{
    EmbeddedStruct member;
};
```

3.4 Creating User Data Types with Extensible Markup Language (XML)

You can describe user data types with Extensible Markup Language (XML) notation. Connex DDS provides DTD and XSD files that describe the XML format; see `<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti_dds_topic_types.dtd` and `<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti_dds_topic_types.xsd`, respectively (in *5.x.y*, the *x* and *y* stand for the version numbers of the current release). (`<NDDSHOME>` is described in [Paths Mentioned in Documentation \(Section on page xxxviii\)](#).)

The XML validation performed by *RTI Code Generator* always uses the DTD definition. If the `<!DOCTYPE>` tag is not in the XML file, *RTI Code Generator* will look for the default DTD document in `<NDDSHOME>/resource/schema`. Otherwise, it will use the location specified in `<!DOCTYPE>`.

We recommend including a reference to the XSD/DTD files in the XML documents. This provides helpful features in code editors such as Visual Studio® and Eclipse™, including validation and auto-completion while you are editing the XML. We recommend including the reference to the XSD document in the XML files because it provides stricter validation and better auto-completion than the DTD document.

To include a reference to the XSD document in your XML file, use the attribute `xsi:noNamespaceSchemaLocation` in the `<types>` tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
"<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti_dds_topic_types.xsd">
    ...
</types>
```

To include a reference to the DTD document in your XML file, use the `<!DOCTYPE>` tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE types SYSTEM
"<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti_dds_topic_types.dtd">
  <types>
    ...
  </types>
```

Table 3.12 Mapping Type System Constructs to XML shows how to map the type system constructs into XML.

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
char	char	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="char_member" type="char"/> </struct></pre>
wchar	wchar	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="wchar_member" type="wchar"/> </struct></pre>
octet	octet	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="octet_member" type="octet"/> </struct></pre>
short	short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="short_member" type="short"/> </struct></pre>
unsigned short	unsignedShort	<pre>struct PrimitiveStruct { unsigned short unsigned_short_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="unsigned_short_ member" type="unsignedShort"/> </struct></pre>
long	long	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="long_ member" type="long"/> </struct></pre>
unsigned long	unsignedLong	<pre>struct PrimitiveStruct { unsigned long unsigned_long_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name=" unsigned_long_ member" type="unsignedLong"/> </struct></pre>

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
long long	longLong	<pre>struct PrimitiveStruct { long long long_long_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="long_long_member" type="longLong"/> </struct></pre>
unsigned long long	unsignedLongLong	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_ member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="unsigned_long_long_ member" type="unsignedLongLong"/> </struct></pre>
float	float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="float_member" type="float"/> </struct></pre>
double	double	<pre>struct PrimitiveStruct { double double_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="double_member" type="double"/> </struct></pre>
long double	longDouble	<pre>struct PrimitiveStruct { long double long_double_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="long_double_member" type="longDouble"/> </struct></pre>
boolean	boolean	<pre>struct PrimitiveStruct { boolean boolean_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="boolean_member" type="boolean"/> </struct></pre>
unbounded string	string without stringMaxLength attribute or with stringMaxLength set to -1	<pre>struct PrimitiveStruct { string string_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="string_member" type="string"/> </struct> or <struct name="PrimitiveStruct"> <member name="string_member" type="string" stringMaxLength="-1"/> </struct></pre>
bounded string	string with stringMaxLength attribute	<pre>struct PrimitiveStruct { string<20> string_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="string_member" type="string" stringMaxLength="20"/> </struct></pre>

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
unbounded wstring	wstring without stringMaxLength attribute or with stringMaxLength set to -1	<pre>struct PrimitiveStruct { wstring wstring_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="wstring_member" type="wstring"/> </struct> or <struct name="PrimitiveStruct"> <member name="wstring_member" type="wstring" stringMaxLength="-1"/> </struct></pre>
bounded wstring	wstring with stringMaxLength attribute	<pre>struct PrimitiveStruct { wstring<20> wstring_ member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="wstring_member" type="wstring" stringMaxLength="20"/> </struct></pre>
pointer	pointer attribute with values true,false,0 or 1 Default (if not present): 0	<pre>struct PrimitiveStruct { long * long_member; };</pre>	<pre><struct name="PointerStruct"> <member name="long_member" type="long" pointer="true"/> </struct></pre>
bitfield ¹	bitfield attribute with the bitfield length	<pre>struct BitfieldStruct { short short_member: 1; unsigned short unsignedShort_member: 1; short short_nmember_2: 0; long long_member : 5; };</pre>	<pre><struct name="BitFieldStruct"> <member name="short_member" type="short" bitField="1"/> <member name="unsignedShort_member" type="unsignedShort" bitField="1"/> <member type="short" bitField="0"/> <member name="long_member" type="long" bitField="5"/> </struct></pre>
key directive ²	key attribute with values true, false, 0 or 1 Default (if not present): 0	<pre>struct KeyedPrimitiveStruct { short short_member; //@key };</pre>	<pre><struct name="KeyedPrimitiveStruct"> <member name="short_member" type="short" key="true"/> </struct></pre>

¹Data types containing bitfield members are not supported by DynamicData ([Interacting Dynamically with User Data Types \(Section 3.8 on page 141\)](#)).

²Directives are RTI extensions to the standard IDL grammar. For additional information about directives see [Using Custom Directives \(Section 3.3.9 on page 115\)](#).

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
resolve-name directive ¹	resolveName attribute with values true, false, 0 or 1 Default (if not present): 1	<pre>struct UnresolvedPrimitiveStruct { PrimitiveStruct primitive_member; //@resolve-name false };</pre>	<pre><struct name= "UnresolvedPrimitiveStruct"> <member name="primitive_member" type="PrimitiveStruct" resolveName="false"/> </struct></pre>
top-level directive ²	topLevel attribute with values true, false, 0 or 1 Default (if not present): 1	<pre>struct TopLevelPrimitiveStruct { short short_member; }; //@top-level false</pre>	<pre><struct name="TopLevelPrimitiveStruct" topLevel="false"> <member name="short_member" type="short"/> </struct></pre>
Other directives ³	directive tag	<pre>//@copy This text will be copied in the generated files</pre>	<pre><directive kind="copy"> This text will be copied in the generated files </directive></pre>
enum	enum tag	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 };</pre>	<pre><enum name="PrimitiveEnum"> <enumerator name="ENUM1"/> <enumerator name="ENUM2"/> <enumerator name="ENUM3"/> </enum></pre>
		<pre>enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre><enum name="PrimitiveEnum"> <enumerator name="ENUM1" value="10"/> <enumerator name="ENUM2" value="20"/> <enumerator name="ENUM3" value="30"/> </enum></pre>
constant	const tag	<pre>const double PI = 3.1415;</pre>	<pre><const name="PI" type="double" value="3.1415"/></pre>

¹Directives are RTI extensions to the standard IDL grammar. For additional information about directives see [Using Custom Directives \(Section 3.3.9 on page 115\)](#).

²Directives are RTI extensions to the standard IDL grammar. For additional information about directives see [Using Custom Directives \(Section 3.3.9 on page 115\)](#).

³Directives are RTI extensions to the standard IDL grammar. For additional information about directives see [Using Custom Directives \(Section 3.3.9 on page 115\)](#).

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
struct	struct tag	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre><struct name="PrimitiveStruct"> <member name="short_member" type="short"/> </struct></pre>
union	union tag	<pre>union PrimitiveUnion switch (long) { case 1: short short_member; case 2: float float_member; default: long long_member; };</pre>	<pre><union name="PrimitiveUnion"> <discriminator type="long"/> <case> <caseDiscriminator value="1"/> <member name="short_member" type="short"/> </case> <case> <caseDiscriminator value="2"/> <caseDiscriminator value="3"/> <member name="float_member" type="float"/> </case> <case> <caseDiscriminator value="default"/> <member name="long_member" type="long"/> </case> </union></pre>
valuetype	valuetype tag	<pre>valuetype BaseValueType { public long long_member; }; valuetype DerivedValueType: BaseValueType { public long long_member_2; };</pre>	<pre><valuetype name="BaseValueType"> <member name="long_member" type="long" visibility="public"/> </valuetype> <valuetype name="DerivedValueType" baseClass="BaseValueType"> <member name="long_member_2" type="long" visibility="public"/> </valuetype></pre>
typedef	typedef tag	<pre>typedef short ShortType;</pre>	<pre><typedef name="ShortType" type="short"/></pre>
		<pre>struct PrimitiveStruct { short short_member; }; typedef PrimitiveStruct PrimitiveStructType;</pre>	<pre><struct name="PrimitiveStruct"> <member name="short_member" type="short"/> </struct> <typedef name="PrimitiveStructType" type="nonBasic" nonBasicTypeName="PrimitiveStruct"/></pre>

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
arrays	Attribute arrayDimensions	<pre>struct OneArrayStruct { short short_array[2]; };</pre>	<pre><struct name="OneArrayStruct"> <member name="short_array" type="short" arrayDimensions="2"/> </struct></pre>
		<pre>struct TwoArrayStruct { short short_array[1][2]; };</pre>	<pre><struct name="TwoArrayStruct"> <member name="short_array" type="short" arrayDimensions="1,2"/> </struct></pre>
bounded sequence	Attribute sequenceMaxLength > 0	<pre>struct SequenceStruct { sequence<short,4> short_sequence; };</pre>	<pre><struct name="SequenceStruct"> <member name="short_sequence" type="short" sequenceMaxLength="4"/> </struct></pre>
unbounded sequence	Attribute sequenceMaxLength set to -1	<pre>struct SequenceStruct { sequence<short> short_sequence; };</pre>	<pre><struct name="SequenceStruct"> <member name="short_sequence" type="short" sequenceMaxLength="-1"/> </struct></pre>
array of sequences	Attributes sequenceMaxLength and arrayDimensions	<pre>struct ArrayOfSequencesStruct { sequence<short,4> short_sequence_array[2]; };</pre>	<pre><struct name= "ArrayOfSequenceStruct"> <member name= "short_sequence_array" type="short" arrayDimensions="2" sequenceMaxLength="4"/> </struct></pre>
sequence of arrays	Must be implemented with a typedef tag	<pre>typedef short ShortArray[2]; struct SequenceOfArraysStruct { sequence<ShortArray,2> short_array_sequence; };</pre>	<pre><typedef name="ShortArray" type="short" dimensions="2"/> <struct name= "SequenceOfArrayStruct"> <member name= "short_array_ sequence" type="nonBasic" nonBasicTypeName="ShortSequence" sequenceMaxLength="2"/> </struct></pre>

Table 3.12 Mapping Type System Constructs to XML

Type/Construct		Example	
IDL	XML	IDL	XML
sequence of sequences	Must be implemented with a typedef tag	<pre>typedef sequence<short,4> ShortSequence; struct SequenceOfSequencesStruct { sequence<ShortSequence,2> short_sequence_sequence; };</pre>	<pre><typedef name="ShortSequence" type="short"sequenceMaxLength="4"/> <struct name="SequenceofSequencesStruct"> <member name="short_sequence_ sequence" type="nonBasic" nonBasicTypeName="ShortSequence" sequenceMax-Length="2"/> </struct></pre>
module	module tag	<pre>module PackageName { struct PrimitiveStruct { long long_member; }; };</pre>	<pre><module name="PackageName"> <struct name="PrimitiveStruct"> <member name="long_member" type="long"/> </struct> </module></pre>
include	include tag	<pre>#include "PrimitiveTypes.idl"</pre>	<pre><include file="PrimitiveTypes.xml"/></pre>

3.4.1 Primitive Types

The primitive types char, wchar, long double, and wstring are not supported natively in XSD. Connexx DDS provides definitions for these types in the file `<NDDSHOME>/resource/app/app_support/rtiddsgen/schema`. All files that use the primitive types char, wchar, long double and wstring must reference `rti_dds_topic_types_common.xsd`. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:dds="http://www.omg.org/dds">
  <xsd:import namespace="http://www.omg.org/dds"
  schemaLocation="rti_dds_topic_types_common.xsd"/>
  <xsd:complexType name="Foo">
    <xsd:sequence>
      <xsd:element name="myChar" minOccurs="1"
maxOccurs="1" type="dds:char"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

3.5 Creating User Data Types with XML Schemas (XSD)

You can describe data types with XML schemas (XSD). The format is based on the standard IDL-to-WSDL mapping described in the OMG document "CORBA to WSDL/SOAP Interworking

Specification."

Example Header for XSD:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:dds="http://www.omg.org/dds"
  xmlns:tns="http://www.omg.org/IDL-Mapped/"
  targetNamespace="http://www.omg.org/IDL-Mapped/">
<xsd:import namespace="http://www.omg.org/dds"
  schemaLocation="rti_dds_topic_types_common.xsd"/>
...
</xsd:schema>
```

Mapping Type System Constructs to XSD (Section Table 3.13 below) describes how to map IDL types to XSD. The Connex DDS code generator, *rtiddsgen*, will only accept XSD files that follow this mapping.

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
char	dds:char ^a	<pre>struct PrimitiveStruct { char char_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="char_member" minOccurs="1" maxOccurs="1" type="dds:char"> </xsd:sequence> </xsd:complexType></pre>
wchar	dds:wchar ^b	<pre>struct PrimitiveStruct { wchar wchar_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="wchar_member" minOccurs="1" maxOccurs="1" type="dds:wchar"> </xsd:sequence> </xsd:complexType></pre>
octet	xsd: unsignedByte	<pre>struct PrimitiveStruct { octet octet_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="octet_member" minOccurs="1" maxOccurs="1" type="xsd:unsignedByte"> </xsd:sequence> </xsd:complexType></pre>

^a All files that use the primitive types char, wchar, long double and wstring must reference `rti_dds_topic_types_common.xsd`. See [Primitive Types \(Section 3.4.1 on the previous page\)](#).

^b All files that use the primitive types char, wchar, long double and wstring must reference `rti_dds_topic_types_common.xsd`. See [Primitive Types \(Section 3.4.1 on the previous page\)](#).

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
short	xsd:short	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="short_member" minOccurs="1" maxOccurs="1" type="xsd:short"/> </xsd:sequence> </xsd:complexType></pre>
unsigned short	xsd:unsignedShort	<pre>struct PrimitiveStruct { unsigned short unsigned_ short_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="unsigned_short_member" minOccurs="1" maxOccurs="1" type="xsd:unsignedShort"/> </xsd:sequence> </xsd:complexType></pre>
long	xsd:int	<pre>struct PrimitiveStruct { long long_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="long_member" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType></pre>
unsigned long	xsd:unsignedInt	<pre>struct PrimitiveStruct { unsigned long unsigned_ long_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="unsigned_long_member" minOccurs="1" maxOccurs="1" type="xsd:unsignedInt"/> </xsd:sequence> </xsd:complexType></pre>
long long	xsd:long	<pre>struct PrimitiveStruct { long long long_long_ member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:elementname="long_long_member" minOccurs="1" maxOccurs="1" type="xsd:long"/> </xsd:sequence> </xsd:complexType></pre>
unsigned long long	xsd:unsignedLong	<pre>struct PrimitiveStruct { unsigned long long unsigned_long_long_ member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="unsigned_long_long_member" minOccurs="1" maxOccurs="1" type="xsd:unsignedLong"/> </xsd:sequence> </xsd:complexType></pre>

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
float	xsd:float	<pre>struct PrimitiveStruct { float float_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="float_member" minOccurs="1" maxOccurs="1" type="xsd:float"/> </xsd:sequence> </xsd:complexType></pre>
double	xsd:double	<pre>struct PrimitiveStruct { double double_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="double_member" minOccurs="1" maxOccurs="1" type="xsd:double"/> </xsd:sequence> </xsd:complexType></pre>
long double	dds:longDouble	<pre>struct PrimitiveStruct { long double long_double_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="long_double_member" minOccurs="1" maxOccurs="1" type="dds:longDouble"/> </xsd:sequence> </xsd:complexType></pre>
boolean	xsd:boolean	<pre>struct PrimitiveStruct { boolean boolean_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="boolean_member" minOccurs="1" maxOccurs="1" type="xsd:boolean"/> </xsd:sequence> </xsd:complexType></pre>
unbounded string	xsd:string	<pre>struct PrimitiveStruct { string string_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="string_member" minOccurs="1" maxOccurs="1" type="xsd:string"/> </xsd:sequence> </xsd:complexType></pre>
bounded string	xsd:string with restriction to specify the maximum length	<pre>struct PrimitiveStruct { string<20> string_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct_string_member_BoundedString"> <xsd:sequence> <xsd:element name="item" minOccurs="1" maxOccurs="1"> <xsd:simpleType> <xsd:restriction base="xsd:string"> <xsd:maxLength value="20" fixed="true"/> </xsd:restriction> </xsd:simpleType> </xsd:element> </xsd:sequence> </xsd:complexType> <xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="string_member" minOccurs="1" maxOccurs="1" type="tns:PrimitiveStruct_string_member_BoundedString"/> </xsd:sequence> </xsd:complexType></pre>

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
unbounded wstring	dds:wstring ^a	<pre>struct PrimitiveStruct { wstring wstring_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="wstring_member" minOccurs="1" maxOccurs="1" type="dds:wstring"/> </xsd:sequence> </xsd:complexType></pre>
bounded wstring	xsd:wstring with restriction to specify the maximum length	<pre>struct PrimitiveStruct { wstring<20> wstring_ member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct_wstring_member_ BoundedString"> <xsd:sequence> <xsd:element name="item" minOccurs="1" maxOccurs="1"> <xsd:simpleType> <xsd:restriction base="dds:wstring"> <xsd:maxLength value="20" fixed="true"/> </xsd:restriction> </xsd:simpleType> </xsd:element> </xsd:sequence> </xsd:complexType> <xsd:complexType name= "PrimitiveStruct"> <xsd:sequence> <xsd:element name="wstring_member" minOccurs="1" maxOccurs="1" type= "tns:PrimitiveStruct_wstring_member_BoundedString"/> </xsd:sequence> </xsd:complexType></pre>
pointer	<pre><!-- @pointer <true false 1 0> --></pre> Default (if not specified): false	<pre>struct PrimitiveStruct { long * long_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="long_member" minOccurs="1" maxOccurs="1" type="xsd:int"/> <!-- @pointer true --> </xsd:sequence> </xsd:complexType></pre>
key directive ^b	<pre><!-- @key <true false 1 0> --></pre> Default (if not specified): false	<pre>struct KeyedPrimitiveStruct { long long_member; //@key };</pre>	<pre><xsd:complexType name="KeyedPrimitiveStruct"> <xsd:sequence> <xsd:element name="long_member" minOccurs="1" maxOccurs="1" type="xsd:int"/> <!-- @key true --> </xsd:sequence> </xsd:complexType></pre>

^aAll files that use the primitive types char, wchar, long double and wstring must reference rti_dds_topic_types_common.xsd. See [Primitive Types \(Section 3.4.1 on page 128\)](#)

^bDirectives are RTI extensions to the standard IDL grammar. For additional information about directives, see [Using Custom Directives](#).

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
resolvename directive ^a	<pre><!-- @resolveName <true false 1 0> --> Default (if not specified): true</pre>	<pre>struct UnresolvedPrimitiveStruct { PrimitiveStruct primitive_member; //@resolve-name false };</pre>	<pre><xsd:complexType name="UnresolvedPrimitiveStruct"> <xsd:sequence> <xsd:element name="primitive_member" minOccurs="1" maxOccurs="1" type="PrimitiveStruct"/> <!-- @resolveName false --> </xsd:sequence> </xsd:complexType></pre>
top-level directive ^b	<pre><!-- @topLevel <true false 1 0> --> Default (if not specified): true</pre>	<pre>struct TopLevelPrimitiveStruct { short short_member; }; //@top-level false</pre>	<pre><xsd:complexType name="TopLevelPrimitiveStruct"> <xsd:sequence> <xsd:element name="short_member" minOccurs="1" maxOccurs="1" type="xsd:short"/> </xsd:sequence> </xsd:complexType> <!-- @topLevel false --></pre>
other directives	<pre><!-- @<directive kind> <value> --></pre>	<pre>//@copy This text will be copied in the generated files</pre>	<pre><!--@copy This text will be copied in the generated files --></pre>
enum	<pre>xsd:simpleType with enumeration</pre>	<pre>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 }; enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</pre>	<pre><xsd:simpleType name="PrimitiveEnum"> <xsd:restriction base="xsd:string"> <xsd:enumeration value="ENUM1"/> <xsd:enumeration value="ENUM2"/> <xsd:enumeration value="ENUM3"/> </xsd:restriction> </xsd:simpleType> <xsd:simpleType name="PrimitiveEnum"> <xsd:restriction base="xsd:string"> <xsd:enumeration value="ENUM1"> <xsd:annotation> <xsd:appinfo> <ordinal>10</ordinal> </xsd:appinfo> </xsd:annotation> </xsd:enumeration> <xsd:enumeration value="ENUM2"> <xsd:annotation> <xsd:appinfo> <ordinal>20</ordinal> </xsd:appinfo> </xsd:annotation> </xsd:enumeration> <xsd:enumeration value="ENUM3"> <xsd:annotation> <xsd:appinfo> <ordinal>30</ordinal> </xsd:appinfo> </xsd:annotation> </xsd:enumeration> </xsd:restriction> </xsd:simpleType></pre>
constant	IDL constants are mapped by substituting their value directly in the generated file		

^a Directives are RTI extensions to the standard IDL grammar. For additional information about directives, see [Using Custom Directives](#).

^b Directives are RTI extensions to the standard IDL grammar. For additional information about directives, see [Using Custom Directives](#).

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
struct	xsd:complexType with xsd:sequence	<pre>struct PrimitiveStruct { short short_member; };</pre>	<pre><xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="short_member" minOccurs="1" maxOccurs="1" type="xsd:short"/> </xsd:sequence> </xsd:complexType></pre>
union	xsd:complexType with xsd:choice	<pre>union PrimitiveUnion switch (long) { case 1: short short_member; default: long long_member; };</pre>	<pre><xsd:complexType name="PrimitiveUnion"> <xsd:sequence> <xsd:element name="discriminator" type="xsd:int"/> <xsd:choice> <!-- case 1 -->^a <xsd:element name="short_ member" minOccurs="0" maxOccurs="1" type="xsd:short"> <xsd:annotation> <xsd:appinfo> <case>1</case> </xsd:appinfo> </xsd:annotation> </xsd:element> <!-- case default --> <xsd:element name="long_member" minOccurs="0" maxOccurs="1" type="xsd:int"> <xsd:annotation> <xsd:appinfo> <case>default</case> </xsd:appinfo> </xsd:annotation> </xsd:element> </xsd:choice> </xsd:sequence> </xsd:complexType></pre>
valuetype	xsd:complexType with @valuetype directive	<pre>valuetype BaseValueType { public long long_member; }; valuetype DerivedValueType: BaseValueType { public long long_member2; public long long_member3; };</pre>	<pre><xsd:complexType name="BaseValueType"> <xsd:sequence> <xsd:element name="long_member" maxOccurs="1" minOccurs="1" type="xs:int"/> <!-- @visibility public --> </xsd:sequence> </xs:complexType> <!-- @valuetype true - -> <xs:complexType name="DerivedValueType"> <xs:complexContent> <xs:extension base="BaseValueType"> <xs:sequence> <xs:element name="long_member2" maxOccurs="1" minOccurs="1" type="xs:int"/> <!-- @visibility public --> <xs:element name="long_member3" maxOccurs="1" minOccurs="1" type="xs:int"/> <!-- @visibility public --> </xs:sequence> </xs:extension> </xs:complexContent> </xs:complexType> <!-- @valuetype true --></pre>

^aThe discriminant values can be described using comments (as specified by the standard) or `xsd:annotation` tags. We recommend using annotations because comments may be removed by XSD/XML parsers.

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
typedef	Type definitions are mapped to XML schema type restrictions	<pre>typedef short ShortType; struct PrimitiveStruct { short short_member; }; typedef PrimitiveType = PrimitiveStructType;</pre>	<pre><xsd:simpleType name="ShortType"> <xsd:restriction base="xsd:short"/> </xsd:simpleType> <!-- Struct definition --> <xsd:complexType name="PrimitiveStruct"> <xsd:sequence> <xsd:element name="short_member" minOccurs="1" maxOccurs="1" type="xsd:short"/> </xsd:sequence> </xsd:complexType> <!-- Typedef definition --> <xsd:complexType name="PrimitiveTypeStructType"> <xsd:complexContent> <xsd:restriction base="PrimitiveStruct"> <xsd:sequence> <xsd:element name="short_member" minOccurs="1" maxOccurs="1" type="xsd:short"/> </xsd:sequence> </xsd:restriction> </xsd:complexContent> </xsd:complexType></pre>
arrays	<p>n xsd:complexType with sequence containing one element with min & max occurs</p> <p>There is one xsd:complexType per array dimension</p>	<pre>struct OneArrayStruct { short short_array[2]; };</pre>	<pre><!-- Array type --> <xsd:complexType name="OneArrayStruct_short_array_ArrayOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="2" maxOccurs="2" type="xsd:short"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Struct w unidimensional array member --> <xsd:complexType name="OneArrayStruct"> <xsd:sequence> <xsd:element name="short_array" minOccurs="1" maxOccurs="1" type= "OneArrayStruct_short_array_ArrayOfShort"/> </xsd:sequence> </xsd:complexType></pre>

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
arrays (cont'd)	<p>n xsd:complexType with sequence containing one element with min & max occurs</p> <p>There is one xsd:complexType per array dimension</p>	<pre>struct TwoArrayStruct { short short_array[2][1]; };</pre>	<pre><!--Second dimension array type --> <xsd:complexType name= "TwoArrayStruct_short_array_ArrayOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="2" maxOccurs="2" type="xsd:short"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- First dimension array type --> <xsd:complexType name= "TwoArrayStruct_ short_array_ArrayOfArrayOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="1" maxOccurs="1" type= "TwoArrayStruct_short_array_ArrayOfShort"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Struct containing a bidimensional array member --> <xsd:complexType name="TwoArrayStruct"> <xsd:sequence> <xsd:element name="short_array" minOccurs="1" maxOccurs="1" type= "TwoArrayStruct_short_ array_ArrayOfArrayOfShort"/> </xsd:sequence> </xsd:complexType></pre>
bounded sequence	<p>xsd:complexType with sequence containing one element with min & max occurs</p>	<pre>struct SequenceStruct { sequence<short,4> short_sequence; };</pre>	<pre><!-- Sequence type --> <xsd:complexType name= "SequenceStruct_short_sequence_ SequenceOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="0" maxOccurs="4" type="xsd:short"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Struct containing a bounded sequence member --> <xsd:complexType name="SequenceStruct"> <xsd:sequence> <xsd:element name="short_sequence" minOccurs="1" maxOccurs="1" type= "SequenceStruct_short_ sequence_SequenceOfShort"/> </xsd:sequence> </xsd:complexType></pre>
unbounded sequence	<p>xsd:complexType with sequence containing one element with min & max occurs</p>	<pre>struct SequenceStruct { sequence<short> short_ sequence; };</pre>	<pre><!-- Sequence type --> <xsd:complexType name= "SequenceStruct_short_sequence_SequenceOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="0" maxOccurs="unbounded" type="xsd:short"/> </xsd:sequence> </xsd:complexType> <!-- Struct containing unbounded sequence member --> <xsd:complexType name="SequenceStruct"> <xsd:sequence> <xsd:element name="short_sequence" minOccurs="1" maxOccurs="1" type= "SequenceStruct_short_sequence_SequenceOfShort"/> </xsd:sequence> </xsd:complexType></pre>

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
array of sequences	<p>$n + 1$ <code>xsd:complexType</code> with sequence containing one element with min & max occurrences.</p> <p>There is one <code>xsd:complexType</code> per array dimension and one <code>xsd:complexType</code> for the sequence.</p>	<pre>struct ArrayOfSequencesStruct { sequence<short,4> sequence_sequence[2]; }</pre>	<pre><!-- Sequence declaration --> <xsd:complexType name= "ArrayOfSequencesStruct_sequence_array_SequenceOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="0" maxOccurs="4" type="xsd:short"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Array declaration --> <xsd:complexType name= "ArrayOfSequencesStruct_sequence_array_ArrayOf SequenceOfShort"> <xsd:sequence> <xsd:element name="item" minOccurs="2" maxOccurs="2" type= "ArrayOfSequencesStruct_sequence_array_SequenceOfShort"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Structure containing a member that is an array of sequences --> <xsd:complexType name="ArrayOfSequencesStruct"> <xsd:sequence> <xsd:element name="sequence_array" minOccurs="1" maxOccurs="1" type= "ArrayOfSequencesStruct_sequence_ array_ArrayOf SequenceOfShort"/> </xsd:sequence> </xsd:complexType></pre>
sequence of arrays	<p>Sequences of arrays must be implemented using an explicit type definition (typedef) for the array</p>	<pre>typedef short ShortArray [2]; struct SequenceOfArraysStruct { sequence<ShortArray,2> arrays_sequence; }</pre>	<pre><!-- Array declaration --> <xsd:complexType name="ShortArray"> <xsd:sequence> <xsd:element name="item" minOccurs="2" maxOccurs="2" type="xsd:short"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Sequence declaration --> <xsd:complexType name= "SequencesOfArraysStruct_array_ sequence_SequenceOfShortArray"> <xsd:sequence> <xsd:element name="item" minOccurs="0" maxOccurs="2" type="ShortArray"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Struct containing a sequence of arrays --> <xsd:complexType name="SequenceOfArraysStruct"> <xsd:sequence> <xsd:element name="arrays_sequence" minOccurs="1" maxOccurs="1" type= "SequencesOfArraysStruct_arrays_sequence_ SequenceOfShortArray"/> </xsd:sequence> </xsd:complexType></pre>

Table 3.13 Mapping Type System Constructs to XSD

Type/Construct		Example	
IDL	XSD	IDL	XSD
sequence of sequences	Sequences of sequences must be implemented using an explicit type definition (typedef) for the second sequence	<pre>typedef sequence<short,4> ShortSequence; struct SequenceOfSequences { sequence<ShortSequence, 2> sequences_sequence; };</pre>	<pre><!-- Internal sequence declaration --> <xsd:complexType name="ShortSequence"> <xsd:sequence> <xsd:element name="item" minOccurs="0" maxOccurs="4" type="xsd:short"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- External sequence declaration -- > <xsd:complexType name= "SequencesOfSequences_sequences_sequence_ SequenceOfShortSequence"> <xsd:sequence> <xsd:element name="item" minOccurs="0" maxOccurs="2" type="ShortSequence"> </xsd:element> </xsd:sequence> </xsd:complexType> <!-- Struct containing a sequence of sequences --> <xsd:complexType name="SequenceOfSequences"> <xsd:sequence> <xsd:element name="sequences_sequence" minOccurs="1" maxOccurs="1" type="SequencesOfSequences_ sequences_sequence_SequenceOfShortSequence"/> </xsd:sequence> </xsd:complexType></pre>
module	Modules are mapped adding the name of the module before the name of each type inside the module	<pre>module PackageName { struct PrimitiveStruct { long long_member; }; };</pre>	<pre><xsd:complexType name= "PackageName.PrimitiveStruct"> <xsd:sequence> <xsd:element name="long_member" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType></pre>
include	xsd:include	<pre>#include "PrimitiveType.idl"</pre>	<pre><xsd:include schemaLocation= "PrimitiveType.xsd"/></pre>

3.6 Using RTI Code Generator (rtiddsgen)

RTI Code Generator creates the code needed to define and register a user-data type with Connexx DDS. Using this tool is optional if:

- You are using dynamic types (see [Managing Memory for Built-in Types \(Section 3.2.7 on page 62\)](#))
- You are using one of the built-in types (see [Built-in Data Types \(Section 3.2 on page 30\)](#))

See the *RTI Code Generator User's Manual* for more information.

3.7 Using Generated Types without Connex DDS (Standalone)

You can use the generated type-specific source and header files without linking the Connex DDS libraries or even including the Connex DDS header files. That is, the files generated by *RTI Code Generator* for your data types can be used standalone.

The directory `<NDDSHOME>/resource/app/app_support/rtiddsgen/standalone` contains the required helper files:

- include: header and templates files for C and C++.
- src: source files for C and C++.
- class: Java jar file.

Note: You must use *RTI Code Generator*'s **-notypecode** option to generate code for standalone use. See the *RTI Code Generator User's Manual* for more information.

3.7.1 Using Standalone Types in C

The generated files that can be used standalone are:

- `<idl file name>.c`: Types source file
- `<idl file name>.h`: Types header file

The type plug-in code (`<idl file>Plugin.[c,h]`) and type-support code (`<idl file>Support.[c,h]`) cannot be used standalone.

To use the generated types in a standalone manner:

1. Make sure you use *rtiddsgen*'s **-notypecode** option to generate the code.
2. Include the directory `<NDDSHOME>/resource/app/app_support/rtiddsgen/standalone/include` in the list of directories to be searched for header files.
3. Add the source files, `ndds_standalone_type.c` and `<idl file name>.c`, to your project.
4. Include the file `<idl file name>.h` in the source files that will use the generated types in a standalone manner.
5. Compile the project using the following two preprocessor definitions:
 - `NDDS_STANDALONE_TYPE`
 - The definition for your platform (`RTI_VXWORKS`, `RTI_QNX`, `RTI_WIN32`, `RTI_INTY`, `RTI_LYNX` or `RTI_UNIX`)

3.7.2 Using Standalone Types in C++

(This section applies to the Traditional C++ API only)

The generated files that can be used standalone are:

- **<idl file name>.cxx**: Types source file
- **<idl file name>.h**: Types header file

The type-plugin code (**<idl file>Plugin.[cxx,h]**) and type-support code (**<idl file>Support.[cxx,h]**) cannot be used standalone.

To use the generated types in a standalone manner:

1. Make sure you use *RTI Code Generator's* **-notypecode** option to generate the code.
2. Include the directory **<NDDSHOME>/resource/app/app_support/rtiddsgen/standalone/include** in the list of directories to be searched for header files.
3. Add the source files, **ndds_standalone_type.cxx** and **<idl file name>.cxx**, to your project.
4. Include the file **<idl file name>.h** in the source files that will use the *RTI Code Generator* types in a standalone manner.
5. Compile the project using the following two preprocessor definitions:
 - **NDDS_STANDALONE_TYPE**
 - The definition for your platform (such as **RTI_VXWORKS**, **RTI_QNX**, **RTI_WIN32**, **RTI_INTY**, **RTI_LYNX** or **RTI_UNIX**)

3.7.3 Standalone Types in Java

The generated files that can be used standalone are:

- **<idl type>.java**
- **<idl type>Seq.java**

The type code (**<idl file>TypeCode.java**), type-support code (**<idl type>TypeSupport.java**), *DataReader* code (**<idl file>DataReader.java**) and *DataWriter* code (**<idl file>DataWriter.java**) cannot be used standalone.

To use the generated types in a standalone manner:

1. Make sure you use *RTI Code Generator's* **-notypecode** option to generate the code.
2. Include the file **ndds_standalone_type.jar** in the classpath of your project.

3. Compile the project using the standalone types files (*<idl type>.java* and *<idl type>Seq.java*).

3.8 Interacting Dynamically with User Data Types

3.8.1 Type Schemas and TypeCode Objects

Type schemas—the names and definitions of a type and its fields—are represented by TypeCode objects, described in [Introduction to TypeCode \(Section 3.1.3 on page 29\)](#).

3.8.2 Defining New Types

This section does not apply when using the separate add-on product, *Ada Language Support*, which does not support Dynamic Types.

Locally, your application can access the type code for a generated type "Foo" by calling the **FooTypeSupport::get_typecode()** (Traditional C++ Notation) operation in the code for the type generated by *RTI Code Generator* (unless type-code support is disabled with the **-notypecode** option). But you can also create TypeCodes at run time without any code generation.

Creating a TypeCode is parallel to the way you would define the type statically: you define the type itself with some name, then you add members to it, each with its own name and type.

For example, consider the following statically defined type. It might be in C, C++, or IDL; the syntax is largely the same.

```
struct MyType {
    long my_integer;
    float my_float;
    bool my_bool;
    string<128> my_string; // @key
};
```

This is how you would define the same type at run time in the Traditional C++ API:

```
DDS_ExceptionCode_t ex = DDS_NO_EXCEPTION_CODE;
DDS_StructMemberSeq structMembers; // ignore for now
DDS_TypeCodeFactory* factory =
    DDS_TypeCodeFactory::get_instance();
DDS_TypeCode* structTc = factory->create_struct_tc(
    "MyType", structMembers, ex);
// If structTc is NULL, check 'ex' for more information.
structTc->add_member(
    "my_integer", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_LONG)
    DDS_TYPECODE_NONKEY_REQUIRED_MEMBER, ex);
structTc->add_member(
    "my_float", DDS_TYPECODE_MEMBER_ID_INVALID,
```

```

        factory->get_primitive_tc(DDS_TK_FLOAT),
        DDS_TYPECODE_NONKEY_REQUIRED_MEMBER, ex);
structTc->add_member(
    "my_bool", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_BOOLEAN),
    DDS_TYPECODE_NONKEY_REQUIRED_MEMBER, ex);
structTc->add_member(
    "my_string", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->create_string_tc(128),
    DDS_TYPECODE_KEY_MEMBER, ex);

```

More detailed documentation for the methods and constants you see above, including example code, can be found in the API Reference HTML documentation, which is available for all supported programming languages.

If, as in the example above, you know all of the fields that will exist in the type at the time of its construction, you can use the **StructMemberSeq** to simplify the code:

```

DDS_StructMemberSeq structMembers;
structMembers.ensure_length(4, 4);
DDS_TypeCodeFactory* factory = DDS_TypeCodeFactory::get_instance();
structMembers[0].name = DDS_String_dup("my_integer");
structMembers[0].type = factory->get_primitive_tc(DDS_TK_LONG);
structMembers[1].name = DDS_String_dup("my_float");
structMembers[1].type = factory->get_primitive_tc(DDS_TK_FLOAT);
structMembers[2].name = DDS_String_dup("my_bool");
structMembers[2].type = factory->get_primitive_tc(DDS_TK_BOOLEAN);
structMembers[3].name = DDS_String_dup("my_string");
structMembers[3].type = factory->create_string_tc(128);
structMembers[3].is_key = DDS_BOOLEAN_TRUE;
DDS_ExceptionCode_t ex = DDS_NO_EXCEPTION_CODE;
DDS_TypeCode* structTc =
    factory->create_struct_tc(
        "MyType", structMembers, ex);

```

After you have defined the TypeCode, you will register it with a *DomainParticipant* using a logical name (note: this step is not required in the Modern C++ API). You will use this logical name later when you create a *Topic*.

```

DDSDynamicDataTypesupport* type_support =
    new DDSDynamicDataTypesupport(structTc,
        DDS_DYNAMIC_DATA_TYPE_PROPERTY_DEFAULT);
DDS_ReturnCode_t retcode =
    type_support->register_type(participant,
        "My Logical Type Name");

```

For code examples for the Modern C++ API, please refer to the API Reference HTML documentation: Modules, Programming How-To's, DynamicType and DynamicData Use Cases.

Now that you have created a type, you will need to know how to interact with objects of that type. See [Sending Only a Few Fields \(Section 3.8.3 below\)](#) for more information.

3.8.3 Sending Only a Few Fields

In some cases, your data model may contain a large number of potential fields, but it may not be desirable or appropriate to include a value for every one of them with every DDS data sample.

- **It may use too much bandwidth.** You may have a very large data structure, parts of which are updated very frequently. Rather than resending the entire data structure with every change, you may wish to send only those fields that have changed and rely on the recipients to reassemble the complete state themselves.
- **It may not make sense.** Some fields may only have meaning in the presence of other fields. For example, you may have an event stream in which certain fields are only relevant for certain kinds of events.

To support these and similar cases, Connex DDS supports mutable types and optional members (see the [RTI Connex DDS Core Libraries Getting Started Guide Addendum for Extensible Types](#)).

3.8.4 Sending Type Codes on the Network

In addition to being used locally, serialized type codes are typically published automatically during discovery as part of the built-in topics for publications and subscriptions. See [Built-in DataReaders \(Section 16.2 on page 773\)](#). This allows applications to publish or subscribe to topics of arbitrary types. This functionality is useful for generic system monitoring tools like the *rtiddsspy* debug tool. For details on using *rtiddsspy*, see the API Reference HTML documentation (select **Modules, Programming Tools**).

Note: Type codes are not cached by Connex DDS upon receipt and are therefore not available from the built-in data returned by the *DataWriter's* `get_matched_subscription_data()` operation or the *DataReader's* `get_matched_publication_data()` operation.

If your data type has an especially complex type code, you may need to increase the value of the `type_code_max_serialized_length` field in the *DomainParticipant's* `DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)` (Section 8.5.4 on page 593). Or, to prevent the propagation of type codes altogether, you can set this value to zero (0). Be aware that some features of monitoring tools, as well as some features of the middleware itself (such as `ContentFilteredTopics`) will not work correctly if you disable `TypeCode` propagation.

3.8.4.1 Type Codes for Built-in Types

The type codes associated with the built-in types are generated from the following IDL type definitions:

```

module DDS {
    /* String */
    struct String {
        string<max_size> value;
    };
    /* KeyedString */
    struct KeyedString {
        string<max_size> key; //@key
        string<max_size> value;
    };
    /* Octets */
    struct Octets {
        sequence<octet, max_size> value;
    };
    /* KeyedOctets */
    struct KeyedOctets {
        string<max_size> key; //@key
        sequence<octet, max_size> value;
    };
};

```

The maximum size (**max_size**) of the strings and sequences that will be included in the type code definitions can be configured on a per-*DomainParticipant*-basis by using the properties in [Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant](#).

Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant

Built-in Type	Property	Description
String	dds.builtin_type.string.max_size	Maximum size of the strings published by the <i>DataWriters</i> and received by the <i>DataReaders</i> belonging to a <i>DomainParticipant</i> (includes the NULL-terminated character). Default: 1024
KeyedString	dds.builtin_type.keyed_string.max_key_size	Maximum size of the keys used by the <i>DataWriters</i> and <i>DataReaders</i> belonging to a <i>DomainParticipant</i> (includes the NULL-terminated character). Default: 1024
	dds.builtin_type.keyed_string.max_size	Maximum size of the strings published by the <i>DataWriters</i> and received by the <i>DataReaders</i> belonging to a <i>DomainParticipant</i> using the built-in type (includes the NULL-terminated character). Default: 1024
Octets	dds.builtin_type.octets.max_size	Maximum size of the octet sequences published by the <i>DataWriters</i> and <i>DataReaders</i> belonging to a <i>DomainParticipant</i> . Default: 2048

Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant

Built-in Type	Property	Description
Keyed-Octets	dds.builtin_type.keyed_octets.max_key_size	Maximum size of the key published by the <i>DataWriter</i> and received by the <i>DataReaders</i> belonging to the <i>DomainParticipant</i> (includes the NULL-terminated character). Default: 1024.
	dds.builtin_type.keyed_octets.max_size	Maximum size of the octet sequences published by the <i>DataWriters</i> and <i>DataReaders</i> belonging to a <i>DomainParticipant</i> . Default: 2048

3.9 Working with DDS Data Samples

You should now understand how to define and work with data types, whether you're using the simple data types built into the middleware (see [Built-in Data Types \(Section 3.2 on page 30\)](#)), dynamically defined types (see [Managing Memory for Built-in Types \(Section 3.2.7 on page 62\)](#)), or code generated from IDL or XML files (see [Creating User Data Types with IDL \(Section 3.3 on page 69\)](#) and [Creating User Data Types with Extensible Markup Language \(XML\) \(Section 3.4 on page 121\)](#)).

Now that you have chosen one or more data types to work with, this section will help you understand how to create and manipulate objects of those types.

3.9.1 Objects of Concrete Types

If you use one of the built-in types or decide to generate custom types from an IDL or XML file, your Connext DDS data type is like any other data type in your application: a class or structure with fields, methods, and other members that you interact with directly.

In C and Traditional C++:

You create and delete your own objects from factories, just as you create Connext DDS objects from factories. In the case of user data types, the factory is a singleton object called the type support. Objects allocated from these factories are deeply allocated and fully initialized.

```

/* In the generated header file: */
struct MyData {
    char* myString;
};
/* In your code: */
MyData* sample = MyDataTypeSupport_create_data();
char* str = sample->myString; /*empty, non-NULL string*/
/* ... */
MyDataTypeSupport_delete_data(sample);

```

In Traditional C++:

You create and delete objects using the TypeSupport factories.

```
MyData* sample = MyDataTypeSupport::create_data(); char* str = sample-
>myString; // empty, non-NULL string // ... MyDataTypeSupport::delete_data
(sample);
```

In Modern C++:

Generated types have value-type semantics and provide a default constructor, a constructor with parameters to initialize all the members, a copy constructor and assignment operator, a move constructor and move-assignment operator (C++11 only), a destructor, equality operators, a swap function and an overloaded operator<<. Data members are accessed using getters and setters.

```
// In the generated header file
class MyData {
public:
    MyData();
    explicit MyData(const dds::core::string& myString);

    // Note: the implicit destructor, copy and
    // move constructors, and assignment operators
    // are available
    dds::core::string& myString() OMG_NOEXCEPT;
    const dds::core::string& myString() const OMG_NOEXCEPT;
    void myString(const dds::core::string& value);

    bool operator == (const MyData& other_) const;
    bool operator != (const MyData& other_) const;
private:
    // ...
};
void swap(MyData& a, MyData& b) OMG_NOEXCEPT
    std::ostream& operator <<
    (std::ostream& o, const MyData& sample);
// In your code:
MyData sample("Hello");
sample.myString("Bye");
```

In C# and C++/CLI:

You can use a no-argument constructor to allocate objects. Those objects will be deallocated by the garbage collector as appropriate.

```
// In the generated code (C++/CLI):
public ref struct MyData {
public: System::String^ myString;
```

```
};
// In your code, if you are using C#:
MyData sample = new MyData();
System.String str = sample.myString;
// empty, non-null string
// In your code, if you are using C++/CLI:
MyData^ sample = gcnew MyData();
System::String^ str = sample->myString;
// empty, non-nullptr string
```

In Java:

You can use a no-argument constructor to allocate objects. Those objects will be deallocated by the garbage collector as appropriate.

```
// In the generated code:
public class MyData {
    public String myString = "";
}
// In your code:
MyData sample = new MyData();
String str = sample->myString;
// empty, non-null string
```

3.9.2 Objects of Dynamically Defined Types

If you are working with a data type that was discovered or defined at run time, you will use the reflective API provided by the `DynamicData` class to get and set the fields of your object.

Consider the following type definition:

```
struct MyData {
    long myInteger;
};
```

As with a statically defined type, you will create objects from a `TypeSupport` factory. How to create or otherwise obtain a `TypeCode`, and how to subsequently create from it a `DynamicDataTypeSupport`, is described in [Defining New Types \(Section 3.8.2 on page 141\)](#). In the Modern C++ API you will use the `DynamicData` constructor, which receives a `DynamicType`.

For more information about the `DynamicData` and `DynamicDataTypeSupport` classes, consult the API Reference HTML documentation, which is available for all supported programming languages (select **Modules, RTI Connex DDS API Reference, Topic Module, Dynamic Data**).

In C:

```
DDS_DynamicDataTypeSupport* support = ...;
DDS_DynamicData* sample = DDS_DynamicDataTypeSupport_create_data(support);
DDS_Long theInteger = 0;
```



```

DDS_ReturnCode_t success = DDS_DynamicData_set_long(sample,
    "myInteger", DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 5);
/* Error handling omitted. */
success = DDS_DynamicData_get_long(    sample, &theInteger,
    "myInteger", DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
/* Error handling omitted. "theInteger" now contains the value 5
   if no error occurred.
*/

```

In Traditional C++:

```

DDSDynamicDataTypeSupport* support = ...;
DDS_DynamicData* sample = support->create_data();
DDS_ReturnCode_t success = sample->set_long("myInteger",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 5);
// Error handling omitted.
DDS_Long theInteger = 0;
success = sample->get_long(    &theInteger, "myInteger",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
// Error handling omitted.
// "theInteger" now contains the value 5 if no error occurred.

```

In Modern C++:

```

using namespace dds::core::xtypes;

StructType type(
    "MyData", {
        Member("myInteger", primitive_type<int32_t>())
    }
);
DynamicData sample(type);
sample.value("myInteger", 5);
int32_t the_int = sample.value<int32_t>("myInteger");
// "the_int" now contains the value 5 if no exception was thrown

```

In C++/CLI:

```

using DDS;
DynamicDataTypeSupport^ support = ...;
DynamicData^ sample = support->create_data();
sample->set_long("myInteger",
    DynamicData::MEMBER_ID_UNSPECIFIED, 5);
int theInteger = sample->get_long("myInteger",
    0 /*redundant w/ field name*/);
/* Exception handling omitted.
 * "theInteger" now contains the value 5 if no error occurred.
*/

```

In C#:

```

using namespace DDS;
DynamicDataTypeSupport support = ...;
DynamicData sample = support.create_data();

```

```

sample.set_long("myInteger", DynamicData.MEMBER_ID_UNSPECIFIED, 5);
int theInteger = sample.get_long("myInteger",
    DynamicData.MEMBER_ID_UNSPECIFIED);
/* Exception handling omitted.
 * "theInteger" now contains the value 5 if no error occurred.
 */

```

In Java:

```

import com.rti.dds.dynamicdata.*;
DynamicDataSupport support = ...;
DynamicData sample = (DynamicData) support.create_data();
sample.set_int("myInteger", DynamicData.MEMBER_ID_UNSPECIFIED, 5);
int theInteger = sample.get_int("myInteger",
    DynamicData.MEMBER_ID_UNSPECIFIED);
/* Exception handling omitted.
 * "theInteger" now contains the value 5 if no error occurred.
 */

```

The Modern C++ API provides convenience functions to convert among DynamicData samples and typed samples (such as MyData, from the previous example). For example:

```

#include "MyData.hpp"
// ...
MyData typed_sample(44);
DynamicData dynamic_sample = rti::core::xtypes::convert(typed_sample);
assert (dynamic_sample.value<int32_t>("myInteger") == 44);
dynamic_sample.value("myInteger", 33);
typed_sample = rti::core::xtypes::convert<Foo>(dynamic_sample);
assert (typed_sample.myInteger() == 33);

```

3.9.3 Serializing and Deserializing Data Samples

There are two TypePlugin operations to serialize a sample into a buffer and deserialize a sample from a buffer. The sample serialization/deserialization uses CDR representation.

The feature is supported in the following languages: C, Modern and Traditional C++, Java, and .NET.

C:

```

#include "FooSupport.h"
FooTypeSupport_serialize_data_to_cdr_buffer(...)
FooTypeSupport_deserialize_data_from_cdr_buffer(...)

```

Traditional C++

```

#include "FooSupport.h"
FooTypeSupport::serialize_data_to_cdr_buffer(...)
FooTypeSupport::deserialize_data_from_cdr_buffer(...)

```

Modern C++

```
#include "Foo.hpp"
dds::topic::topic_type_support<Foo>::to_cdr_buffer(...)
dds::topic::topic_type_support<Foo>::from_cdr_buffer(...)
```

Java:

```
FooTypeSupport.get_instance().serialize_to_cdr_buffer(...)
FooTypeSupport.get_instance().deserialize_from_cdr_buffer(...)
```

C++/CLI:

```
FooTypeSupport::serialize_data_to_cdr_buffer(...)
FooTypeSupport::deserialize_data_from_cdr_buffer(...)
```

C#:

```
FooTypeSupport.serialize_data_to_cdr_buffer(...)
FooTypeSupport.deserialize_data_from_cdr_buffer(...)
```

3.9.4 Accessing the Discriminator Value in a Union

A union type can only hold a single member. The **member_id** for this member is equal to the discriminator value. To get the value of the discriminator, use the operation **get_member_info_by_index()** on the `DynamicData` using an index value of 0. This operation fills in a `DynamicDataMemberInfo` structure, which includes a **member_id** field that is the value of the discriminator.

Once you know the discriminator value, you can use the proper version of **get_<type>()** (such as **get_long()**) to access the member value.

For example:

```
DynamicDataMemberInfo memberInfo = new DynamicDataMemberInfo();
myDynamicData.get_member_info_by_index(memberInfo, 0);
int discriminatorValue = memberInfo.member_id;
int myMemberValue = myDynamicData.get_long(null, discriminatorValue);
```

The Modern C++ API provides the method **discriminator_value()** to achieve the same result:

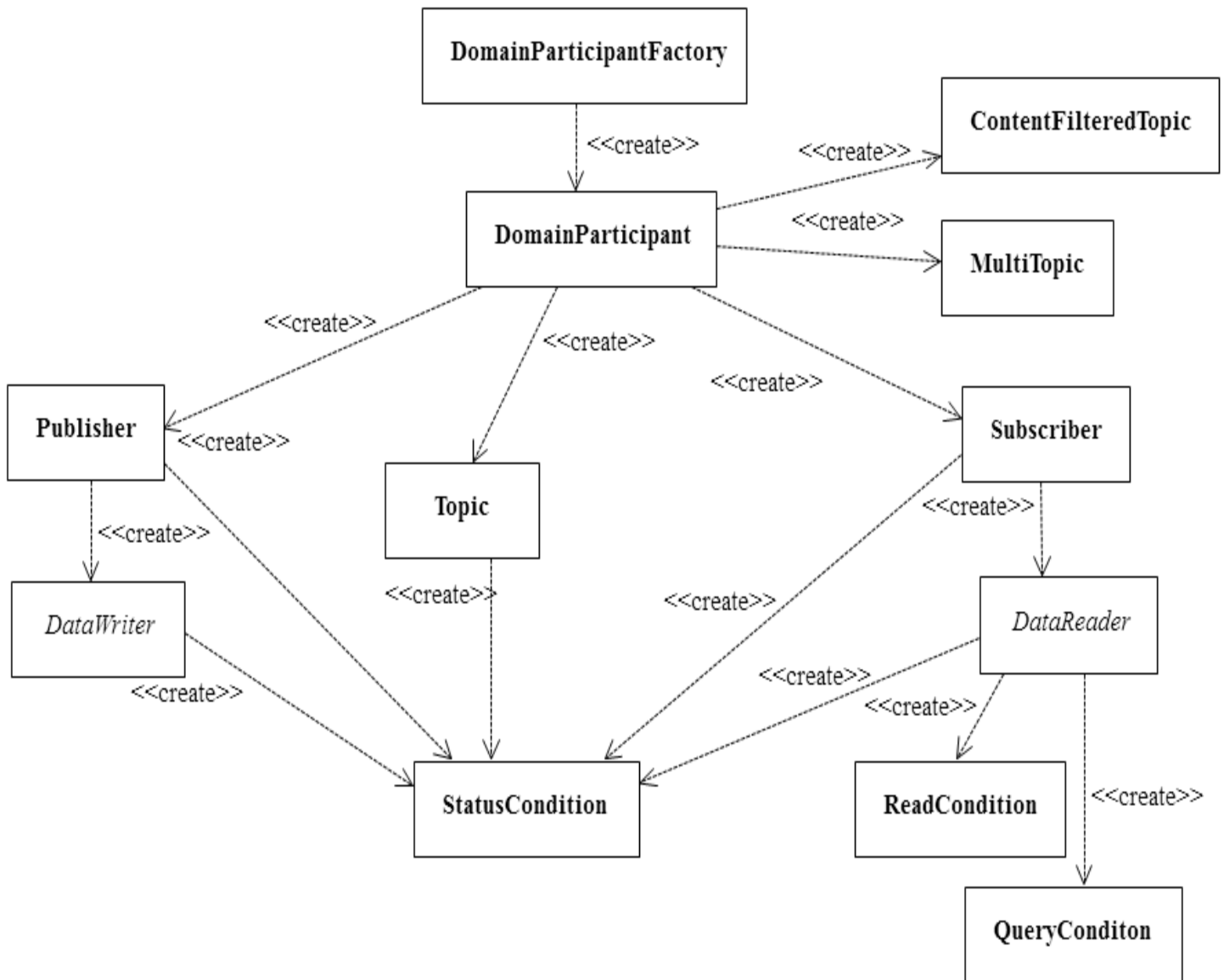
```
int32_t my_member_value = my_dynamic_data.value<int32_t>(
    my_dynamic_data.discriminator_value());
```

Chapter 4 DDS Entities

The main classes extend an abstract base class called a *DDS Entity*. Every *DDS Entity* has a set of associated events known as statuses and a set of associated Quality of Service Policies (QoS Policies). In addition, a *Listener* may be registered with the *Entity* to be called when status changes occur. *DDS Entities* may also have attached *DDS Conditions*, which provide a way to wait for status changes. [Figure 4.1 Overview of DDS Entities on the next page](#) presents an overview in a UML diagram.

This section describes the common operations and general designed patterns shared by all *DDS Entities* including *DomainParticipants*, *Topics*, *Publishers*, *DataWriters*, *Subscribers*, and *DataReaders*. In subsequent chapters, the specific statuses, *Listeners*, *Conditions*, and QoS Policies for each class will be discussed in detail.

Figure 4.1 Overview of DDS Entities



4.1 Common Operations for All DDS Entities

All DDS *Entities* (*DomainParticipants*, *Topics*, *Publishers*, *DataWriters*, *Subscribers*, and *DataReaders*) provide operations for:

4.1.1 Creating and Deleting DDS Entities

- C, Traditional C++, Java, and .NET:

The factory design pattern is used in creating and deleting DDS *Entities*. Instead of declaring and constructing or destructing *Entities* directly, a factory object is used to create an *Entity*. Almost all *Entity* factories are objects that are also *Entities*. The only exception is the factory for a *DomainParticipant*. See [Table 4.1 Entity Factories](#).

Table 4.1 Entity Factories

Entity	Created by
DomainParticipant	DomainParticipantFactory (a static singleton object provided by Connex DDS)
Topic	DomainParticipant
Publisher	
Subscriber	
DataWriter	
DataReader	
DataWriter	Publisher
DataReader	Subscriber

All *Entities* that are factories have:

- Operations to create and delete child *Entities*. For example:

DDSPublisher::create_datawriter()

DDSDomainParticipant::delete_topic()

- Operations to get and set the default QoS values used when creating child *Entities*. For example:

DDSSubscriber::get_default_datareader_qos()

DDSDomainParticipantFactory::set_default_participant_qos()

- And [ENTITYFACTORY QosPolicy \(Section 6.4.2 on page 315\)](#) to specify whether or not the newly created child *Entity* should be automatically enabled upon creation.

DataWriters may be created by a *DomainParticipant* or a *Publisher*. Similarly, *DataReaders* may be created by a *DomainParticipant* or a *Subscriber*.

An entity that is a factory cannot be deleted until *all* the child *Entities* created by it have been deleted.

Each *Entity* obtained through `create_<entity>()` must eventually be deleted by calling `delete_<entity>()`, or by calling `delete_contained_entities()`.

- Modern C++:

In the Modern C++ API the factory pattern is not explicit. Entities have constructors and destructors. The first argument to an Entity's constructor is its "factory" (except for the DomainParticipant). For example:

```
// Note: this example shows the simplest version of each Entity's constructor:
dds::domain::DomainParticipant participant(MY_DOMAIN_ID);
dds::topic::Topic<Foo> topic(participant, "Example Foo");
dds::sub::Subscriber subscriber(participant);
dds::sub::DataReader<Foo> reader(subscriber, topic);
dds::pub::Publisher publisher(participant);
dds::pub::DataWriter<Foo> writer(publisher, topic);
```

Entities are *reference types*. In a reference type copy operations, such as copy-construction and copy-assignment are shallow. The reference types are modeled after shared pointers. Similar to pointers, it is important to distinguish between an entity and a reference (or handle) to it. A single entity may have multiple references. Copying a reference does not copy the entity it is referring to—creating additional references from the existing reference(s) is a relatively inexpensive operation.

The lifecycle of references and the entity they are referring to is not the same. In general, the entity lives as long as there is at least one reference to it. When the last reference to the entity ceases to exist, the entity it is referring to is destroyed.

Applications can override the automatic destruction of Entities. An Entity can be explicitly closed (by calling the method `close()`) or retained (by calling `retain()`)

Closing an Entity destroys the underlying object and invalidates all references to it.

Retaining an Entity disables the automatic destruction when it loses all its reference. A retained Entity can be looked up (see [Looking Up DomainParticipants \(Section 8.2.4 on page 546\)](#)) and has to be explicitly destroyed with `close()`.

4.1.2 Enabling DDS Entities

The `enable()` operation changes an *Entity* from a non-operational to an operational state. *Entity* objects can be created disabled or enabled. This is controlled by the value of the `ENTITYFACTORY QosPolicy` ([Section 6.4.2 on page 315](#)) on the corresponding *factory* for the *Entity* (not on the *Entity* itself).

By default, all *Entities* are automatically created in the enabled state. This means that as soon as the *Entity* is created, it is ready to be used. In some cases, you may want to create the *Entity* in a 'disabled' state. For example, by default, as soon as you create a *DataReader*, the *DataReader* will start receiving new DDS

samples for its *Topic* if they are being sent. However, your application may still be initializing other components and may not be ready to process the data at that time. In that case, you can tell the *Subscriber* to create the *DataReader* in a disabled state. After all of the other parts of the application have been created and initialized, then the *DataReader* can be enabled to actually receive messages.

To create a particular entity in a disabled state, modify the EntityFactory QoS Policy of its corresponding *factory entity* before calling `create_<entity>()`. For example, to create a disabled *DataReader*, modify the *Subscriber's* QoS as follows:

```
DDS_SubscriberQos subscriber_qos;
subscriber->get_qos(subscriber_qos);
subscriber_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
subscriber->set_qos(subscriber_qos);
DDSDataReader* datareader =
    subscriber->create_datareader(topic, DDS_DATAREADER_QOS_DEFAULT, listener);
```

When the application is ready to process received data, it can enable the *DataReader*:

```
datareader->enable();
```

4.1.2.1 Rules for Calling enable()

In the following, a ‘Factory’ refers to a *DomainParticipant*, *Publisher*, or *Subscriber*; a ‘child’ refers to an entity created by the factory:

- If the factory is disabled, its children are always created disabled, regardless of the setting in the factory's EntityFactoryQoS.
- If the factory is enabled, its children will be created either enabled or disabled, according to the setting in the factory's EntityFactory QoS.
- Calling `enable()` on a child whose factory object is still disabled will fail and return `DDS_RECODE_RECONDITION_NOT_MET`.
- Calling `enable()` on a factory with EntityFactoryQoS set to `DDS_BOOLEAN_TRUE` will *recursively* enable all of the factory's children. If the factory's EntityFactoryQoS is set to `DDS_BOOLEAN_FALSE`, only the factory itself will be enabled.
- Calling `enable()` on an entity that is already enabled returns `DDS_RETCODE_OK` and has no effect.
- There is no complementary “**disable**” operation. You cannot disable an entity after it is enabled. Disabled *Entities* must have been created in that state.
- An entity's *Listener* will only be invoked if the entity is enabled.
- The existence of an entity is not propagated to other *DomainParticipants* until the entity is enabled (see [Discovery \(Section Chapter 14 on page 709\)](#)).

- If a *DataWriter/DataReader* is to be created in an enabled state, then the associated *Topic* must already be enabled. The enabled state of the *Topic* does not matter, if the *Publisher/Subscriber* has its *EntityFactory QosPolicy* to create children in a disabled state.
- When calling *enable()* for a *DataWriter/DataReader*, both the *Publisher/Subscriber* and the *Topic* must be enabled, or the operation will fail and return `DDS_RETCODE_PRECONDITION_NOT_MET`.

The following operations may be invoked on disabled *Entities*:

- **get_qos()** and **set_qos()** Some DDS-specified *QosPolicies* are *immutable*—they cannot be changed after an *Entity* is enabled. This means that for those policies, if the entity was created in the disabled state, **get/set_qos()** can be used to change the values of those policies until **enable()** is called on the *Entity*. After the *Entity* is enabled, changing the values of those policies will not affect the *Entity*. However, there are *mutable* *QosPolicies* whose values can be changed at anytime—even after the *Entity* has been enabled.

Finally, there are extended *QosPolicies* that are not a part of the DDS specification but offered by Connex DDS to control extended features for an *Entity*. Some of those extended *QosPolicies* cannot be changed after the *Entity* has been created—regardless of whether the *Entity* is enabled or disabled.

Into which exact categories a *QosPolicy* falls—mutable at any time, immutable after enable, immutable after creation—is described in the documentation for the specific policy.

- **get_status_changes()** and **get_*_status()** The status of an *Entity* can be retrieved at any time (but the status of a disabled *Entity* never changes). (Note: **get_*_status()** resets the related status so it no longer considered “changed.”)
- **get_statuscondition()** An *Entity’s StatusCondition* can be checked at any time (although the status of a disabled *Entity* never changes).
- **get_listener()** and **set_listener()** An *Entity’s Listener* can be changed at any time.
- **create_***() and **delete_***() A factory *Entity* can still be used to create or delete any child *Entity* that it can produce. Note: following the rules discussed previously, a disabled *Entity* will always create its children in a disabled state, no matter what the value of the *EntityFactory QosPolicy* is.
- **lookup_***() An *Entity* can always look up children it has previously created.

Most other operations are not allowed on disabled *Entities*. Executing one of those operations when an *Entity* is disabled will result in a return code of `DDS_RETCODE_NOT_ENABLED`. The documentation for a particular operation will explicitly state if it is not allowed to be used if the *Entity* is disabled.

The builtin transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter/DataReader* is created, or (c) you look up a builtin data reader, whichever happens first. Any changes to the builtin transport properties that are made after the builtin transports have been registered will have no affect on any *DataWriters/DataReaders*.

4.1.3 Getting an Entity's Instance Handle

The *Entity* class provides an operation to retrieve an instance handle for the object. The operation is simply:

```
InstanceHandle_t get_instance_handle()
```

An instance handle is a global ID for the entity that can be used in methods that allow user applications to determine if the entity was locally created, if an entity is owned (created) by another entity, etc.

4.1.4 Getting Status and Status Changes

The `get_status_changes()` operation retrieves the set of events, also known in DDS terminology as *communication statuses*, in the *Entity* that have changed since the last time `get_status_changes()` was called. This method actually returns a value that must be bitwise AND'ed with an enumerated bit mask to test whether or not a specific status has changed. The operation can be used in a polling mechanism to see if any statuses related to the *Entity* have changed. If an entity is disabled, all communication statuses are in the “unchanged” state so the list returned by the `get_status_changes()` operation will be empty.

A set of statuses is defined for each class of *Entities*. For each status, there is a corresponding operation, `get_<status-name>_status()`, that can be used to get its current value. For example, a *DataWriter* has a **DDS_OFFERED_DEADLINE_MISSED** status; it also has a `get_offered_deadline_missed_status()` operation:

```
DDS_StatusMask statuses;
DDS_OfferedDeadlineMissedStatus deadline_stat;
statuses = datawriter->get_status_changes();
if (statuses & DDS_OFFERED_DEADLINE_MISSED_STATUS) {
    datawriter->get_offered_deadline_missed_status(
        &deadline_stat);
    printf("Deadline missed %d times.\n",
        deadline_stat.total_count);
}
```

To reset a status (so that it is no longer considered “changed”), call `get_<status-name>_status()`. Or, in the case of the **DDS_DATA_AVAILABLE** status, call `read()`, `take()`, or one of their variants.

If you use a *StatusCondition* to be notified that a particular status has changed, the *StatusCondition*'s **trigger_value** will remain true unless you call `get_*_status()` to reset the status.

See also: [Statuses \(Section 4.3 on page 169\)](#) and [StatusConditions \(Section 4.6.8 on page 197\)](#).

4.1.5 Getting and Setting Listeners

Each type of *Entity* has an associated *Listener*, see [Listeners \(Section 4.4 on page 177\)](#). A *Listener* represents a set of functions that users may install to be called asynchronously when the state of *communication statuses* change.

The `get_listener()` operation returns the current *Listener* attached to the *Entity*.

The `set_listener()` operation installs a *Listener* on an *Entity*. The *Listener* will only be invoked on the changes of statuses specified by the accompanying mask. Only one listener can be attached to each *Entity*. If a *Listener* was already attached, `set_listener()` will replace it with the new one.

The `get_listener()` and `set_listener()` operations are directly provided by the *DomainParticipant*, *Topic*, *Publisher*, *DataWriter*, *Subscriber*, and *DataReader* classes so that listeners and masks used in the argument list are specific to each *Entity*.

Note: The `set_listener()` operation is not synchronized with the listener callbacks, so it is possible to set a new listener on an participant while the old listener is in a callback. Therefore you should be careful not to delete any listener that has been set on an enabled participant unless some application-specific means are available of ensuring that the old listener cannot still be in use.

See [Listeners \(Section 4.4 on page 177\)](#) for more information about *Listeners*.

4.1.6 Getting the StatusCondition

Each type of *Entity* may have an attached *StatusCondition*, which can be accessed through the `get_statuscondition()` operation. You can attach the *StatusCondition* to a *WaitSet*, to cause your application to wait for specific status changes that affect the *Entity*.

See [Conditions and WaitSets \(Section 4.6 on page 187\)](#) for more information about *StatusConditions* and *WaitSets*.

4.1.7 Getting, Setting, and Comparing QoS Policies

Each type of *Entity* has an associated set of QoS Policies (see [QoS Policies \(Section 4.2 on page 162\)](#)). QoS Policies allow you to configure and set properties for the *Entity*.

While most QoS Policies are defined by the DDS specification, some are offered by Connex DDS as extensions to control parameters specific to the implementation.

There are two ways to specify a QoS policy:

- Programmatically, as described in this section.
- QoS Policies can also be configured from XML resources (files, strings)—with this approach, you can change the QoS without recompiling the application. The QoS settings are automatically loaded

by the `DomainParticipantFactory` when the first `DomainParticipant` is created. See [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

The `get_qos()` operation retrieves the current values for the set of QoS Policies defined for the *Entity*.

QoS Policies can be set programmatically when an Entity is created, or modified with the Entity's `set_qos()` operation.

The `set_qos()` operation sets the QoS Policies of the entity. Note: not all QoS Policy changes will take effect instantaneously; there may be a delay since some QoS Policies set for one entity, for example, a *DataReader*, may actually affect the operation of a matched entity in another application, for example, a *DataWriter*.

The `get_qos()` and `set_qos()` operations are passed QoS structures that are specific to each derived entity class, since the set of QoS Policies that effect each class of *Entities* is different.

The `equals()` operation compares two Entity's QoS structures for equality. It takes two parameters for the two *Entities*' QoS structures to be compared, then returns TRUE if they are equal (all values are the same) or FALSE if they are not equal.

Each QoS Policy has default values (listed in the API Reference HTML documentation). If you want to use custom values, there are three ways to change QoS Policy settings:

- Before Entity creation (if custom values should be used for multiple *Entities*). See [Changing the QoS Defaults Used to Create DDS Entities: set_default_*_qos\(\)](#) (Section 4.1.7.1 on the next page).
- During Entity creation (if custom values are only needed for a particular Entity). See [Setting QoS During Entity Creation](#) (Section 4.1.7.2 on the next page).
- After Entity creation (if the values initially specified for a particular Entity are no longer appropriate). See [Changing the QoS for an Existing Entity](#) (Section 4.1.7.3 on page 161).

Regardless of when or how you make QoS changes, there are some rules to follow:

- Some QoS Policies interact with each other and thus must be set in a consistent manner. For instance, the maximum value of the HISTORY QoS Policy's *depth* parameter is limited by values set in the RESOURCE_LIMITS QoS Policy. If the values within a QoS Policy structure are inconsistent, then `set_qos()` will return the error `INCONSISTENT_POLICY`, and the operation will have no effect.
- Some policies can only be set when the *Entity* is created, or before the *Entity* is enabled. Others can be changed at any time. In general, all standard DDS QoS Policies can be changed before the *Entity* is enabled. A subset can be changed after the *Entity* is enabled. Connex DDS-specific QoS Policies either cannot be changed after creation or can be changed at any time. The changeability of each QoS Policy is documented in the API Reference HTML documentation as well as in [Table 4.2 QoS Policies](#). If you attempt to change a policy after it cannot be changed, `set_qos()` will fail with a return `IMMUTABLE_POLICY`.

4.1.7.1 Changing the QoS Defaults Used to Create DDS Entities: `set_default*_qos()`

Each parent factory has a set of default QoS settings that are used when the child entity is created. The *DomainParticipantFactory* has default QoS values for creating *DomainParticipants*. A *DomainParticipant* has a set of default QoS for each type of entity that can be created from the *DomainParticipant* (*Topic*, *Publisher*, *Subscriber*, *DataWriter*, and *DataReader*). Likewise, a *Publisher* has a set of default QoS values used when creating *DataWriters*, and a *Subscriber* has a set of default QoS values used when creating *DataReaders*.

An entity's QoS are set when it is created. Once an entity is created, all of its QoS—for itself and its child *Entities*—are fixed unless you call `set_qos()` or `set_qos_with_profile()` on that entity. Calling `set_default_<entity>_qos()` on a parent entity will have no effect on child *Entities* that have already been created.

You can change these default values so that they are automatically applied when new child *Entities* are created. For example, suppose you want all *DataWriters* for a particular *Publisher* to have their RELIABILITY QoSPolicy set to RELIABLE. Instead of making this change for each *DataWriter* when it is created, you can change the default used when any *DataWriter* is created from the *Publisher* by using the *Publisher's* `set_default_datawriter_qos()` operation.

```
DDS_DataWriterQos default_datawriter_qos;
// get the current default values
publisher->get_default_datawriter_qos(default_datawriter_qos);
// change to desired default values
default_datawriter_qos.reliability.kind =
    DDS_RELIABLE_RELIABILITY_QOS;
// set the new default values
publisher->set_default_datawriter_qos(default_datawriter_qos);
// created datawriters will use new default values
datawriter =
    publisher->create_datawriter(topic, NULL, NULL, NULL);
```

It is not safe to get or set the default QoS values for an entity while another thread may be simultaneously calling `get_default_<entity>_qos()`, `set_default_<entity>_qos()`, or `create_<entity>()` with `DDS_<ENTITY>_QOS_DEFAULT` as the `qos` parameter (for the same entity).

Another way to make QoS changes is by using XML resources (files, strings). For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

4.1.7.2 Setting QoS During Entity Creation

If you only want to change a QoSPolicy for a particular entity, you can pass in the desired QoS Policies for an entity in its creation routine.

To customize an entity's QoS before creating it:

1. (C API Only) Initialize a QoS object with the appropriate INITIALIZER constructor.
2. Call the relevant `get_<entity>_default_qos()` method.
3. Modify the QoS values as desired.
4. Create the entity.

For example, to change the RELIABLE QoSPolicy for a *DataWriter* before creating it:

```
// Initialize the QoS object
DDS_DataWriterQos datawriter_qos;
// Get the default values
publisher->get_default_datawriter_qos(datawriter_qos);
// Modify the QoS values as desired
datawriter_qos.reliability.kind = DDS_BEST_EFFORT_RELIABILITY_QOS;
// Create the DataWriter with new values
datawriter = publisher->create_datawriter(
    topic, datawriter_qos, NULL, NULL);
```

Another way to set QoS during entity creation is by using a QoS profile. For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

4.1.7.3 Changing the QoS for an Existing Entity

Some policies can also be changed after the entity has been created. To change such a policy after the entity has been created, use the entity's `set_qos()` operation.

For example, suppose you want to tweak the DEADLINE QoS for an existing *DataWriter*:

```
DDS_DataWriterQos datawriter_qos;
// get the current values
datawriter->get_qos(datawriter_qos);
// make desired changes
datawriter_qos.deadline.period.sec = 3;
datawriter_qos.deadline.period.nanosec = 0;
// set new values
datawriter->set_qos(datawriter_qos);
```

Another way to make QoS changes is by using a QoS profile. For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Note: In the code examples presented in this section, we are not testing for the return code for the `set_qos()`, `set_default_*_qos()` functions. If the values used in the QoSPolicy structures are inconsistent then the functions will fail and return `INCONSISTENT_POLICY`. In addition, `set_qos()` may return `IMMUTABLE_POLICY` if you try to change a QoSPolicy on an *Entity* after that policy has become immutable. *User code should test for and address those anomalous conditions.*

4.1.7.4 Default QoS Values

Connex DDS provides special constants for each *Entity* type that can be used in `set_qos()` and `set_default_*_qos()` to reset the QoSPolicy values to the original DDS default values:

- `DDS_PARTICIPANT_QOS_DEFAULT`
- `DDS_PUBLISHER_QOS_DEFAULT`
- `DDS_SUBSCRIBER_QOS_DEFAULT`
- `DDS_DATAWRITER_QOS_DEFAULT`
- `DDS_DATAREADER_QOS_DEFAULT`
- `DDS_TOPIC_QOS_DEFAULT`

For example, if you want to set a *DataWriter's* QoS back to their DDS-specified default values:

```
datawriter->set_qos(DDS_DATAWRITER_QOS_DEFAULT);
```

Or if you want to reset the default QoS Policies used by a *Publisher* to create *DataWriters* back to their DDS-specified default values:

```
publisher->set_default_datawriter_qos(DDS_DATAWRITER_QOS_DEFAULT);
```

These defaults *cannot* be used to initialize a QoS structure for an entity. For example, the following is NOT allowed:

```
DataWriterQos dataWriterQos = DATAWRITER_QOS_DEFAULT;
// modify QoS...
create_datawriter(dataWriterQos);
```

4.2 QoS Policies

Connex DDS's behavior is controlled by the Quality of Service (QoS) policies of the data communication *Entities* (*DomainParticipant*, *Topic*, *Publisher*, *Subscriber*, *DataWriter*, and *DataReader*) used in your applications. This section summarizes each of the QoS Policies that you can set for the various *Entities*.

The *QoSPolicy* class is the abstract base class for all the QoS Policies. It provides the basic mechanism for an application to specify quality of service parameters. [Table 4.2 QoS Policies](#) lists each supported QoS Policy (in alphabetical order), provides a summary, and points to a section in the manual that provides further details.

The detailed description of a QoS Policy that applies to multiple *Entities* is provided in the first chapter that discusses an *Entity* whose behavior the QoS affects. Otherwise, the discussion of a QoS Policy can be found in the chapter of the particular *Entity* to which the policy applies. As you will see in the detailed

description sections, all QosPolicies have one or more parameters that are used to configure the policy. The how's and why's of tuning the parameters are also discussed in those sections.

As first discussed in [Controlling Behavior with Quality of Service \(QoS\) Policies \(Section 2.6.1 on page 19\)](#), QosPolicies may interact with each other, and certain values of QosPolicies can be incompatible with the values set for other policies.

The `set_qos()` operation will fail if you attempt to specify a set of values would result in an inconsistent set of policies. To indicate a failure, `set_qos()` will return `INCONSISTENT_POLICY`. [QoS Requested vs. Offered Compatibility—the RxO Property \(Section 4.2.1 on page 167\)](#) provides further information on QoS compatibility within an *Entity* as well as across matching *Entities*, as does the discussion/reference section for each QosPolicy listed in [Table 4.2 QosPolicies](#).

The values of some QosPolicies cannot be changed after the *Entity* is created or after the *Entity* is enabled. Others may be changed at any time. The detailed section on each QosPolicy states when each policy can be changed. If you attempt to change a QosPolicy after it becomes immutable (because the associated *Entity* has been created or enabled, depending on the policy), `set_qos()` will fail with a return code of `IMMUTABLE_POLICY`.

Table 4.2 QosPolicies

QosPolicy	Summary
Asynchronous-Publisher	Configures the mechanism that sends user data in an external middleware thread. See ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1 on page 313) .
Availability	This QoS policy is used in the context of two features: For a <i>Collaborative DataWriter</i> , specifies the group of <i>DataWriters</i> expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip DDS samples. For a <i>Durable Subscription</i> , configures a set of Durable Subscriptions on a <i>DataWriter</i> . See AVAILABILITY QosPolicy (DDS Extension) (Section 6.5.1 on page 337) .
Batch	Specifies and configures the mechanism that allows Connex DDS to collect multiple DDS data samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput. See BATCH QosPolicy (DDS Extension) (Section 6.5.2 on page 341) .
Database	Various settings and resource limits used by Connex DDS to control its internal database. See DATABASE QosPolicy (DDS Extension) (Section 8.5.1 on page 577) .
DataReaderProtocol	This QosPolicy configures the Connex DDS on-the-network protocol, RTPS. See DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1 on page 511) .
DataReaderResourceLimits	Various settings that configure how <i>DataReaders</i> allocate and use physical memory for internal resources. See DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2 on page 517) .
DataWriterProtocol	This QosPolicy configures the Connex DDS on-the-network protocol, RTPS. See DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.3 on page 347) .

Table 4.2 QosPolicies

QosPolicy	Summary
DataWriterResourceLimits	Controls how many threads can concurrently block on a write() call of this <i>DataWriter</i> . Also controls the number of batches managed by the <i>DataWriter</i> and the instance-replacement kind used by the <i>DataWriter</i> . See DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.4 on page 359) .
Deadline	For a <i>DataReader</i> , specifies the maximum expected elapsed time between arriving DDS data samples. For a <i>DataWriter</i> , specifies a commitment to publish DDS samples with no greater elapsed time between them. See DEADLINE QosPolicy (Section 6.5.5 on page 363) .
DestinationOrder	Controls how Connex DDS will deal with data sent by multiple <i>DataWriters</i> for the same topic. Can be set to "by reception timestamp" or to "by source timestamp." See DESTINATION_ORDER QosPolicy (Section 6.5.6 on page 365) .
Discovery	Configures the mechanism used by Connex DDS to automatically discover and connect with new remote applications. See DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2 on page 580) .
DiscoveryConfig	Controls the amount of delay in discovering <i>Entities</i> in the system and the amount of discovery traffic in the network. See DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3 on page 585) .
DomainParticipantResourceLimits	Various settings that configure how <i>DomainParticipants</i> allocate and use physical memory for internal resources, including the maximum sizes of various properties. See DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4 on page 593) .
Durability	Specifies whether or not Connex DDS will store and deliver data that were previously published to new <i>DataReaders</i> . See DURABILITY QosPolicy (Section 6.5.7 on page 368) .
DurabilityService	Various settings to configure the external Persistence Service used by Connex DDS for <i>DataWriters</i> with a Durability QoS setting of Persistent Durability. See DURABILITY_SERVICE QosPolicy (Section 6.5.8 on page 372) .
EntityFactory	Controls whether or not child <i>Entities</i> are created in the enabled state. See ENTITYFACTORY QosPolicy (Section 6.4.2 on page 315) .
EntityName	Assigns a name and role_name to an <i>Entity</i> . See ENTITY_NAME QosPolicy (DDS Extension) (Section 6.5.9 on page 374) .
Event	Configures the <i>DomainParticipant's</i> internal thread that handles timed events. See EVENT QosPolicy (DDS Extension) (Section 8.5.5 on page 602) .
ExclusiveArea	Configures multi-thread concurrency and deadlock prevention capabilities. See EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3 on page 318) .
GroupData	Along with TOPIC_DATA QosPolicy (Section 5.2.1 on page 209) and USER_DATA QosPolicy (Section 6.5.26 on page 417) , this QosPolicy is used to attach a buffer of bytes to Connex DDS's discovery meta-data. See GROUP_DATA QosPolicy (Section 6.4.4 on page 320) .

Table 4.2 QoS Policies

QoS Policy	Summary
History	Specifies how much data must be stored by Connex DDS for the <i>DataWriter</i> or <i>DataReader</i> . This QoS Policy affects the RELIABILITY QoS Policy (Section 6.5.19 on page 400) as well as the DURABILITY QoS Policy (Section 6.5.7 on page 368) . See HISTORY QoS Policy (Section 6.5.10 on page 376) .
LatencyBudget	Suggestion to Connex DDS on how much time is allowed to deliver data. See LATENCYBUDGET QoS Policy (Section 6.5.11 on page 380) .
Lifespan	Specifies how long Connex DDS should consider data sent by an user application to be valid. See LIFESPAN QoS Policy (Section 6.5.12 on page 381) .
Liveliness	Specifies and configures the mechanism that allows <i>DataReaders</i> to detect when <i>DataWriters</i> become disconnected or "dead." See LIVELINESS QoS Policy (Section 6.5.13 on page 382) .
Logging	Configures the properties associated with Connex DDS logging. See LOGGING QoS Policy (DDS Extension) (Section 8.4.1 on page 572) .
MultiChannel	Configures a <i>DataWriter</i> 's ability to send data on different multicast groups (addresses) based on the value of the data. See MULTI_CHANNEL QoS Policy (DDS Extension) (Section 6.5.14 on page 386) .
Ownership	Along with Ownership Strength, specifies if <i>DataReaders</i> for a topic can receive data from multiple <i>DataWriters</i> at the same time. See OWNERSHIP QoS Policy (Section 6.5.15 on page 389) .
OwnershipStrength	Used to arbitrate among multiple <i>DataWriters</i> of the same instance of a Topic when Ownership QoS Policy is EXCLUSIVE. See OWNERSHIP_STRENGTH QoS Policy (Section 6.5.16 on page 393) .
Partition	Adds string identifiers that are used for matching <i>DataReaders</i> and <i>DataWriters</i> for the same Topic. See PARTITION QoS Policy (Section 6.4.5 on page 323) .
Presentation	Controls how Connex DDS presents data received by an application to the <i>DataReaders</i> of the data. See PRESENTATION QoS Policy (Section 6.4.6 on page 330) .
Profile	Configures the way that XML documents containing QoS profiles are loaded by RTI. See PROFILE QoS Policy (DDS Extension) (Section 8.4.2 on page 573) .
Property	Stores name/value(string) pairs that can be used to configure certain parameters of Connex DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See PROPERTY QoS Policy (DDS Extension) (Section 6.5.17 on page 394) .
PublishMode	Specifies how Connex DDS sends application data on the network. By default, data is sent in the user thread that calls the <i>DataWriter</i> 's write() operation. However, this QoS Policy can be used to tell Connex DDS to use its own thread to send the data. See PUBLISH_MODE QoS Policy (DDS Extension) (Section 6.5.18 on page 397) .
ReaderDataLifeCycle	Controls how a <i>DataReader</i> manages the lifecycle of the data that it has received. See READER_DATA_LIFECYCLE QoS Policy (Section 7.6.3 on page 523) .

Table 4.2 QosPolicies

QosPolicy	Summary
ReceiverPool	Configures threads used by Connex DDS to receive and process data from transports (for example, UDP sockets). See RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.6 on page 604) .
Reliability	Specifies whether or not Connex DDS will deliver data reliably. See RELIABILITY QosPolicy (Section 6.5.19 on page 400) .
ResourceLimits	Controls the amount of physical memory allocated for <i>Entities</i> , if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) .
Service	Intended for use by RTI infrastructure services. User applications should not modify its value. See SERVICE QosPolicy (DDS Extension) (Section 6.5.21 on page 408) .
SystemResourceLimits	Configures <i>DomainParticipant</i> -independent resources used by Connex DDS. Mainly used to change the maximum number of <i>DomainParticipants</i> that can be created within a single process (address space). See SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 8.4.3 on page 575) .
TimeBasedFilter	Set by a <i>DataReader</i> to limit the number of new data values received over a period of time. See TIME_BASED_FILTER QosPolicy (Section 7.6.4 on page 526) .
TopicData	Along with Group Data QosPolicy and User Data QosPolicy, used to attach a buffer of bytes to Connex DDS's discovery meta-data. See TOPIC_DATA QosPolicy (Section 5.2.1 on page 209) .
TransportBuiltin	Specifies which built-in transport plugins are used. See TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.7 on page 606) .
TransportMulticast	Specifies the multicast address on which a <i>DataReader</i> wants to receive its data. Can specify a port number as well as a subset of the available transports with which to receive the multicast data. See TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5 on page 529) .
TransportMulticastMapping	Specifies the automatic mapping between a list of topic expressions and multicast address that can be used by a <i>DataReader</i> to receive data for a specific topic. See TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension) (Section 8.5.8 on page 608) .
TransportPriority	Set by a <i>DataWriter</i> or <i>DataReader</i> to tell Connex DDS that the data being sent is a different "priority" than other data. See TRANSPORT_PRIORITY QosPolicy (Section 6.5.22 on page 409) .
TransportSelection	Allows you to select which physical transports a <i>DataWriter</i> or <i>DataReader</i> may use to send or receive its data. See TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.23 on page 411) .
TransportUnicast	Specifies a subset of transports and port number that can be used by an <i>Entity</i> to receive data. See TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.24 on page 412) .
TypeConsistencyEnforcement	Defines rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it. See TYPE_CONSISTENCY_ENFORCEMENT QosPolicy (Section 7.6.6 on page 532) .

Table 4.2 QoS Policies

QoS Policy	Summary
TypeSupport	Used to attach application-specific value(s) to a <i>DataWriter</i> or <i>DataReader</i> . These values are passed to the serialization or deserialization routine of the associated data type. Also controls whether padding bytes are set to 0 during serialization. See TYPESUPPORT QoS Policy (DDS Extension) (Section 6.5.25 on page 416) .
UserData	Along with Topic Data QoS Policy and Group Data QoS Policy, used to attach a buffer of bytes to Connex DDS's discovery meta-data. See USER_DATA QoS Policy (Section 6.5.26 on page 417) .
WireProtocol	Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See WIRE_PROTOCOL QoS Policy (DDS Extension) (Section 8.5.9 on page 610) .
WriterDataLifeCycle	Controls how a <i>DataWriter</i> handles the lifecycle of the instances (keys) that the <i>DataWriter</i> is registered to manage. See WRITER_DATA_LIFECYCLE QoS Policy (Section 6.5.27 on page 419) .

4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property

Some QoS Policies that apply to *Entities* on the sending and receiving sides must have their values set in a compatible manner. This is known as the policy's 'requested vs. offered' (RxO) property. *Entities* on the publishing side 'offer' to provide a certain behavior. *Entities* on the subscribing side 'request' certain behavior. For Connex DDS to connect the sending entity to the receiving entity, the offered behavior must satisfy the requested behavior.

For some QoS Policies, the allowed values may be graduated in a way that the offered value will satisfy the requested value if the offered value is either greater than or less than the requested value. For example, if a *DataWriter*'s DEADLINE QoS Policy specifies a duration less than or equal to a *DataReader*'s DEADLINE QoS Policy, then the *DataWriter* is promising to publish data at least as fast or faster than the *DataReader* requires new data to be received. This is a compatible situation (see [DEADLINE QoS Policy \(Section 6.5.5 on page 363\)](#)).

Other QoS Policies require the values on the sending side and the subscribing side to be exactly equal for compatibility to be met. For example, if a *DataWriter*'s OWNERSHIP QoS Policy is set to SHARED, and the matching *DataReader*'s value is set to EXCLUSIVE, then this is an incompatible situation since the *DataReader* and *DataWriter* have different expectations of what will happen if more than one *DataWriter* publishes an instance of the *Topic* (see [OWNERSHIP QoS Policy \(Section 6.5.15 on page 389\)](#)).

Finally there are QoS Policies that do not require compatibility between the sending entity and the receiving entity, or that only apply to one side or the other. Whether or not related *Entities* on the publishing and subscribing sides must use compatible settings for a QoS Policy is indicated in the policy's RxO property, which is provided in the detailed section on each QoS Policy.

- **RxO = YES** The policy is set at both the publishing and subscribing ends and the values must be set in a compatible manner. What it means to be compatible is defined by the QoS Policy.

- **RxO = NO** The policy is set only on one end or at both the publishing and subscribing ends, but the two settings are independent. There the requested vs. offered semantics are not used for these QoS Policies.

For those QoS Policies that follow the RxO semantics, Connex DDS will compare the values of those policies for compatibility. If they are compatible, then Connex DDS will connect the sending entity to the receiving entity allowing data to be sent between them. If they are found to be incompatible, then Connex DDS will not interconnect the *Entities* preventing data to be sent between them.

In addition, Connex DDS will record this event by changing the associated communication status in both the sending and receiving applications, see [Types of Communication Status \(Section 4.3.1 on page 170\)](#). Also, if you have installed *Listeners* on the associated *Entities*, then Connex DDS will invoke the associated callback functions to notify user code that an incompatible QoS combination has been found, see [Types of Listeners \(Section 4.4.1 on page 177\)](#).

For *Publishers* and *DataWriters*, the status corresponding to this situation is **OFFERED_INCOMPATIBLE_QOS_STATUS**. For *Subscribers* and *DataReaders*, the corresponding status is **REQUESTED_INCOMPATIBLE_QOS_STATUS**. The question of why a *DataReader* is not receiving data sent from a matching *DataWriter* can often be answered if you have instrumented the application with *Listeners* for the statuses noted previously.

4.2.2 Special QoS Policy Handling Considerations for C

Many QoS Policy structures contain variable-length sequences to store their parameters. In the C++, C++/CLI, C# and Java languages, the memory allocation related to sequences are handled automatically through constructors/destructors and overloaded operators. However, the C language is limited in what it provides to automatically handle memory management. Thus, Connex DDS provides functions and macros in C to initialize, copy, and finalize (free) QoS Policy structures defined for *Entities*.

In the C language, it is not safe to use an *Entity*'s QoS Policy structure declared in user code unless it has been initialized first. In addition, user code should always finalize an *Entity*'s QoS Policy structure to release any memory allocated for the sequences—even if the *Entity*'s QoS Policy structure was declared as a local, stack variable.

Thus, for a general *Entity*'s QoS Policy, Connex DDS will provide:

- **DDS_<Entity>Qos_INITIALIZER** This is a macro that should be used when a **DDS_<Entity>Qos** structure is declared in a C application.

```
struct DDS_<Entity>Qos qos = DDS_<Entity>Qos_INITIALIZER;
```

- **DDS_<Entity>Qos_initialize()** This is a function that can be used to initialize a **DDS_<Entity>Qos** structure instead of the macro above.

```
struct DDS_<Entity>Qos qos;
DDS_<Entity>QoS_initialize(&qos);
```

- **DDS_<Entity>Qos_finalize()** This is a function that should be used to finalize a **DDS_<Entity>Qos** structure when the structure is no longer needed. It will free any memory allocated for sequences contained in the structure.

```
struct DDS_<Entity>Qos qos = DDS_<Entity>QoS_INITIALIZER;
...
<use qos>
...
// now done with qos
DDS_<Entity>QoS_finalize(&qos);
```

- **DDS<Entity>Qos_copy()** This is a function that can be used to copy one **DDS_<Entity>Qos** structure to another. It will copy the sequences contained in the source structure and allocate memory for sequence elements if needed. In the code below, both **dstQos** and **srcQos** must have been initialized at some point earlier in the code.

```
DDS_<Entity>QoS_copy(&dstQos, &srcQos);
```

4.3 Statuses

This section describes the different *statuses* that exist for an entity. A status represents a state or an event regarding the entity. For instance, maybe Connex DDS found a matching *DataReader* for a *DataWriter*, or new data has arrived for a *DataReader*.

Your application can retrieve an *Entity's* status by:

- explicitly checking for *any* status changes with **get_status_changes()**.
- explicitly checking a *specific* status with **get_<status_name>_status()**.
- using a *Listener*, which provides asynchronous notification when a status changes.
- using *StatusConditions* and *WaitSets*, which provide a way to wait for status changes.

If you want your application to be notified of status changes asynchronously: create and install a *Listener* for the *Entity*. Then internal Connex DDS threads will call the listener methods when the status changes. See [Listeners \(Section 4.4 on page 177\)](#).

If you want your application to wait for status changes: set up *StatusConditions* to indicate the statuses of interest, attach the *StatusConditions* to a *WaitSet*, and then call the *WaitSet's* **wait()** operation. The call to **wait()** will block until statuses in the attached *Conditions* changes (or until a timeout period expires). See [Conditions and WaitSets \(Section 4.6 on page 187\)](#).

This section includes the following:

4.3.1 Types of Communication Status

Each *Entity* is associated with a set of *Status* objects representing the “communication status” of that *Entity*. The list of statuses actively monitored by Connex DDS is provided in [Table 4.3 Communication Statuses](#). A status structure contains values that give you more information about the status; for example, how many times the event has occurred since the last time the user checked the status, or how many time the event has occurred in total.

Changes to status values cause activation of corresponding *StatusCondition* objects and trigger invocation of the corresponding *Listener* functions to asynchronously inform the application that the status has changed. For example, a change in a *Topic*'s **INCONSISTENT_TOPIC_STATUS** may trigger the *TopicListener*'s **on_inconsistent_topic()** callback routine (if such a *Listener* is installed).

Table 4.3 Communication Statuses

Related Entity	Status (DDS_*_STATUS)	Description	Reference
Topic	INCONSISTENT_TOPIC	Another <i>Topic</i> exists with the same name but different characteristics—for example, a different type.	INCONSISTENT_TOPIC Status (Section 5.3.1 on page 211)
Data-Writer	APPLICATION_ACKNOWLEDGMENT	This status indicates that a <i>DataWriter</i> has received an application-level acknowledgment for a DDS sample. The listener provides the identities of the DDS sample and acknowledging <i>DataReader</i> , as well as user-specified response data sent from the <i>DataReader</i> by the acknowledgment message.	Application Acknowledgment (Section 6.3.12 on page 288)
	DATA_WRITER_CACHE	The status of the <i>DataWriter</i> 's cache. This status does not have a Listener.	DATA_WRITER_CACHE_STATUS (Section 6.3.6.2 on page 272)
	DATA_WRITER_PROTOCOL	The status of a <i>DataWriter</i> 's internal protocol related metrics (such as the number of DDS samples pushed, pulled, filtered) and the status of wire protocol traffic. This status does not have a Listener.	DATA_WRITER_PROTOCOL_STATUS (Section 6.3.6.3 on page 273)

Table 4.3 Communication Statuses

Related Entity	Status (DDS_*_STATUS)	Description	Reference
Data-Writer cont'd	LIVELINESS_LOST	The liveliness that the <i>DataWriter</i> has committed to (through its Liveliness QoSPolicy) was not respected (assert_liveliness() or write() not called in time), thus <i>DataReaders</i> may consider the <i>DataWriter</i> as no longer active.	LIVELINESS_LOST Status (Section 6.3.6.4 on page 276)
	OFFERED_DEADLINE_MISSED	The deadline that the <i>DataWriter</i> has committed through its Deadline QoSPolicy was not respected for a specific instance of the <i>Topic</i> .	OFFERED_DEADLINE_MISSED Status (Section 6.3.6.5 on page 277)
	OFFERED_INCOMPATIBLE_QOS	An offered QoSPolicy value was incompatible with what was requested by a <i>DataReader</i> of the same <i>Topic</i> .	OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.6.6 on page 277)
	PUBLICATION_MATCHED	The <i>DataWriter</i> found a <i>DataReader</i> that matches the <i>Topic</i> , has compatible QoSs and a common partition, or a previously matched <i>DataReader</i> has been deleted.	PUBLICATION_MATCHED Status (Section 6.3.6.7 on page 278)
	RELIABLE_WRITER_CACHE_CHANGED	The number of unacknowledged DDS samples in a reliable <i>DataWriter's</i> cache has reached one of the predefined trigger points.	RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.6.8 on page 279)
	RELIABLE_READER_ACTIVITY_CHANGED	One or more reliable <i>DataReaders</i> has either been discovered, deleted, or changed between active and inactive state as specified by the LivelinessQoSPolicy of the <i>DataReader</i> .	RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.6.9 on page 281)
Subscriber	DATA_ON_READERS	New data is available for any of the readers that were created from the <i>Subscriber</i> .	Statuses for Subscribers (Section 7.2.9 on page 458)

Table 4.3 Communication Statuses

Related Entity	Status (DDS_*_STATUS)	Description	Reference
Data-Reader	DATA_AVAILABLE	New data (one or more DDS samples) are available for the specific <i>DataReader</i> .	DATA_AVAILABLE Status (Section 7.3.7.1 on page 471)
	DATA_READER_CACHE	The status of the reader's cache. This status does not have a Listener.	DATA_READER_CACHE_STATUS (Section 7.3.7.2 on page 471)
	DATA_READER_PROTOCOL	The status of a <i>DataReader</i> 's internal protocol related metrics (such as the number of DDS samples received, filtered, rejected) and the status of wire protocol traffic. This status does not have a Listener.	DATA_READER_PROTOCOL_STATUS (Section 7.3.7.3 on page 472)
	LIVELINESS_CHANGED	The liveliness of one or more <i>DataWriters</i> that were writing instances read by the <i>DataReader</i> has either been discovered, deleted, or changed between active and inactive state as specified by the <i>LivelinessQosPolicy</i> of the <i>DataWriter</i> .	LIVELINESS_CHANGED Status (Section 7.3.7.4 on page 475)
Data-Reader cont'd	REQUESTED_DEADLINE_MISSED	New data was not received for an instance of the <i>Topic</i> within the time period set by the <i>DataReader</i> 's <i>Deadline QosPolicy</i> .	REQUESTED_DEADLINE_MISSED Status (Section 7.3.7.5 on page 476)
	REQUESTED_INCOMPATIBLE_QOS	A requested <i>QosPolicy</i> value was incompatible with what was offered by a <i>DataWriter</i> of the same <i>Topic</i> .	REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.7.6 on page 477)
	SAMPLE_LOST	A DDS sample sent by Connex DDS has been lost (never received).	SAMPLE_LOST Status (Section 7.3.7.7 on page 478)
	SAMPLE_REJECTED	A received DDS sample has been rejected due to a resource limit (buffers filled).	SAMPLE_REJECTED Status (Section 7.3.7.8 on page 479)
	SUBSCRIPTION_MATCHED	The <i>DataReader</i> has found a <i>DataWriter</i> that matches the <i>Topic</i> , has compatible QoSs and a common partition, or an existing matched <i>DataWriter</i> has been deleted.	SUBSCRIPTION_MATCHED Status (Section 7.3.7.9 on page 482)

Statuses can be grouped into two categories:

- Plain communication status:

In addition to a flag that indicates whether or not a status has changed, a *plain* communication status also contains state and thus has a corresponding structure to hold its current value.

- Read communication status:

A read communication status is more like an event and has no state other than whether or not it has occurred. Only two statuses listed in [Table 4.3 Communication Statuses](#) are *read* communications statuses: **DATA_AVAILABLE** and **DATA_ON_READERS**.

As mentioned in [Getting Status and Status Changes \(Section 4.1.4 on page 157\)](#), all *Entities* have a **get_status_changes()** operation that can be used to explicitly poll for changes in any status related to the entity. For *plain* statuses, each entry has operations to get the current value of the status; for example, the *Topic* class has a **get_inconsistent_topic_status()** operation. For *read* statuses, your application should use the **take()** operation on the *DataReader* to retrieve the newly arrived data that is indicated by **DATA_AVAILABLE** and **DATA_ON_READER**.

Note that the two read communication statuses do not change independently. If data arrives for a *DataReader*, then its **DATA_AVAILABLE** status changes. At the same time, the **DATA_ON_READERS** status changes for the *DataReader*'s *Subscriber*.

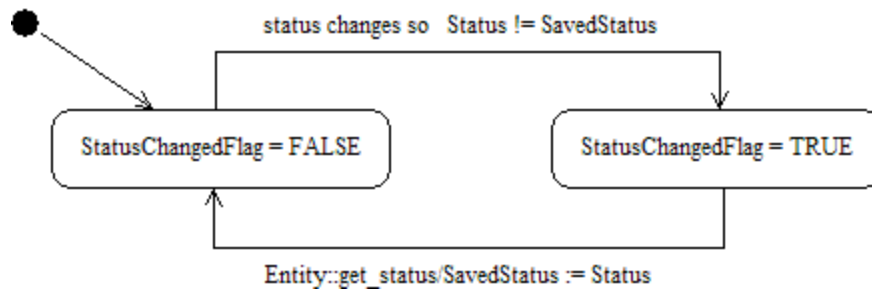
Both types of status have a **StatusChangedFlag**. This flag indicates whether that particular communication status has changed since the last time the status was read by the application. The way the **StatusChangedFlag** is maintained is slightly different for the *plain* communication status and the *read* communication status, as described in the following sections:

- [Changes in Plain Communication Status \(Section 4.3.1.1 below\)](#)
- [Changes in Read Communication Status \(Section 4.3.1.2 on the next page\)](#)

4.3.1.1 Changes in Plain Communication Status

As seen in [Figure 4.2 Status Changes for Plain Communication Status on the next page](#), for the plain communication status, the **StatusChangedFlag** flag is initially set to **FALSE**. It becomes **TRUE** whenever the plain communication status changes and is reset to **FALSE** each time the application accesses the plain communication status via the proper **get_*_status()** operation.

Figure 4.2 Status Changes for Plain Communication Status



The communication status is also reset to FALSE whenever the associated listener operation is called, as the listener implicitly accesses the status which is passed as a parameter to the operation.

The fact that the status is reset prior to calling the listener means that if the application calls the `get_*_status()` operation from inside the listener, it will see the status already reset.

An exception to this rule is when the associated listener is the 'nil' listener. The 'nil' listener is treated as a NO-OP and the act of calling the 'nil' listener does not reset the communication status. (See [Types of Listeners \(Section 4.4.1 on page 177\)](#).)

For example, the value of the StatusChangedFlag associated with the **REQUESTED_DEADLINE_MISSED** status will become TRUE each time new deadline occurs (which increases the RequestedDeadlineMissed status' `total_count` field). The value changes to FALSE when the application accesses the status via the corresponding `get_requested_deadline_missed_status()` operation on the proper Entity.

4.3.1.2 Changes in Read Communication Status

As seen in [Figure 4.3 Status Changes for Read Communication Status on the facing page](#), for the read communication status, the StatusChangedFlag flag is initially set to FALSE. The StatusChangedFlag becomes TRUE when either a DDS data sample arrives or the ViewStateKind, SampleStateKind, or InstanceStateKind of any existing DDS sample changes for any reason other than a call to one of the read/take operations. Specifically, any of the following events will cause the StatusChangedFlag to become TRUE:

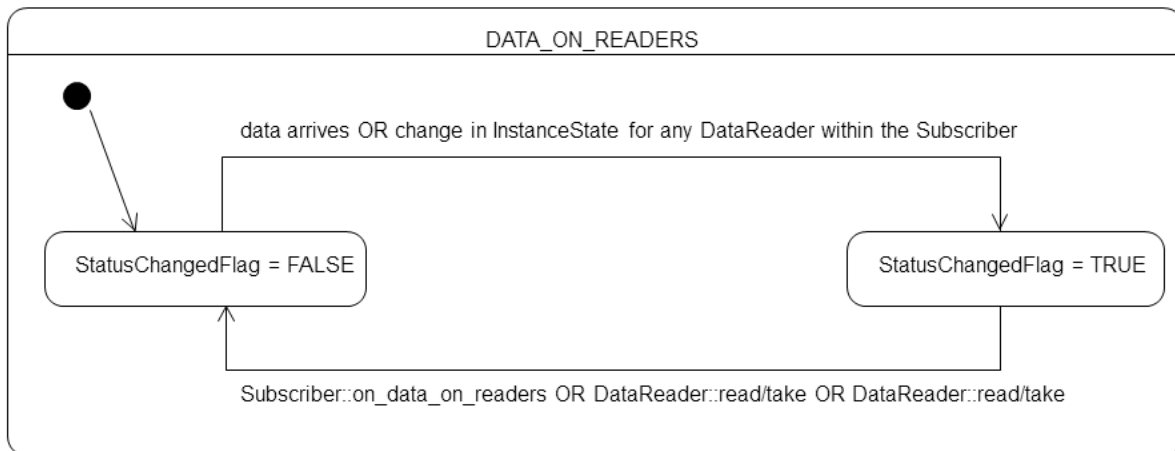
- The arrival of new data.
- A change in the InstanceStateKind of a contained instance. This can be caused by either:
 - Notification that an instance has been disposed by:
 - the *DataWriter* that owns it, if OWNERSHIP = EXCLUSIVE
 - or by any *DataWriter*, if OWNERSHIP = SHARED
 - The loss of liveness of the *DataWriter* of an instance for which there is no other *DataWriter*.

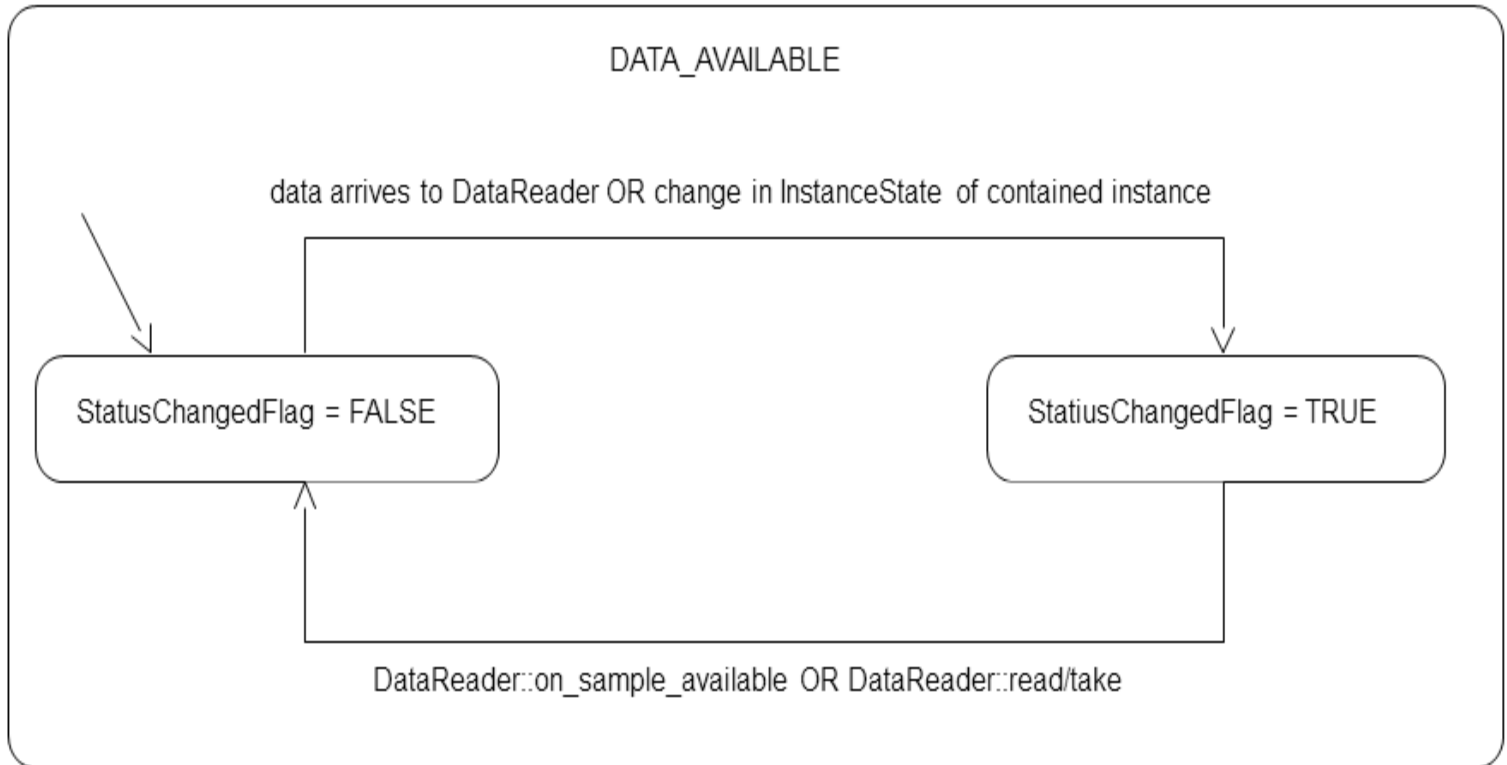
- The arrival of the notification that an instance has been unregistered by the only *DataWriter* that is known to be writing the instance.

Depending on the **kind** of **StatusChangedFlag**, the flag transitions to FALSE (that is, the status is reset) as follows:

- The **DATA_AVAILABLE** StatusChangedFlag becomes FALSE when either **on_data_available()** is called or the read/take operation (or their variants) is called on the associated *DataReader*.
- The **DATA_ON_READERS** StatusChangedFlag becomes FALSE when any of the following occurs:
 - **on_data_on_readers()** is called.
 - **on_data_available()** is called on any *DataReader* belonging to the *Subscriber*.
 - **read()**, **take()**, or one of their variants is called on any *DataReader* belonging to the *Subscriber*.

Figure 4.3 Status Changes for Read Communication Status





4.3.2 Special Status-Handling Considerations for C

Some status structures contain variable-length sequences to store their values. In the C++, C++/CLI, C# and Java languages, the memory allocation related to sequences are handled automatically through constructors/destructors and overloaded operators. However, the C language is limited in what it provides to automatically handle memory management. Thus, Connex DDS provides functions and macros in C to initialize, copy, and finalize (free) status structures.

In the C language, it is not safe to use a status structure that has internal sequences declared in user code unless it has been initialized first. In addition, user code should always finalize a status structure to release any memory allocated for the sequences—even if the status structure was declared as a local, stack variable.

Thus, for a general status structure, Connex DDS will provide:

- `DDS_<STATUS>STATUS_INITIALIZER` This is a macro that should be used when a `DDS_<Status>Status` structure is declared in a C application.

```

struct DDS_<Status>Status status =
    DDS_<Status>Status_INITIALIZER;
  
```

- **DDS_<Status>Status_initialize()** This is a function that can be used to initialize a **DDS_<Status>Status** structure instead of the macro above.

```
struct DDS_<Status>Status status;
DDS_<Status>Status_initialize(&Status);
```

- **DDS_<Status>Status_finalize()** This is a function that should be used to finalize a **DDS_<Status>Status** structure when the structure is no longer needed. It will free any memory allocated for sequences contained in the structure.

```
struct DDS_<Status>Status status =
    DDS_<Status>Status_INITIALIZER;
...
<use status>
...
// now done with Status
DDS_<Status>Status_finalize(&status);
```

- **DDS<Status>Status_copy()** This is a function that can be used to copy one **DDS_<Status>Status** structure to another. It will copy the sequences contained in the source structure and allocate memory for sequence elements if needed. In the code below, both **dstStatus** and **srcStatus** must have been initialized at some point earlier in the code.

```
DDS_<Status>Status_copy(&dstStatus, &srcStatus);
```

Note that many status structures do not have sequences internally. For those structures, you do not need to use the macro and methods provided above. However, they have still been created for your convenience.

4.4 Listeners

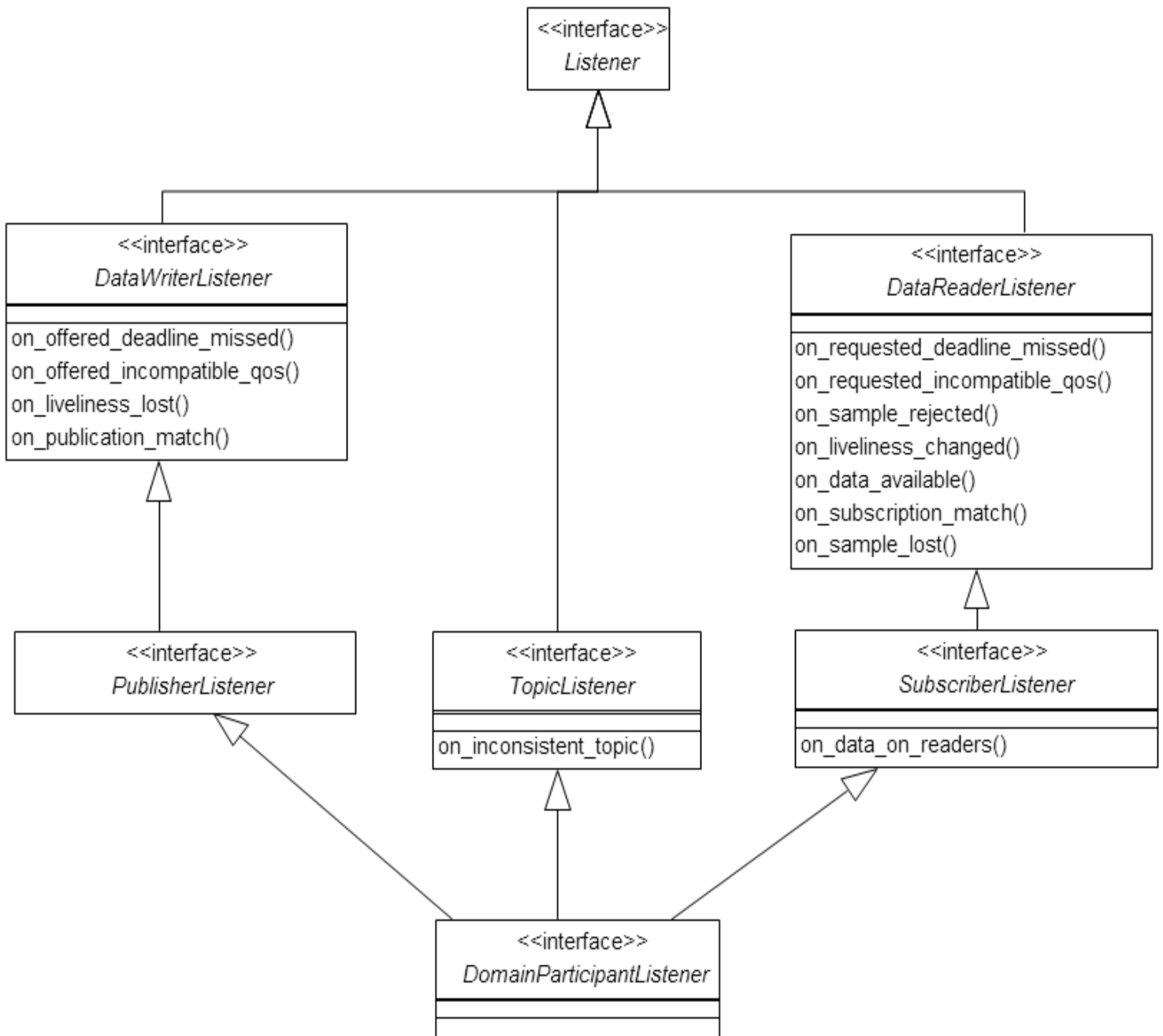
Listeners are triggered by changes in an entity's status. For instance, maybe Connex DDS found a matching *DataReader* for a *DataWriter*, or new data has arrived for a *DataReader*.

This section describes *Listeners* and how to use them:

4.4.1 Types of Listeners

The *Listener* class is the abstract base class for all listeners. Each entity class (*DomainParticipant*, *Topic*, *Publisher*, *DataWriter*, *Subscriber*, and *DataReader*) has its own derived *Listener* class that add methods for handling entity-specific statuses. The hierarchy of *Listener* classes is presented in [Figure 4.4 Listener Class Hierarchy on the next page](#). The methods are called by an internal Connex DDS thread when the corresponding status for the *Entity* changes value.

Figure 4.4 Listener Class Hierarchy



You can choose which changes in status will trigger a callback by installing a listener with a bit-mask. Bits in the mask correspond to different statuses. The bits that are true indicate that the listener will be called back when there are changes in the corresponding status.

You can specify a listener and set its bit-mask before or after you create an *Entity*:

During Entity creation:

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
    DDS_DATA_AVAILABLE_STATUS;
datareader = subscriber->create_datareader(topic,
    DDS_DATAREADER_QOS_DEFAULT,
    listener, mask);
```

or afterwards:

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
    DDS_DATA_AVAILABLE_STATUS;
datareader->set_listener(listener, mask);
```

As you can see in the above examples, there are two components involved when setting up listeners: the listener itself and the mask. Both of these can be null. [Table 4.4 Effect of Different Combinations of Listeners and Status Bit Masks](#) describes what happens when a status change occurs. See [Hierarchical Processing of Listeners \(Section 4.4.4 on the next page\)](#) for more information.

Table 4.4 Effect of Different Combinations of Listeners and Status Bit Masks

	No Bits Set in Mask	Some/All Bits Set in Mask
Listener is Specified	Connex DDS finds the next most relevant listener for the changed status.	For the statuses that are enabled in the mask, the most relevant listener will be called. The 'statusChangedFlag' for the relevant status is reset.
Listener is NULL	Connex DDS behaves as if the listener is not installed and finds the next most relevant listener for that status.	Connex DDS behaves as if the listener callback is installed, but the callback is doing nothing. This is called a 'nil' listener.

4.4.2 Creating and Deleting Listeners

There is no factory for creating or deleting a *Listener*; use the natural means in each language binding (for example, “new” or “delete” in C++ or Java). For example:

```
class HelloWorldListener : public DDSDataReaderListener {
    virtual void on_data_available(DDSDataReader* reader);
};
void HelloWorldListener::on_data_available(DDSDataReader* reader)
{
    printf("received data\n");
}
// Create a Listener
HelloWorldListener *reader_listener = NULL;
reader_listener = new HelloWorldListener();
// Delete a Listener
delete reader_listener;
```


A listener cannot be deleted until the entity it is attached to has been deleted. For example, you must delete the *DataReader* before deleting the *DataReader*'s listener.

Note: Due to a thread-safety issue, the destruction of a *DomainParticipantListener* from an enabled *DomainParticipant* should be avoided—even if the *DomainParticipantListener* has been removed from the *DomainParticipant*. (This limitation does not affect the Java API.)

4.4.3 Special Considerations for Listeners in C

In C, a *Listener* is a structure with function pointers to the user callback routines. Often, you may only be interested in a subset of the statuses that can be monitored with the *Listener*. In those cases, you may not set all of the functions pointers in a listener structure to a valid function. In that situation, we recommend that the unused, callback-function pointers are set to **NULL**. While setting the **DDS_StatusMask** to enable only the callbacks for the statuses in which you are interested (and thus only enabling callbacks on the functions that actually exist) is safe, we still recommend that you clear all of the unused callback pointers in the *Listener* structure.

To help, in the C language, we provide a macro that can be used to initialize a Listener structure so that all of its callback pointers are set to **NULL**. For example

```
DDS_<Entity>Listener listener = DDS_<Entity>Listener_INITIALIZER;
// now only need to set the listener callback pointers for statuses // to be monitored
```

There is no need to do this in languages other than C.

4.4.4 Hierarchical Processing of Listeners

As seen in [Listener Class Hierarchy \(Section Figure 4.4 on page 178\)](#), *Listeners* for some *Entities* derive from the Connex DDS *Listeners* for related *Entities*. This means that the derived *Listener* has all of the methods of its parent class. You can install *Listeners* at all levels of the object hierarchy. At the top is the *DomainParticipantListener*; only one can be installed in a *DomainParticipant*. Then every *Subscriber* and *Publisher* can have their own *Listener*. Finally, each *Topic*, *DataReader* and *DataWriter* can have their own listeners. All are optional.

Suppose, however, that an *Entity* does not install a *Listener*, or installs a *Listener* that does not have particular communication status selected in the bitmask. In this case, if/when that particular status changes for that *Entity*, the corresponding *Listener* for that *Entity*'s parent is called. Status changes are “propagated” from child *Entity* to parent *Entity* until a *Listener* is found that is registered for that status. Connex DDS will give up and drop the status-change event only if no *Listeners* have been installed in the object hierarchy to be called back for the specific status. This is true for *plain* communication statuses. *Read* communication statuses are handle somewhat differently, see [Processing Read Communication Statuses \(Section 4.4.4.1 on the facing page\)](#).

For example, suppose that Connex DDS finds a matching *DataWriter* for a local *DataReader*. This event will change the **SUBSCRIPTION_MATCHED** status. So the local *DataReader* object is checked to see

if the application has installed a listener that handles the **SUBSCRIPTION_MATCH** status. If not, the *Subscriber* that created the *DataReader* is checked to see if it has a listener installed that handles the same event. If not, the *DomainParticipant* is checked. The *DomainParticipantListener* methods are called only if none of the descendent *Entities* of the *DomainParticipant* have listeners that handle the particular status that has changed. Again, all listeners are optional. Your application does not have to handle any communication statuses.

[Table 4.5 Listener Callback Functions](#) lists the callback functions that are available for each *Entity*'s status listener.

Table 4.5 Listener Callback Functions

Entity Listener for:		Callback Functions
DomainParticipants	Topics	on_inconsistent_topic()
	Publishers and DataWriters	on_liveliness_lost()
		on_offered_deadline_missed()
		on_offered_incompatible_qos()
		on_publication_matched()
		on_reliable_reader_activity_changed()
		on_reliable_writer_cache_changed()
	Subscribers	on_data_on_readers()
	Subscribers and DataReaders	on_data_available
		on_liveliness_changed()
		on_requested_deadline_missed()
		on_requested_incompatible_qos()
		on_sample_lost()
		on_sample_rejected()
		on_subscription_matched()

4.4.4.1 Processing Read Communication Statuses

The processing of the **DATA_ON_READERS** and **DATA_AVAILABLE** read communication statuses are handled slightly differently since, when new data arrives for a *DataReader*, both statuses change simultaneously. However, only one, if any, *Listener* will be called to handle the event.

If there is a *Listener* installed to handle the **DATA_ON_READERS** status in the *DataReader*'s *Subscriber* or in the *DomainParticipant*, then that *Listener*'s **on_data_on_readers()** function will be called back. The *DataReaderListener*'s **on_data_available()** function is called only if the **DATA_ON_READERS** status is not handle by any relevant listeners.

This can be useful if you have generic processing to do whenever new data arrives for any *DataReader*. You can execute the generic code in the **on_data_on_readers()** method, and then dispatch the processing of the actual data to the specific *DataReaderListener*'s **on_data_available()** function by calling the **notify_datareaders()** method on the *Subscriber*.

For example:

```
void on_data_on_readers (DDSSubscriber *subscriber)
{
    // Do some general processing that needs to be done
    // whenever new data arrives, but is independent of
    // any particular DataReader
    < generic processing code here >
    // Now dispatch the actual processing of the data
    // to the specific DataReader for which the data
    // was received
    subscriber->notify_datareaders();
}
```

4.4.5 Operations Allowed within Listener Callbacks

Due to the potential for deadlock, some Connex DDS APIs should not be invoked within the functions of listener callbacks. Exactly which Connex DDS APIs are restricted depends on the *Entity* upon which the *Listener* is installed, as well as the configuration of 'Exclusive Areas,' as discussed in [Exclusive Areas \(EAs\) \(Section 4.5 below\)](#).

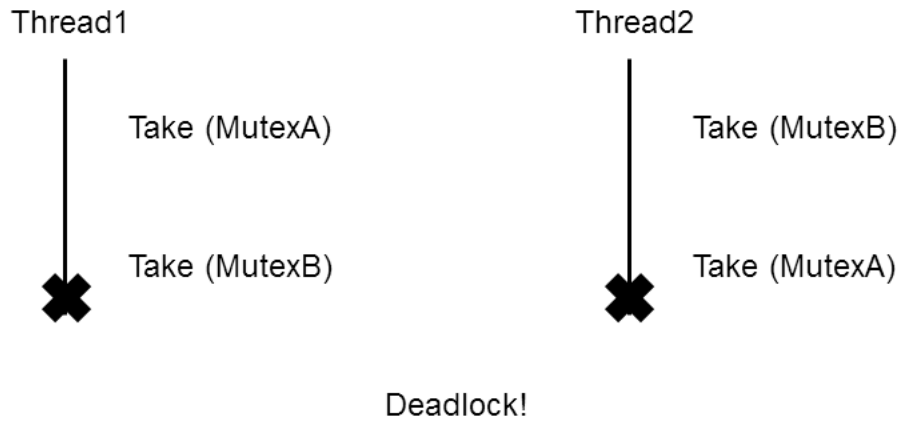
Please read and understand [Exclusive Areas \(EAs\) \(Section 4.5 below\)](#) and [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#) to ensure that the calls made from your *Listeners* are allowed and will not cause potential deadlock situations.

4.5 Exclusive Areas (EAs)

Listener callbacks are invoked by internal Connex DDS threads. To prevent undesirable, multi-threaded interaction, the internal threads may take and hold semaphores (mutexes) used for mutual exclusion. In your listener callbacks, you may want to invoke functions provided by the Connex DDS API. Internally, those Connex DDS functions also may take mutexes to prevent errors due to multi-threaded access to critical data or operations.

Once there are multiple mutexes to protect different critical regions, the possibility for deadlock exists. Consider [Figure 4.5 Multiple Mutexes Leading to a Deadlock Condition on the facing page](#)'s scenario, in which there are two threads and two mutexes.

Figure 4.5 Multiple Mutexes Leading to a Deadlock Condition



Thread1 takes MutexA while simultaneously Thread2 takes MutexB. Then, Thread1 takes MutexB and simultaneously Thread2 takes MutexA. Now both threads are blocked since they hold a mutex that the other thread is trying to take. This is a deadlock condition.

While the probability of entering the deadlock situation in [Figure 4.5 Multiple Mutexes Leading to a Deadlock Condition above](#) depends on execution timing, when there are multiple threads and multiple mutexes, care must be taken in writing code to prevent those situations from existing in the first place. Connex DDS has been carefully created and analyzed so that we know our threads internally are safe from deadlock interactions.

However, when Connex DDS threads that are holding mutexes call user code in listeners, it is possible for user code to inadvertently cause the threads to deadlock if Connex DDS APIs that try to take other mutexes are invoked. To help you avoid this situation, RTI has defined a concept known as *Exclusive Areas*, some restrictions regarding the use of Connex DDS APIs within user callback code, and a QoS policy that allows you to configure *Exclusive Areas*.

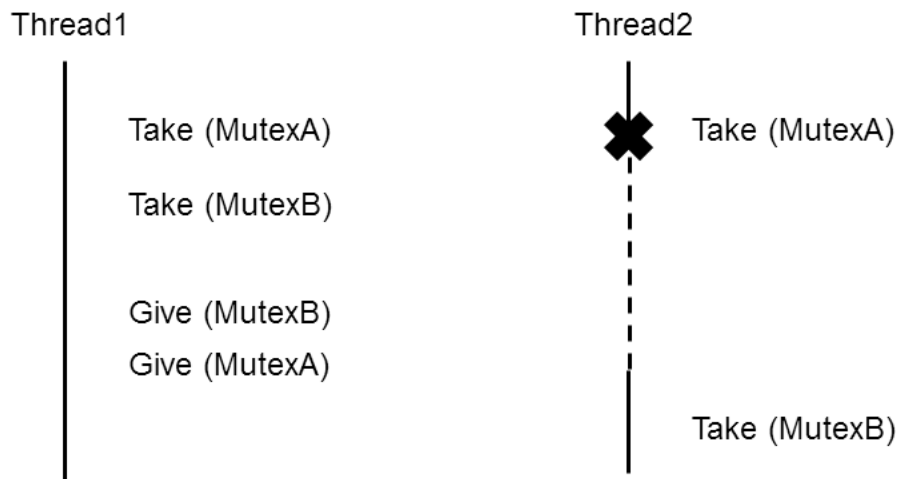
Connex DDS uses *Exclusive Areas* (EAs) to encapsulate mutexes and critical regions. Only one thread at a time can be executing code within an EA. The formal definition of EAs and their implementation ensures safety from deadlock and efficient entering and exiting of EAs. While every *Entity* created by Connex DDS has an associated EA, EAs may be shared among several *Entities*. A thread is automatically in the entity's EA when it is calling the entity's listener.

Connex DDS allows you to configure all the *Entities* within an application in a single DDS domain to share a single *Exclusive Area*. This would greatly restrict the concurrency of thread execution within Connex DDS's multi-threaded core. However, doing so would release all restrictions on using Connex DDS APIs within your callback code.

You may also have the best of both worlds by configuring a set of *Entities* to share a global EA and others to have their own. For the *Entities* that have their own EAs, the types of Connex DDS operations that you can call from the *Entity*'s callback are restricted.

To understand why the general EA framework limits the operations that can be called in an EA, consider a modification to the example previously presented in [Figure 4.5 Multiple Mutexes Leading to a Deadlock Condition on the previous page](#). Suppose we create a rule that is followed when we write our code. “For all situations in which a thread has to take multiple mutexes, we write our code so that the mutexes are always taken in the same order.” Following the rule will ensure us that the code we write cannot enter a deadlock situation due to the taking of the mutexes, see [Figure 4.6 Taking Multiple Mutexes in a Specific Order to Eliminate Deadlock below](#).

Figure 4.6 Taking Multiple Mutexes in a Specific Order to Eliminate Deadlock



By creating an order in which multiple mutexes are taken, you can guarantee that no deadlock situation will arise. In this case, if a thread must take both MutexA and MutexB, we write our code so that in those cases MutexA is always taken before MutexB.

Connex DDS defines an ordering of the mutexes it creates. Generally speaking, there are three ordered levels of Exclusive Areas:

- **ParticipantEA**

There is only one ParticipantEA per participant. The creation and deletion of all *Entities* (**create_XXX()**, **delete_XXX()**) take the ParticipantEA. In addition, the **enable()** method for an *Entity* and the setting of the *Entity*'s QoS, **set_qos()**, also take the ParticipantEA. There are other functions that take the ParticipantEA: **get_discovered_participants()**, **get_publishers()**, **get_subscribers()**, **get_**

`discovered_topics()`, `ignore_participant()`, `ignore_topic()`, `ignore_publication()`, `ignore_subscription()`, `remove_peer()`, and `register_type()`.

- **SubscriberEA**

This EA is created on a per-*Subscriber* basis by default. You can assume that the methods of a *Subscriber* will take the SubscriberEA. In addition, the *DataReaders* created by a *Subscriber* share the EA of its parent. This means that the methods of a *DataReader* (including `take()` and `read()`) will take the EA of its *Subscriber*. Therefore, **operations on DataReaders of the same Subscriber, will be serialized**, even when invoked from multiple concurrent application threads. As mentioned, the `enable()` and `set_qos()` methods of both *Subscribers* and *DataReaders* will take the ParticipantEA. The same is true for the `create_datareader()` and `delete_datareader()` methods of the *Subscriber*.

- **PublisherEA**

This EA is created on a per-*Publisher* basis by default. You can assume that the methods of a *Publisher* will take the PublisherEA. In addition, the *DataWriters* created by a *Publisher* share the EA of its parent. This means that the methods of a *DataWriter* including `write()` will take the EA of its *Publisher*. Therefore, **operations on DataWriters of the same Publisher will be serialized**, even when invoked from multiple concurrent application threads. As mentioned, the `enable()` and `set_qos()` methods of both *Publishers* and *DataWriters* will take the ParticipantEA, as well as the `create_datawriter()` and `delete_datawriter()` methods of the *Publisher*.

In addition, you should also be aware that:

- The three EA levels are ordered in the following manner:
ParticipantEA < SubscriberEA < PublisherEA
- When executing user code in a listener callback of an *Entity*, the internal Connex DDS thread is already in the EA of that *Entity* or used by that *Entity*.
- If a thread is in an EA, it can call methods associated with either a higher EA level or that share the *same* EA. It cannot call methods associated with a lower EA level *nor* ones that use a *different* EA at the same level.

4.5.1 Restricted Operations in Listener Callbacks

Based on the background and rules provided in [Exclusive Areas \(EAs\) \(Section 4.5 on page 182\)](#), this section describes how EAs restrict you from using various Connex DDS APIs from within the Listener callbacks of different *Entities*. Reader callbacks take the SubscriberEA. Writer callbacks take the PublisherEA. DomainParticipant callbacks take the ParticipantEA.

These restrictions do not apply to builtin topic listener callbacks.

By default, each *Publisher* and *Subscriber* creates and uses its own EA, and shares it with its children *DataWriters* and *DataReaders*, respectively. In that case:

Within a *DataWriter/DataReader's Listener* callback, do not:

- Create any *Entities*
- Delete any *Entities*
- Enable any *Entities*
- Set QoS on any *Entities*

Within a *Subscriber/DataReader's Listener* callback, do not call any operations on:

- Other *Subscribers*
- *DataReaders* that belong to other *Subscribers*
- *Publishers/DataWriters* that have been configured to use the ParticipantEA (see below)

Within a *Publisher/DataWriter Listener* callback, do not call any operations on:

- Other *Publishers*
- *DataWriters* that belong to other *Publishers*
- Any *Subscribers*
- Any *DataReaders*

Connex DDS will enforce the rules to avoid deadlock, and any attempt to call an illegal method from within a *Listener* callback will return `DDS_RETCODE_ILLEGAL_OPERATION`.

However, as previously mentioned, if you are willing to trade-off concurrency for flexibility, you may configure individual *Publishers* and *Subscribers* (and thus their *DataWriters* and *DataReaders*) to share the EA of their participant. In the limit, only a single ParticipantEA is shared among all *Entities*. When doing so, the restrictions above are lifted at a cost of greatly reduced concurrency. You may create/delete/enable/set_qos's and generally call all of the methods of any other entity in the Listener callbacks of *Entities* that share the ParticipantEA.

Use the [EXCLUSIVE_AREA QoS Policy \(DDS Extension\) \(Section 6.4.3 on page 318\)](#) of the *Publisher* or *Subscriber* to set whether or not to use a shared exclusive area. By default, *Publishers* and *Subscribers* will create and use their own individual EAs. You can configure a subset of the *Publishers* and *Subscribers* to share the ParticipantEA if you need the Listeners associated with those *Entities* or child *Entities* to be able to call any of the restricted methods listed above.

Regardless of how the `EXCLUSIVE_AREA` QosPolicy is set, the following operations are never allowed in any *Listener* callback:

- Destruction of the entity to which the *Listener* is attached. For instance, a *DataWriter/DataReader Listener* callback must not destroy its *DataWriter/DataReader*.
- Within the *TopicListener* callback, you cannot call any operations on *DataReaders*, *DataWriters*, *Publishers*, *Subscribers* or *DomainParticipants*.

4.6 Conditions and WaitSets

Conditions and *WaitSets* provide another way for Connex DDS to communicate status changes (including the arrival of data) to your application. While a *Listener* is used to provide a callback for asynchronous access, *Conditions* and *WaitSets* provide synchronous data access. In other words, *Listeners* are notification-based and *Conditions* are wait-based.

A *WaitSet* allows an application to wait until one or more attached *Conditions* becomes true (or until a timeout expires).

Briefly, your application can create a *WaitSet*, attach one or more *Conditions* to it, then call the *WaitSet*'s `wait()` operation. The `wait()` blocks until one or more of the *WaitSet*'s attached *Conditions* becomes `TRUE`.

A *Condition* has a `trigger_value` that can be `TRUE` or `FALSE`. You can retrieve the current value by calling the *Condition*'s only operation, `get_trigger_value()`.

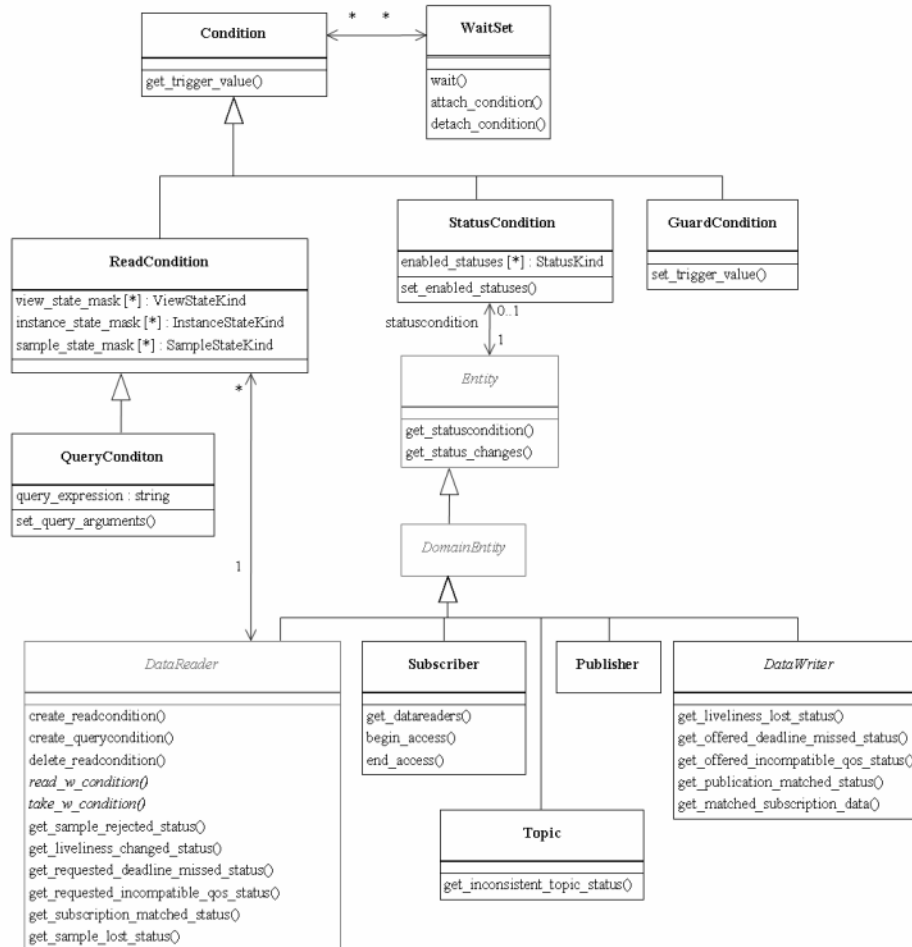
There are three kinds of *Conditions*. A *Condition* is a root class for all the conditions that may be attached to a *WaitSet*. This basic class is specialized in three classes:

- [GuardConditions \(Section 4.6.6 on page 194\)](#) are created by your application. Each *GuardCondition* has a single, user-settable, boolean `trigger_value`. Your application can manually trigger the *GuardCondition* by calling `set_trigger_value()`. Connex DDS does not trigger or clear this type of condition—it is completely controlled by your application.
- [ReadConditions and QueryConditions \(Section 4.6.7 on page 195\)](#) are created by your application, but triggered by Connex DDS. *ReadConditions* provide a way for you to specify the DDS data samples that you want to wait for, by indicating the desired sample-states, view-states, and instance-states¹.
- [StatusConditions \(Section 4.6.8 on page 197\)](#) are created automatically by Connex DDS, one for each *Entity*. A *StatusCondition* is triggered by Connex DDS when there is a change to any of that *Entity*'s enabled statuses.

¹These states are described in [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#).

Figure 4.7 Conditions and WaitSets below shows the relationship between these objects and other *Entities* in the system.

Figure 4.7 Conditions and WaitSets



A *WaitSet* can be associated with more than one *Entity* (including multiple *DomainParticipants*). It can be used to wait on *Conditions* associated with different *DomainParticipants*. A *WaitSet* can only be in use by one application thread at a time.

4.6.1 Creating and Deleting WaitSets

There is no factory for creating or deleting a *WaitSet*; use the natural means in each language binding (for example, “new” or “delete” in C++ or Java).

There are two ways to create a *WaitSet*—with or without specifying *WaitSet* properties (**DDS_**[WaitSetProperty_t](#), described in [Table 4.6 WaitSet Properties \(DDS_](#)[WaitSet_Property_t](#)[\)](#)). [Waiting for Conditions \(Section 4.6.3 on the next page\)](#) describes how the properties are used.

Table 4.6 WaitSet Properties (DDS_[WaitSet_Property_t](#)**)**

Type	Field Name	Description
long	max_event_count	Maximum number of trigger events to cause a <i>WaitSet</i> to wake up.
DDS_	max_	Maximum delay from occurrence of first trigger event to cause a <i>WaitSet</i> to wake up.
Duration_	event_	This value should reflect the maximum acceptable latency increase (time delay from occurrence of the event to waking up the <i>WaitSet</i>) incurred as a result of waiting for additional events before waking up the <i>WaitSet</i> .
t	delay	

To create a *WaitSet* with default behavior:

```
WaitSet* waitset = new WaitSet();
```

To create a *WaitSet* with properties:

```
DDS_
```

To delete a *WaitSet*:

```
delete waitset;
```

4.6.2 WaitSet Operations

WaitSets have only a few operations, as listed in [Table 4.7 WaitSet Operations](#). For details, see the API Reference HTML documentation.

Table 4.7 WaitSet Operations

Operation	Description
attach_condition	Attaches a <i>Condition</i> to this <i>WaitSet</i> . You may attach a <i>Condition</i> to a <i>WaitSet</i> that is currently being waited upon (via the <code>wait()</code> operation). In this case, if the <i>Condition</i> has a trigger_value of TRUE, then attaching the <i>Condition</i> will unblock the <i>WaitSet</i> . Adding a <i>Condition</i> that is already attached to the <i>WaitSet</i> has no effect. If the <i>Condition</i> cannot be attached, Connex DDS will return an OUT_OF_RESOURCES error code.
detach_condition	Detaches a <i>Condition</i> from the <i>WaitSet</i> . Attempting to detach a <i>Condition</i> that is not attached to the <i>WaitSet</i> will result in a PRECONDITION_NOT_MET error code.
wait	Blocks execution of the thread until one or more attached <i>Conditions</i> becomes true, or until a user-specified timeout expires. See Waiting for Conditions (Section 4.6.3 below) .
dispatch	(Modern C++ API only) Blocks execution of the thread until one or more attached <i>Conditions</i> becomes true, or until a user-specified timeout expires. Then it calls the handlers attached to the active conditions and returns. For more information see the API Reference HTML documentation for the DDS Modern C++ API (Modules, Infrastructure Module, Conditions and WaitSets).
get_conditions	Retrieves a list of attached <i>Conditions</i> .
get_property	Retrieves the DDS_WaitSetProperty_t structure of the associated <i>WaitSet</i> .
set_property	Sets the DDS_WaitSetProperty_t structure, to configure the associated <i>WaitSet</i> to return after one or more trigger events have occurred.

4.6.3 Waiting for Conditions

The *WaitSet*'s `wait()` operation allows an application thread to wait for any of the attached *Conditions* to trigger (become TRUE).

If any of the attached *Conditions* are already TRUE when `wait()` is called, it returns immediately.

If none of the attached *Conditions* are already TRUE, `wait()` blocks—suspending the calling thread. The waiting behavior depends on whether or not properties were set when the *WaitSet* was created:

- **If properties are not specified when the *WaitSet* is created:**

The *WaitSet* will wake up as soon as a trigger event occurs (that is, when an attached *Condition* becomes true). This is the default behavior if properties are not specified.

This 'immediate wake-up' behavior is optimal if you want to minimize latency (to wake up and process the data or event as soon as possible). However, "waking up" involves a context switch—the operating system must signal and schedule the thread that is waiting on the *WaitSet*. A context

switch consumes significant CPU and therefore waking up on each data update is not optimal in situations where the application needs to maximize throughput (the number of messages processed per second). This is especially true if the receiver is CPU limited.

- **If properties *are* specified when the *WaitSet* is created:**

The properties configure the waiting behavior of a *WaitSet*. If no conditions are true at the time of the call to wait, the *WaitSet* will wait for (a) **max_event_count** trigger events to occur, (b) up to **max_event_delay** time from the occurrence of the first trigger event, or (c) up to the timeout maximum wait duration specified in the call to **wait()**. (**Note:** The resolution of the timeout period is constrained by the resolution of the system clock.)

If **wait()** does not timeout, it returns a list of the attached *Conditions* that became TRUE and therefore unblocked the wait.

If **wait()** does timeout, it returns TIMEOUT and an empty list of *Conditions*.

Only one application thread can be waiting on the same *WaitSet*. If **wait()** is called on a *WaitSet* that already has a thread blocking on it, the operation will immediately return PRECONDITION_NOT_MET.

If you detach a *Condition* from a *Waitset* that is currently in a wait state (that is, you are waiting on it), **wait()** may return OK *and* an empty sequence of conditions.

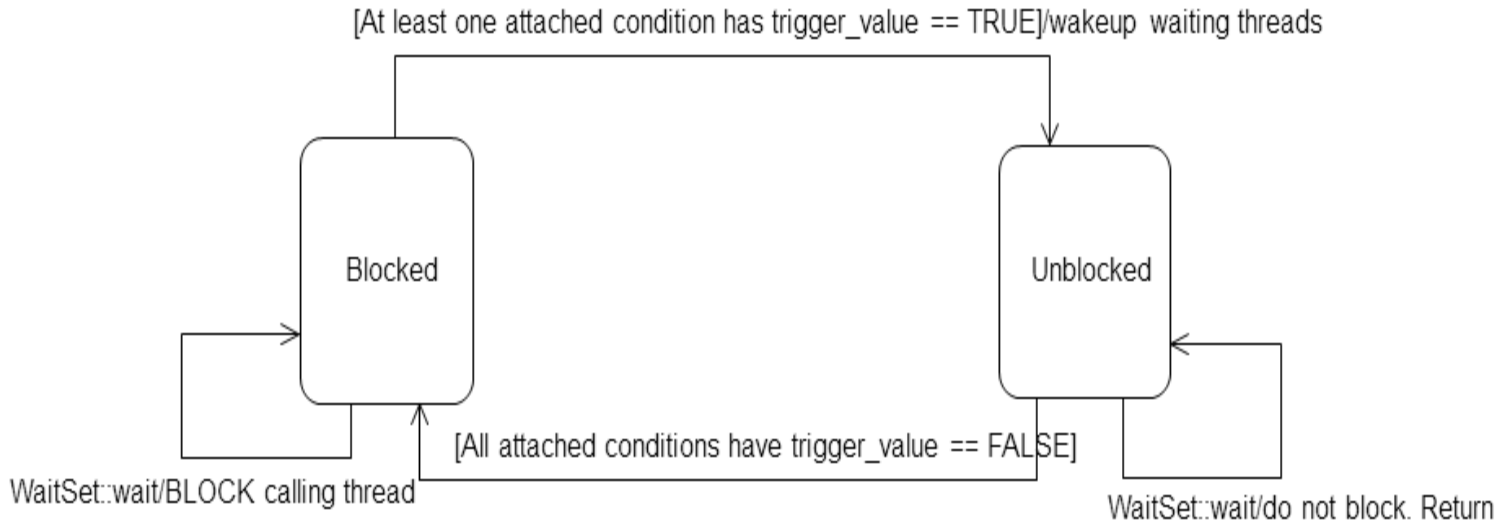
4.6.3.1 How WaitSets Block

The blocking behavior of the *WaitSet* is illustrated in [Figure 4.8 WaitSet Blocking Behavior on the next page](#). The result of a **wait()** operation depends on the state of the *WaitSet*, which in turn depends on whether at least one attached *Condition* has a **trigger_value** of TRUE.

If the **wait()** operation is called on a *WaitSet* with state BLOCKED, it will block the calling thread. If **wait()** is called on a *WaitSet* with state UNBLOCKED, it will return immediately.

When the *WaitSet* transitions from BLOCKED to UNBLOCKED, it wakes up the thread (if there is one) that had called **wait()** on it. There is no implied “event queuing” in the awakening of a *WaitSet*. That is, if several *Conditions* attached to the *WaitSet* have their **trigger_value** transition to true in sequence, Connex DDS will only unblock the *WaitSet* once.

Figure 4.8 WaitSet Blocking Behavior



4.6.4 Processing Triggered Conditions—What to do when Wait() Returns

When **wait()** returns, it provides a list of the attached *Condition* objects that have a **trigger_value** of true. Your application can use this list to do the following for each *Condition* in the returned list:

- If it is a *StatusCondition*:
 - First, call **get_status_changes()** to see what status changed.
 - If the status changes refer to plain communication status: call **get_<communication_status>()** on the relevant *Entity*.
 - If the status changes refer to DATA_ON_READERS¹: call **get_datareaders()** on the relevant *Subscriber*.
 - If the status changes refer to DATA_AVAILABLE: call **read()** or **take()** on the relevant *DataReader*.
- If it is a *ReadCondition* or a *QueryCondition*: You may want to call **read_w_condition()** or **take_w_condition()** on the *DataReader*, with the *ReadCondition* as a parameter (see [read_w_condition](#) and [take_w_condition](#) (Section 7.4.3.6 on page 500)).

¹And then read/take on the returned *DataReader* objects.

Note that this is just a suggestion, you do not have to use the “w_condition” operations (or any read/take operations, for that matter) simply because you used a *WaitSet*. The “w_condition” operations are just a convenient way to use the same status masks that were set on the *ReadCondition* or *QueryCondition*.

- If it is a *GuardCondition*: check to see which *GuardCondition* changed, then react accordingly. Recall that *GuardConditions* are completely controlled by your application.

See [Conditions and WaitSet Example \(Section 4.6.5 below\)](#) to see how to determine which of the attached *Conditions* is in the returned list.

4.6.5 Conditions and WaitSet Example

This example creates a *WaitSet* and then waits for one or more attached *Conditions* to become true.

```
// Create a WaitSet
WaitSet* waitset = new WaitSet();
// Attach Conditions
DDSCondition* cond1 = ...;
DDSCondition* cond2 = entity->get_statuscondition();
DDSCondition* cond3 = reader->create_readcondition(
    DDS_NOT_READ_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
DDSCondition* cond4 = new DDSGuardCondition();
DDSCondition* cond5 = ...;
DDS_ReturnCode_t retcode;

retcode = waitset->attach_condition(cond1);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond2);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond3);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond4);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond5);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
// Wait for a condition to trigger or timeout
DDS_Duration_t timeout = { 0, 1000000 }; // 1ms
DDSConditionSeq active_conditions; // holder for active conditions
bool is_cond1_triggered = false;
```

```

bool is_cond2_triggered = false;
DDS_ReturnCode_t retcode;

retcode = waitset->wait(active_conditions, timeout);
if (retcode == DDS_RETCODE_TIMEOUT) {
    // handle timeout
    printf("Wait timed out. No conditions were triggered.\n");
}
else if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
} else {
    // success
    if (active_conditions.length() == 0) {
        printf("Wait timed out!! No conditions triggered.\n");
    } else
        // check if "cond1" or "cond2" are triggered:
        for(i = 0; i < active_conditions.length(); ++i) {
            if (active_conditions[i] == cond1) {
                printf("Cond1 was triggered!");
                is_cond1_triggered = true;
            }
            if (active_conditions[i] == cond2) {
                printf("Cond2 was triggered!");
                is_cond2_triggered = true;
            }
            if (is_cond1_triggered && is_cond2_triggered) {
                break;
            }
        }
}
}

if (is_cond1_triggered) {
    // ... do something because "cond1" was triggered ...
}
if (is_cond2_triggered) {
    // ... do something because "cond2" was triggered ...
}
// Delete the waitset
delete waitset;
waitset = NULL;

```

4.6.6 GuardConditions

GuardConditions are created by your application. *GuardConditions* provide a way for your application to manually awaken a *WaitSet*. Like all *Conditions*, it has a single boolean **trigger_value**. Your application can manually trigger the *GuardCondition* by calling **set_trigger_value()**.

Connex DDS does not trigger or clear this type of condition—it is completely controlled by your application.

A *GuardCondition* has no factory. It is created as an object directly by the natural means in each language binding (e.g., using “new” in C++ or Java). For example:

```
// Create a Guard Condition
Condition* my_guard_condition = new GuardCondition();
// Delete a Guard Condition
delete my_guard_condition;
```

When first created, the **trigger_value** is FALSE.

A `GuardCondition` has only two operations, **get_trigger_value()** and **set_trigger_value()**.

When your application calls **set_trigger_value(DDS_BOOLEAN_TRUE)**, Connex DDS will awaken any *WaitSet* to which the *GuardCondition* is attached.

4.6.7 ReadConditions and QueryConditions

ReadConditions are created by your application, but triggered by Connex DDS. *ReadConditions* provide a way for you to specify the DDS data samples that you want to wait for, by indicating the desired sample-states, view-states, and instance-states¹. Then Connex DDS will trigger the *ReadCondition* when suitable DDS samples are available.

A *QueryCondition* is a special *ReadCondition* that allows you to specify a query expression and parameters, so you can filter on the locally available (already received) data. *QueryConditions* use the same SQL-based filtering syntax as `ContentFilteredTopics` for query expressions, parameters, etc. Unlike `ContentFilteredTopics`, *QueryConditions* are applied to data already received, so they do not affect the reception of data.

Multiple mask combinations can be associated with a single content filter. This is important because the maximum number of content filters that may be created per *DataReader* is 32, but more than 32 *QueryConditions* may be created per *DataReader*, if they are different mask-combinations of the same content filter.

ReadConditions and *QueryConditions* are created by using the *DataReader*'s **create_readcondition()** and **create_querycondition()** operations. For example:

```
DDSReadCondition* my_read_condition = reader->create_readcondition(
    DDS_NOT_READ_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);

DDSQueryCondition* my_query_condition = reader->create_querycondition(
    DDS_NOT_READ_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE,
    query_expression,
    query_parameters);
```

¹These states are described in [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#).

If you are using a *ReadCondition* to simply detect the presence of new data, consider using a *StatusCondition* ([StatusConditions \(Section 4.6.8 on the facing page\)](#)) with the `DATA_AVAILABLE_STATUS` instead, which will perform better in this situation.

A *DataReader* can have multiple attached *ReadConditions* and *QueryConditions*. A *ReadCondition* or *QueryCondition* may only be attached to one *DataReader*.

To delete a *ReadCondition* or *QueryCondition*, use the *DataReader*'s `delete_readcondition()` operation:

```
DDS_ReturnCode_t delete_readcondition (DDSReadCondition *condition)
```

After a *ReadCondition* is triggered, use the *FooDataReader*'s read/take “with condition” operations (see [read_w_condition](#) and [take_w_condition \(Section 7.4.3.6 on page 500\)](#)) to access the DDS samples.

[ReadCondition and QueryCondition Operations \(Section Table 4.8 below\)](#) lists the operations available on *ReadConditions*.

Table 4.8 ReadCondition and QueryCondition Operations

Operation	Description
<code>get_datareader</code>	Returns the <i>DataReader</i> to which the <i>ReadCondition</i> or <i>QueryCondition</i> is attached.
<code>get_instance_state_mask</code>	Returns the instance states that were specified when the <i>ReadCondition</i> or <i>QueryCondition</i> was created. These are the DDS sample's instance states that Connex DDS checks to determine whether or not to trigger the <i>ReadCondition</i> or <i>QueryCondition</i> .
<code>get_sample_state_mask</code>	Returns the sample-states that were specified when the <i>ReadCondition</i> or <i>QueryCondition</i> was created. These are the sample states that Connex DDS checks to determine whether or not to trigger the <i>ReadCondition</i> or <i>QueryCondition</i> .
<code>get_view_state_mask</code>	Returns the view-states that were specified when the <i>ReadCondition</i> or <i>QueryCondition</i> was created. These are the view states that Connex DDS checks to determine whether or not to trigger the <i>ReadCondition</i> or <i>QueryCondition</i> .

4.6.7.1 How ReadConditions are Triggered

A *ReadCondition* has a **trigger_value** that determines whether the attached *WaitSet* is BLOCKED or UNBLOCKED. Unlike the *StatusCondition*, the **trigger_value** of the *ReadCondition* is tied to the presence of at least one DDS sample with a sample-state, view-state, and instance-state that matches those set in the *ReadCondition*. Furthermore, for the *QueryCondition* to have a **trigger_value**==TRUE, the data associated with the DDS sample must be such that the **query_expression** evaluates to TRUE.

The **trigger_value** of a *ReadCondition* depends on the presence of DDS samples on the associated *DataReader*. This implies that a single ‘take’ operation can potentially change the **trigger_value** of several *ReadConditions* or *QueryConditions*. For example, if all DDS samples are taken, any *ReadConditions* and *QueryConditions* associated with the *DataReader* that had **trigger_value**==TRUE before will see the **trigger_value** change to FALSE. Note that this does not guarantee that *WaitSet* objects that were separately

attached to those conditions will not be awakened. Once we have **trigger_value**==TRUE on a condition, it may wake up the attached *WaitSet*, the condition transitioning to **trigger_value**==FALSE does not necessarily 'unwake' the *WaitSet*, since 'unwakening' may not be possible. The consequence is that an application blocked on a *WaitSet* may return from **wait()** with a list of conditions, some of which are no longer "active." This is unavoidable if multiple threads are concurrently waiting on separate *WaitSet* objects and taking data associated with the same *DataReader*.

Consider the following example: A *ReadCondition* that has a `sample_state_mask = {NOT_READ}` will have a **trigger_value** of TRUE whenever a new DDS sample arrives and will transition to FALSE as soon as all the newly arrived DDS samples are either read (so their status changes to READ) or taken (so they are no longer managed by Connex DDS). However, if the same *ReadCondition* had a **sample_state_mask = {READ, NOT_READ}**, then the **trigger_value** would only become FALSE once all the newly arrived DDS samples are *taken* (it is not sufficient to just *read* them, since that would only change the *SampleState* to READ), which overlaps the mask on the *ReadCondition*.

4.6.7.2 QueryConditions

A *QueryCondition* is a special *ReadCondition* that allows your application to also specify a filter on the locally available data.

The query expression is similar to a SQL WHERE clause and can be parameterized by arguments that are dynamically changeable by the **set_query_parameters()** operation.

QueryConditions are triggered in the same manner as *ReadConditions*, with the additional requirement that the DDS sample must also satisfy the conditions of the content filter associated with the *QueryCondition*.

Table 4.9 QueryCondition Operations

Operation	Description
get_query_expression	Returns the query expression specified when the <i>QueryCondition</i> was created.
get_query_parameters	Returns the query parameters associated with the <i>QueryCondition</i> . That is, the parameters specified on the last successful call to set_query_parameters() , or if set_query_parameters() was never called, the arguments specified when the <i>QueryCondition</i> was created.
set_query_parameters	Changes the query parameters associated with the <i>QueryCondition</i> .

4.6.8 StatusConditions

StatusConditions are created automatically by Connex DDS, one for each *Entity*. Connex DDS will trigger the *StatusCondition* when there is a change to any of that *Entity*'s enabled statuses.

By default, when Connex DDS creates a *StatusCondition*, all status bits are turned on, which means it will check for all statuses to determine when to trigger the *StatusCondition*. If you only want Connex DDS to check for specific statuses, you can use the *StatusCondition*'s `set_enabled_statuses()` operation and set just the desired status bits.

The **trigger_value** of the *StatusCondition* depends on the communication status of the *Entity* (e.g., arrival of data, loss of information, etc.), 'filtered' by the set of enabled statuses on the *StatusCondition*.

The set of enabled statuses and its relation to *Listeners* and *WaitSets* is detailed in [How StatusConditions are Triggered \(Section 4.6.8.1 on the facing page\)](#).

[Table 4.10 StatusCondition Operations](#) lists the operations available on *StatusConditions*.

Table 4.10 StatusCondition Operations

Operation	Description
set_enabled_statuses	<p>Defines the list of communication statuses that are taken into account to determine the trigger_value of the <i>StatusCondition</i>. This operation may change the trigger_value of the <i>StatusCondition</i>.</p> <p><i>WaitSets</i> behavior depend on the changes of the trigger_value of their attached conditions. Therefore, any <i>WaitSet</i> to which the <i>StatusCondition</i> is attached is potentially affected by this operation.</p> <p>If this function is not invoked, the default list of enabled statuses includes all the statuses.</p>
get_enabled_statuses	<p>Retrieves the list of communication statuses that are taken into account to determine the trigger_value of the <i>StatusCondition</i>. This operation returns the statuses that were explicitly set on the last call to <code>set_enabled_statuses()</code> or, if <code>set_enabled_statuses()</code> was never called, the default list</p>
get_entity	<p>Returns the <i>Entity</i> associated with the <i>StatusCondition</i>. Note that there is exactly one <i>Entity</i> associated with each <i>StatusCondition</i>.</p>

Unlike other types of *Conditions*, *StatusConditions* are created by Connex DDS, not by your application. To access an *Entity*'s *StatusCondition*, use the *Entity*'s `get_statuscondition()` operation. For example:

```
Condition* my_status_condition = entity->get_statuscondition();
```

In the Modern C++ API, use the *StatusCondition* constructor to obtain a reference to the *Entity*'s condition. For example:

```
dds::core::cond::StatusCondition my_status_condition(entity)
```

After a *StatusCondition* is triggered, call the *Entity*'s `get_status_changes()` operation to see which status (es) changed.

Note: Not all statuses will activate the *StatusCondition*. Refer to the API Reference HTML documentation of the individual statuses for that information.

4.6.8.1 How StatusConditions are Triggered

The **trigger_value** of a *StatusCondition* is the boolean OR of the **ChangedStatusFlag** of all the communication statuses to which it is sensitive. That is, **trigger_value** is FALSE only if *all* the values of the **ChangedStatusFlags** are FALSE.

The sensitivity of the *StatusCondition* to a particular communication status is controlled by the list of **enabled_statuses** set on the *Condition* by means of the **set_enabled_statuses()** operation.

Once a *StatusCondition*'s **trigger_value** becomes true, it remains true until the status that changed is reset. To reset a status, call the related **get_*_status()** operation. Or, in the case of the data available status, call **read()**, **take()**, or one of their variants.

Therefore, if you are using a *StatusCondition* on a *WaitSet* to be notified of events, your thread will wake up when one of the statuses associated with the *StatusCondition* becomes true. If you do not reset the status, the *StatusCondition*'s **trigger_value** remains true and your *WaitSet* will not block again—it will immediately wake up when you call **wait()**.

4.6.9 Using Both Listeners and WaitSets

You can use *Listeners* and *WaitSets* in the same application. For example, you may want to use *WaitSets* and *Conditions* to access the data, and *Listeners* to be warned asynchronously of erroneous communication statuses.

We recommend that you choose one or the other mechanism for each particular communication status (not both). However, if both are enabled, the *Listener* mechanism is used first, then the *WaitSet* objects are signaled.

Chapter 5 Topics

For a *DataWriter* and *DataReader* to communicate, they need to use the same *Topic*. A *Topic* includes a name and an association with a user data type that has been registered with Connex DDS. Topic names are how different parts of the communication system find each other. *Topics* are named streams of data of the same data type. *DataWriters* publish DDS samples into the stream; *DataReaders* subscribe to data from the stream. More than *one* *Topic* can use the same user data type, but each *Topic* needs a unique name.

Topics, *DataWriters*, and *DataReaders* relate to each other as follows:

- Multiple *Topics* (each with a unique name) can use the same user data type.
- Applications may have multiple *DataWriters* for each *Topic*.
- Applications may have multiple *DataReaders* for each *Topic*.
- *DataWriters* and *DataReaders* must be associated with the same *Topic* in order for them to be connected.
- *Topics* are created and deleted by a *DomainParticipant*, and as such, are owned by that *DomainParticipant*. When two applications (*DomainParticipants*) want to use the same *Topic*, they must both create the *Topic* (even if the applications are on the same node).

Connex DDS uses ‘Builtin Topics’ to discover and keep track of remote entities, such as new participants in the DDS domain. Builtin Topics are discussed in [Built-In Topics \(Section Chapter 16 on page 772\)](#).

This section includes the following sections:

5.1 Topics

Before you can create a *Topic*, you need a user data type (see [Data Types and DDS Data Samples \(Section Chapter 3 on page 23\)](#)) and a *DomainParticipant* ([DomainParticipants \(Section 8.3 on](#)

page 547)). The user data type must be registered with the *DomainParticipant* (see [Type Codes for Built-in Types \(Section 3.8.4.1 on page 143\)](#)).

Once you have created a *Topic*, what do you do with it? Topics are primarily used as parameters in other *Entities*' operations. For instance, a *Topic* is required when a *Publisher* or *Subscriber* creates a *DataWriter* or *DataReader*, respectively. *Topics* do have a few operations of their own, as listed in [Table 5.1 Topic Operations](#). For details on using these operations, see the reference section or the API Reference HTML documentation.

Figure 5.1 Topic Module

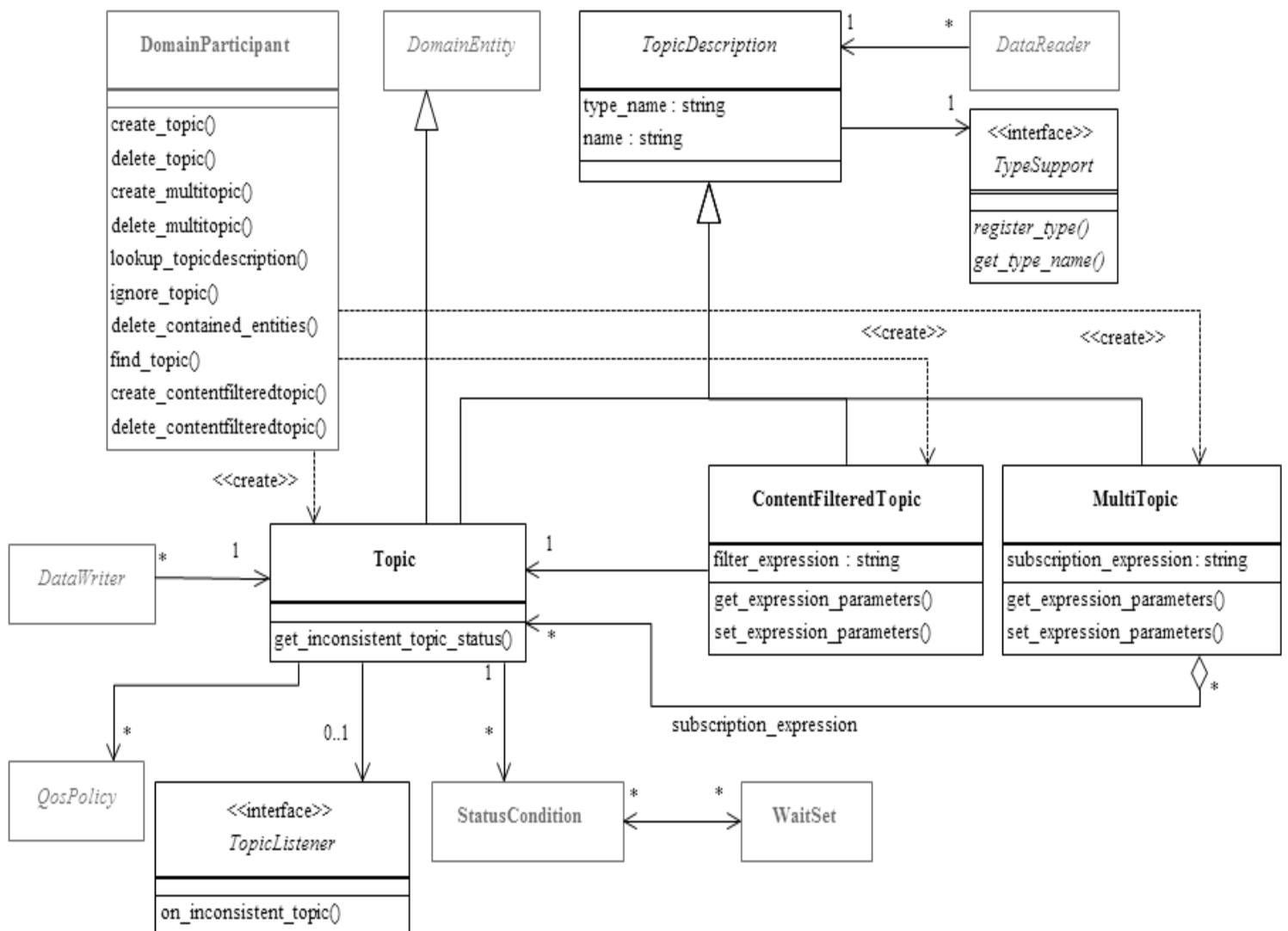


Table 5.1 Topic Operations

Purpose	Operation	Description	Reference
Configuring the Topic	enable	Enables the <i>Topic</i> .	Enabling DDS Entities (Section 4.1.2 on page 154)
	get_qos	Gets the <i>Topic</i> 's current QoSPolicy settings. This is most often used in preparation for calling <code>set_qos()</code> .	Setting Topic QoS Policies (Section 5.1.3 on page 204)
	set_qos	Sets the <i>Topic</i> 's QoS. You can use this operation to change the values for the <i>Topic</i> 's QoS Policies. Note, however, that not all QoS Policies can be changed after the <i>Topic</i> has been created.	
	equals	Compares two <i>Topic</i> 's QoS structures for equality.	Comparing QoS Values (Section 5.1.3.2 on page 207)
	set_qos_with_profile	Sets the <i>Topic</i> 's QoS based on a specified QoS profile.	
	get_listener	Gets the currently installed Listener.	Setting Up TopicListeners (Section 5.1.5 on page 208)
	set_listener	Sets the <i>Topic</i> 's Listener. If you create the <i>Topic</i> without a Listener, you can use this operation to add one later. Setting the listener to NULL will remove the listener from the <i>Topic</i> .	
	narrow	A type-safe way to cast a pointer. This takes a DDS TopicDescription pointer and 'narrows' it to a DDS Topic pointer.	Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.7 on page 281)
Checking Status	get_inconsistent_topic_status	Allows an application to retrieve a <i>Topic</i> 's INCONSISTENT_TOPIC_STATUS status.	INCONSISTENT_TOPIC Status (Section 5.3.1 on page 211)
	get_status_changes	Gets a list of statuses that have changed since the last time the application read the status or the listeners were called.	Getting Status and Status Changes (Section 4.1.4 on page 157)
Navigating Relationships	get_name	Gets the <i>topic_name</i> string used to create the <i>Topic</i> .	Creating Topics (Section 5.1.1 below)
	get_type_name	Gets the <i>type_name</i> used to create the <i>Topic</i> .	
	get_participant	Gets the <i>DomainParticipant</i> to which this <i>Topic</i> belongs.	Finding a Topic's DomainParticipant (Section 5.1.6.1 on page 209)

5.1.1 Creating Topics

Topics are created using the *DomainParticipant*'s `create_topic()` or `create_topic_with_profile()` operation.

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

```
DDSTopic * create_topic (
    const char *topic_name,
    const char *type_name,
    const DDS_TopicQos &qos,
    DDSTopicListener *listener,
    DDS_StatusMask mask)
DDSTopic * create_topic_with_profile (
    const char *topic_name,
    const char *type_name,
    const char *library_name,
    const char *profile_name,
    DDSTopicListener *listener,
    DDS_StatusMask mask)
```

Where:

- topic_name** Name for the new *Topic*, must not exceed 255 characters.
- type_name** Name for the user data type, must not exceed 255 characters. It must be the same name that was used to register the DDS type, and the DDS type must be registered with the same *DomainParticipant* used to create this *Topic*. See [Using RTI Code Generator \(rtiddsgen\) \(Section 3.6 on page 138\)](#).
- qos** If you want to use the default QoS settings (described in the API Reference HTML documentation), use `DDS_TOPIC_QOS_DEFAULT` for this parameter (see [Figure 5.2 Creating a Topic with Default QoS Policies on the facing page](#)). If you want to customize any of the QoS Policies, supply a QoS structure (see [Setting Topic QoS Policies \(Section 5.1.3 on the facing page\)](#)).

If you use `DDS_TOPIC_QOS_DEFAULT`, it is *not* safe to create the topic while another thread may be simultaneously calling the *DomainParticipant's* `set_default_topic_qos()` operation.
- listener** *Listeners* are callback routines. Connex DDS uses them to notify your application of specific events (status changes) that may occur with respect to the *Topic*. The *listener* parameter may be set to NULL if you do not want to install a *Listener*. If you use NULL, the *Listener* of the *DomainParticipant* to which the *Topic* belongs will be used instead (if it is set). For more information on *TopicListeners*, see [Setting Up TopicListeners \(Section 5.1.5 on page 208\)](#).
- mask** This bit-mask indicates which status changes will cause the *Listener* to be invoked. The bits in the mask that are set must have corresponding callbacks implemented in the *Listener*. If you use NULL for the *Listener*, use `DDS_STATUS_MASK_NONE` for this parameter. If the *Listener* implements all callbacks, use `DDS_STATUS_MASK_ALL`. For information on statuses, see [Listeners \(Section 4.4 on page 177\)](#).

library_name A QoS Library is a named set of QoS profiles. See [URL Groups \(Section 17.8 on page 814\)](#). If NULL is used for **library_name**, the *DomainParticipant*'s default library is assumed.

profile_name A QoS profile groups a set of related QoS, usually one per entity. See [URL Groups \(Section 17.8 on page 814\)](#). If NULL is used for **profile_name**, the *DomainParticipant*'s default profile is assumed and **library_name** is ignored.

It is not safe to create a topic while another thread is calling **lookup_topicdescription()** for that same topic (see [Looking up Topic Descriptions \(Section 8.3.7 on page 568\)](#)).

Figure 5.2 Creating a Topic with Default QoS Policies

```
const char *type_name = NULL;
// register the DDS type
type_name = FooTypeSupport::get_type_name();
retcode = FooTypeSupport::register_type(
    participant, type_name);
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// create the topic
DDSTopic* topic = participant->create_topic(
    "Example Foo", type_name,
    DDS_TOPIC_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
if (topic == NULL) {
    // process error here
};
```

For more examples, see [Configuring QoS Settings when the Topic is Created \(Section 5.1.3.1 on page 206\)](#).

5.1.2 Deleting Topics

To delete a Topic, use the *DomainParticipant*'s **delete_topic()** operation:

```
DDS_ReturnCode_t delete_topic (DDSTopic * topic)
```

Note, however, that you cannot delete a Topic if there are any existing *DataReaders* or *DataWriters* (belonging to the same *DomainParticipant*) that are still using it. All *DataReaders* and *DataWriters* associated with the *Topic* must be deleted first.

Note: in the Modern C++ API, *Entities* are automatically destroyed.

5.1.3 Setting Topic QoS Policies

A *Topic*'s QoS Policies control its behavior, or more specifically, the behavior of the *DataWriters* and *DataReaders* of the *Topic*. You can think of the policies as the 'properties' for the *Topic*. The **DDS_**

TopicQos structure has the following format:

```
DDS_TopicQos struct {
    DDS_TopicDataQosPolicy      topic_data;
    DDS_DurabilityQosPolicy     durability;
    DDS_DurabilityServiceQosPolicy durability_service;
    DDS_DeadlineQosPolicy       deadline;
    DDS_LatencyBudgetQosPolicy  latency_budget;
    DDS_LivelinessQosPolicy     liveliness;
    DDS_ReliabilityQosPolicy    reliability;
    DDS_DestinationOrderQosPolicy destination_order;
    DDS_HistoryQosPolicy        history;
    DDS_ResourceLimitsQosPolicy resource_limits;
    DDS_TransportPriorityQosPolicy transport_priority;
    DDS_LifespanQosPolicy       lifespan;
    DDS_OwnershipQosPolicy      ownership;
} DDS_TopicQos;
```

Table 5.2 Topic QoS Policies summarizes the meaning of each policy (arranged alphabetically). For information on *why* you would want to change a particular QoS Policy, see the section noted in the **Reference** column. For defaults and valid ranges, please refer to the API Reference HTML documentation for each policy.

Table 5.2 Topic QoS Policies

QoS Policy	Description
Deadline	For a <i>DataReader</i> , specifies the maximum expected elapsed time between arriving DDS data samples. For a <i>DataWriter</i> , specifies a commitment to publish DDS samples with no greater elapsed time between them. See DEADLINE QoS Policy (Section 6.5.5 on page 363) .
DestinationOrder	Controls how Connex DDS will deal with data sent by multiple <i>DataWriters</i> for the same topic. Can be set to "by reception timestamp" or "by source timestamp". See DESTINATION_ORDER QoS Policy (Section 6.5.6 on page 365) .
Durability	Specifies whether or not Connex DDS will store and deliver data that were previously published to new <i>DataReaders</i> . See DURABILITY QoS Policy (Section 6.5.7 on page 368) .
DurabilityService	Various settings to configure the external Persistence Service used by Connex DDS for <i>DataWriters</i> with a Durability QoS setting of Persistent Durability. See DURABILITY SERVICE QoS Policy (Section 6.5.8 on page 372) .
History	Specifies how much data must be stored by Connex DDS for the <i>DataWriter</i> or <i>DataReader</i> . This QoS Policy affects the RELIABILITY QoS Policy (Section 6.5.19 on page 400) as well as the DURABILITY QoS Policy (Section 6.5.7 on page 368) . See HISTORY QoS Policy (Section 6.5.10 on page 376) .
LatencyBudget	Suggestion to Connex DDS on how much time is allowed to deliver data. See LATENCYBUDGET QoS Policy (Section 6.5.11 on page 380) .

Table 5.2 Topic QoS Policies

QoS Policy	Description
Lifespan	Specifies how long Connex DDS should consider data sent by an user application to be valid. See LIFESPAN QoS Policy (Section 6.5.12 on page 381) .
Liveliness	Specifies and configures the mechanism that allows <i>DataReaders</i> to detect when <i>DataWriters</i> become disconnected or "dead." See LIVELINESS QoS Policy (Section 6.5.13 on page 382) .
Ownership	Along with Ownership Strength, specifies if <i>DataReaders</i> for a topic can receive data from multiple <i>DataWriters</i> at the same time. See OWNERSHIP QoS Policy (Section 6.5.15 on page 389) .
Reliability	Specifies whether or not Connex DDS will deliver data reliably. See RELIABILITY QoS Policy (Section 6.5.19 on page 400) .
ResourceLimits	Controls the amount of physical memory allocated for entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See RESOURCE_LIMITS QoS Policy (Section 6.5.20 on page 405) .
TopicData	Along with Group Data QoS Policy and User Data QoS Policy, used to attach a buffer of bytes to Connex DDS's discovery meta-data. See TOPIC_DATA QoS Policy (Section 5.2.1 on page 209) .
TransportPriority	Set by a <i>DataWriter</i> to tell Connex DDS that the data being sent is a different "priority" than other data. See TRANSPORT_PRIORITY QoS Policy (Section 6.5.22 on page 409) .

5.1.3.1 Configuring QoS Settings when the Topic is Created

As described in [Creating Topics \(Section 5.1.1 on page 202\)](#), there are different ways to create a Topic, depending on how you want to specify its QoS (with or without a QoS profile).

In [Creating a Topic with Default QoS Policies \(Section Figure 5.2 on page 204\)](#), we saw an example of how to create a Topic with default QoS Policies by using the special constant, `DDS_TOPIC_QOS_DEFAULT`, which indicates that the default QoS values for a *Topic* should be used. The default Topic QoS values are configured in the *DomainParticipant*; you can change them with the *DomainParticipant*'s `set_default_topic_qos()` or `set_default_topic_qos_with_profile()` operations (see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on page 568\)](#)).

To create a Topic with non-default QoS values, without using a QoS profile, use the *DomainParticipant*'s `get_default_topic_qos()` operation to initialize a `DDS_TopicQos` structure. Then change the policies from their default values before passing the QoS structure to `create_topic()`.

You can also create a *Topic* and specify its QoS settings via a QoS profile. To do so, call `create_topic_with_profile()`.

If you want to use a QoS profile, but then make some changes to the QoS before creating the Topic, call `get_topic_qos_from_profile()`, modify the QoS and use the modified QoS when calling `create_topic()`.

5.1.3.2 Comparing QoS Values

The `equals()` operation compares two *Topic*'s `DDS_TopicQoS` structures for equality. It takes two parameters for the two *Topics*' QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

5.1.3.3 Changing QoS Settings After the Topic Has Been Created

There are two ways to change an existing *Topic*'s QoS after it has been created—again depending on whether or not you are using a QoS Profile.

To change QoS programmatically (that is, without using a QoS Profile), see the example code in [Figure 5.3 Changing the QoS of an Existing Topic \(without a QoS Profile\)](#) below. It retrieves the current values by calling the *Topic*'s `get_qos()` operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QoS Policies cannot be changed after the *Topic* has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.

You can also change a *Topic*'s (and all other Entities') QoS by using a QoS Profile. For an example, see [Figure 5.4 Changing the QoS of an Existing Topic with a QoS Profile](#) below. For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 5.3 Changing the QoS of an Existing Topic (without a QoS Profile)

```
DDS_TopicQos topic_qos;1
// Get current QoS. topic points to an existing DDS_Topic.
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Next, make changes.
// New ownership kind will be Exclusive
topic_qos.ownership.kind = DDS_EXCLUSIVE_OWNERSHIP_QOS;
// Set the new QoS
if (topic->set_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

Figure 5.4 Changing the QoS of an Existing Topic with a QoS Profile

```
retcode = topic->set_qos_with_profile(
    "FooProfileLibrary", "FooProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

¹For the C API, use `DDS_TopicQos_INITIALIZER` or `DDS_TopicQos_initialize()`. See [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#)

5.1.4 Copying QoS From a Topic to a DataWriter or DataReader

Only the `TOPIC_DATA` QoSPolicy strictly applies to *Topics*—it is described in this section, while the others are described in the sections noted [Table 5.2 Topic QoS Policies](#). The rest of the QoS Policies for a *Topic* can also be set on the corresponding *DataWriters* and/or *DataReaders*. Actually, the values that Connex DDS uses for those policies are taken directly from those set on the *DataWriters* and *DataReaders*. The values for those policies are stored only for reference in the `DDS_TopicQos` structure.

Because many QoS Policies affect the behavior of matching *DataWriters* and *DataReaders*, the `DDS_TopicQos` structure is provided as a convenient way to set the values for those policies in a single place in the application. Otherwise, you would have to modify the individual QoS Policies within separate *DataWriter* and *DataReader* QoS structures. And because some QoS Policies are compared between *DataReaders* and *DataWriters*, you will need to make certain that the individual values that you set are compatible (see [QoS Requested vs. Offered Compatibility—the RxO Property \(Section 4.2.1 on page 167\)](#)).

The use of the `DDS_TopicQos` structure to set the values of any QoS Policy except `TOPIC_DATA`—which only applies to *Topics*—is really a way to share a single set of values with the associated *DataWriters* and *DataReaders*, as well as to avoid creating those entities with inconsistent QoS Policies.

To cause a *DataWriter* to use its *Topic*'s QoS settings, either:

- Pass `DDS_DATAWRITER_QOS_USE_TOPIC_QOS` to `create_datawriter()`, or
- Call the *Publisher*'s `copy_from_topic_qos()` operation

To cause a *DataReader* to use its *Topic*'s QoS settings, either:

- Pass `DDS_DATAREADER_QOS_USE_TOPIC_QOS` to `create_datareader()`, or
- Call the *Subscriber*'s `copy_from_topic_qos()` operation

Please refer to the API Reference HTML documentation for the *Publisher*'s `create_datawriter()` and *Subscriber*'s `create_datareader()` methods for more information about using values from the *Topic* QoS Policies when creating *DataWriters* and *DataReaders*.

5.1.5 Setting Up TopicListeners

When you create a *Topic*, you have the option of giving it a *Listener*. A *TopicListener* includes just one callback routine, `on_inconsistent_topic()`. If you create a *TopicListener* (either as part of the *Topic* creation call, or later with the `set_listener()` operation), Connex DDS will invoke the *TopicListener*'s `on_inconsistent_topic()` method whenever it detects that another application has created a *Topic* with same name but associated with a different user data type. For more information, see [INCONSISTENT_TOPIC Status \(Section 5.3.1 on page 211\)](#).

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

If a *Topic*'s Listener has not been set and Connex DDS detects an inconsistent Topic, the *DomainParticipantListener* (if it exists) will be notified instead (see [Setting Up DomainParticipantListeners \(Section 8.3.5 on page 560\)](#)). So you only need to set up a **TopicListener** if you need to perform specific actions when there is an error on that particular *Topic*. In most cases, you can set the *TopicListener* to NULL and process inconsistent-topic errors in the *DomainParticipantListener* instead.

5.1.6 Navigating Relationships Among Entities

5.1.6.1 Finding a Topic's DomainParticipant

To retrieve a handle to the Topic's DomainParticipant, use the **get_participant()** operation:

```
DDSDomainParticipant* DDSTopicDescription::get_participant()
```

Notice that this method belongs to the **DDSTopicDescription** class, which is the base class for **DDSTopic**.

5.1.6.2 Retrieving a Topic's Name or DDS Type Name

If you want to retrieve the *topic_name* or *type_name* used in the **create_topic()** operation, use these methods:

```
const char* DDSTopicDescription::get_type_name();
const char* DDSTopicDescription::get_name();
```

Notice that these methods belong to the **DDSTopicDescription** class, which is the base class for **DDSTopic**.

5.2 Topic QoS Policies

This section describes the only QoS Policy that strictly applies to Topics (and no other types of *Entities*)—the TOPIC_DATA QoS Policy. For a complete list of the QoS Policies that can be set for Topics, see [Table 5.2 Topic QoS Policies](#).

Most of the QoS Policies that can be set on a Topic can also be set on the corresponding DataWriter and/or DataReader. The Topic's QoS Policy is essentially just a place to store QoS settings that you plan to share with multiple entities that use that Topic (see how in [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#)); they are not used otherwise and are not propagated on the wire.

5.2.1 TOPIC_DATA QoS Policy

This QoS Policy provides an area where your application can store additional information related to the *Topic*. This information is passed between applications during discovery (see [Discovery \(Section Chapter 14 on page 709\)](#)) using builtin-topics (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)). How this information is used will be up to user code. Connex DDS does not do anything with the information

stored as `TOPIC_DATA` except to pass it to other applications. Use cases are usually application-to-application identification, authentication, authorization, and encryption purposes.

The value of the `TOPIC_DATA` QosPolicy is sent to remote applications when they are first discovered, as well as when the *Topic*'s `set_qos()` method is called after changing the value of the `TOPIC_DATA`. User code can set listeners on the builtin *DataReaders* of the builtin *Topics* used by Connex DDS to propagate discovery information. Methods in the builtin topic listeners will be called whenever new applications, *DataReaders*, and *DataWriters* are found. Within the user callback, you will have access to the `TOPIC_DATA` that was set for the associated *Topic*.

Currently, `TOPIC_DATA` of the associated *Topic* is only propagated with the information that declares a *DataWriter* or *DataReader*. Thus, you will need to access the value of `TOPIC_DATA` through `DDS_PublicationBuiltinTopicData` or `DDS_SubscriptionBuiltinTopicData` (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)).

The structure for the `TOPIC_DATA` QosPolicy includes just one field, as seen in [Table 5.3 DDS_TopicDataQosPolicy](#). The field is a sequence of octets that translates to a contiguous buffer of bytes whose contents and length is set by the user. The maximum size for the data are set in the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#).

Table 5.3 DDS_TopicDataQosPolicy

Type	Field Name	Description
DDS_OctetSeq	value	default: empty

This policy is similar to the `GROUP_DATA` ([GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)) and `USER_DATA` ([USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)) policies that apply to other types of *Entities*.

5.2.1.1 Example

One possible use of `TOPIC_DATA` is to send an associated XML schema that can be used to process the data stored in the associated user data structure of the *Topic*. The schema, which can be passed as a long sequence of characters, could be used by an XML parser to take DDS samples of the data received for a *Topic* and convert them for updating some graphical user interface, web application or database.

5.2.1.2 Properties

This QosPolicy can be modified at any time. A change in the QosPolicy will cause Connex DDS to send packets containing the new `TOPIC_DATA` to all of the other applications in the DDS domain.

Because *Topics* are created independently by the applications that use the *Topic*, there may be different instances of the same *Topic* (same topic name and DDS data type) in different applications. The `TOPIC_DATA` for different instances of the same *Topic* may be set differently by different applications.

5.2.1.3 Related QosPolicies

- [GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)
- [USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)
- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#)

5.2.1.4 Applicable DDS Entities

- [Topics \(Section 5.1 on page 200\)](#)

5.2.1.5 System Resource Considerations

As mentioned earlier, the maximum size of the TOPIC_DATA is set in the **topic_data_max_length** field of the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#). Because Connex DDS will allocate memory based on this value, you should only increase this value if you need to. If your system does not use TOPIC_DATA, then you can set this value to 0 to save memory. Setting the value of the TOPIC_DATA QosPolicy to hold data longer than the value set in the **topic_data_max_length** field will result in failure and an INCONSISTENT_QOS_POLICY return code.

However, should you decide to change the maximum size of TOPIC_DATA, you *must* make certain that all applications in the DDS domain have changed the value of `topic_data_max_length` to be the same. If two applications have different limits on the size of TOPIC_DATA, and one application sets the TOPIC_DATA QosPolicy to hold data that is greater than the maximum size set by another application, then the *DataWriters* and *DataReaders* of that *Topic* between the two applications will *not* connect. This is also true for the GROUP_DATA ([GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)) and USER_DATA ([USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)) QosPolicies.

5.3 Status Indicator for Topics

There is only one communication status defined for a *Topic*, ON_INCONSISTENT_TOPIC. You can use the **get_inconsistent_topic_status()** operation to access the current value of the status or use a *TopicListener* to catch the change in the status as it occurs. See [Listeners \(Section 4.4 on page 177\)](#) for a general discussion on Listeners and Statuses.

5.3.1 INCONSISTENT_TOPIC Status

In order for a *DataReader* and a *DataWriter* with the same *Topic* to communicate, their DDS types must be consistent according to the *DataReader's* type-consistency enforcement policy value, defined in its [TYPE_CONSISTENCY_ENFORCEMENT QosPolicy \(Section 7.6.6 on page 532\)](#). This status indicates that another *DomainParticipant* has created a *Topic* using the same name as the local *Topic*, but with an inconsistent DDS type.

The status is a structure of type **DDS_InconsistentTopicStatus**, see [Table 5.4 DDS_InconsistentTopicStatus Structure](#). The **total_count** keeps track of the total number of (*DataReader*, *DataWriter*) pairs with topic names that match the Topic to which this status is attached, but whose DDS types are inconsistent. The *TopicListener*'s **on_inconsistent_topic()** operation is invoked when this status changes (an inconsistent topic is found). You can also retrieve the current value by calling the *Topic*'s **get_inconsistent_topic_status()** operation.

The value of **total_count_change** reflects the number of inconsistent topics that were found since the last time **get_inconsistent_topic_status()** was called by user code or **on_inconsistent_topic()** was invoked by Connex DDS.

Table 5.4 DDS_InconsistentTopicStatus Structure

Type	Field Name	Description
DDS_Long	total_count	Total cumulative count of (<i>DataReader</i> , <i>DataWriter</i>) pairs whose topic names match the Topic to which this status is attached, but whose DDS types are inconsistent.
DDS_Long	total_count_change	The change in total_count since the last time this status was read.

5.4 ContentFilteredTopics

A *ContentFilteredTopic* is a *Topic* with filtering properties. It makes it possible to subscribe to topics and at the same time specify that you are only interested in a subset of the *Topic*'s data.

For example, suppose you have a *Topic* that contains a temperature reading for a boiler, but you are only interested in temperatures outside the normal operating range. A *ContentFilteredTopic* can be used to limit the number of DDS data samples a *DataReader* has to process and may also reduce the amount of data sent over the network.

This section includes the following:

5.4.1 Overview

A *ContentFilteredTopic* creates a relationship between a *Topic*, also called the related topic, and user-specified filtering properties. The filtering properties consist of an expression and a set of parameters.

- The filter expression evaluates a logical expression on the Topic content. The filter expression is similar to the WHERE clause in a SQL expression.
- The parameters are strings that give values to the 'parameters' in the filter expression. There must be one parameter string for each parameter in the filter expression.

A `ContentFilteredTopic` is a type of topic description, and can be used to create *DataReaders*. However, a `ContentFilteredTopic` is *not* an entity—it does not have `QoS` Policies or *Listeners*.

A `ContentFilteredTopic` relates to other entities in Connex DDS as follows:

- `ContentFilteredTopics` are used when creating *DataReaders*, not *DataWriters*.
- Multiple *DataReaders* can be created with the same `ContentFilteredTopic`.
- A `ContentFilteredTopic` belongs to (is created/deleted by) a *DomainParticipant*.
- A `ContentFilteredTopic` and *Topic* must be in the same *DomainParticipant*.
- A `ContentFilteredTopic` can only be related to a single *Topic*.
- A *Topic* can be related to multiple `ContentFilteredTopics`.
- A `ContentFilteredTopic` can have the same name as a *Topic*, but `ContentFilteredTopics` must have unique names within the same *DomainParticipant*.
- A *DataReader* created with a `ContentFilteredTopic` will use the related *Topic*'s `QoS` and *Listeners*.
- Changing filter parameters on a `ContentFilteredTopic` causes *all DataReaders* using the same `ContentFilteredTopic` to see the change.
- A *Topic* cannot be deleted as long as at least one `ContentFilteredTopic` that has been created with it exists.
- A `ContentFilteredTopic` cannot be deleted as long as at least one *DataReader* that has been created with the `ContentFilteredTopic` exists.

5.4.2 Where Filtering is Applied—Publishing vs. Subscribing Side

Filtering may be performed on either side of the distributed application. (The *DataWriter* obtains the filter expression and parameters from the *DataReader* during discovery.)

When batching is enabled, content filtering is always done on the reader side.

Connex DDS also supports network-switch filtering for multi-channel *DataWriters* (see [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#)).

A *DataWriter* will automatically filter DDS data samples for a *DataReader* if *all* of the following are true; otherwise filtering is performed by the *DataReader*.

1. The *DataWriter* is filtering for no more than **writer_resource_limits.max_remote_reader_filters** *DataReaders* at the same time.
 - There is a resource-limit on the *DataWriter* called **writer_resource_limits.max_remote_reader_filters** (see [DATA_WRITER_RESOURCE_LIMITS QoS Policy \(DDS Extension\) \(Section 6.5.4 on page 359\)](#)). This value can be from $[0, (2^{31})-2]$. 0 means do not filter any *DataReader*; 32 (default value) means filter up to 32 *DataReaders*.

- If a *DataWriter* is filtering **max_remote_reader_filters** *DataReaders* at the same time and a new filtered *DataReader* is created, then the newly created *DataReader* (**max_remote_reader_filters** + 1) is not filtered. Even if one of the first (**max_remote_reader_filters**) *DataReaders* is deleted, that already created *DataReader* (**max_remote_reader_filters** + 1) will *still* not be filtered. However, any subsequently created *DataReaders* will be filtered as long as the number of *DataReaders* currently being filtered is not more than **writer_resource_limits.max_remote_reader_filters**.
2. The *DataReader* is not subscribing to data using multicast.
 3. There are no more than 4 matching *DataReaders* in the same locator (see [Peer Descriptor Format \(Section 14.2.1 on page 713\)](#)).
 4. The *DataWriter* has infinite liveliness. (See [LIVELINESS QosPolicy \(Section 6.5.13 on page 382\)](#).)
 5. The *DataWriter* is *not* using an Asynchronous Publisher. (That is, the *DataWriter*'s [PUBLISH_MODE QosPolicy \(DDS Extension\) \(Section 6.5.18 on page 397\)](#) **kind** is set to `DDS_SYNCHRONOUS_PUBLISHER_MODE_QOS`.) See Note below.
 6. If you are using a custom filter (not the default one), it must be registered in the *DomainParticipant* of the *DataWriter* and the *DataReader*.
 7. The *DataWriter* is not configured to use batching.

Notes:

- Connex DDS supports limited writer-side filtering if asynchronous publishing is enabled. The middleware will not send any DDS sample to a destination if the DDS sample is filtered out by all the *DataReaders* on that destination. However, if there is one *DataReader* to which the DDS sample has to be sent, all the *DataReaders* on the destination will do reader side filtering for the incoming DDS sample.
- In addition to filtering new DDS samples, a *DataWriter* can also be configured to filter previously written DDS samples stored in the *DataWriter*'s queue for newly discovered *DataReaders*. To do so, use the **refilter** field in the *DataWriter*'s [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#).
- When batching is enabled, content filtering is always done on the reader side. See [BATCH QosPolicy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#).

5.4.3 Creating ContentFilteredTopics

To create a *ContentFilteredTopic* that uses the default SQL filter, use the *DomainParticipant*'s **create_contentfilteredtopic()** operation:

```
DDS_ContentFilteredTopic *create_contentfilteredtopic(
    const char * name,
    const DDS_Topic * related_topic,
```

```
const char * filter_expression,
const DDS_StringSeq & expression_parameters)
```

Or, to use a custom filter or the builtin STRINGMATCH filter (see [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#)), use the `create_contentfilteredtopic_with_filter()` variation:

```
DDS_ContentFilteredTopic *create_contentfilteredtopic_with_filter(
    const char * name,
    DDS_Topic * related_topic,
    const char * filter_expression,
    const DDS_StringSeq & expression_parameters,
    const char * filter_name = DDS_SQLFILTER_NAME)
```

Where:

- name** Name of the ContentFilteredTopic. Note that it *is* legal for a ContentFilteredTopic to have the same name as a Topic in the same *DomainParticipant*, but a ContentFilteredTopic cannot have the same name as another ContentFilteredTopic in the same *DomainParticipant*. This parameter cannot be NULL.
- related_topic** The related Topic to be filtered. The related topic must be in the same *DomainParticipant* as the ContentFilteredTopic. This parameter cannot be NULL. The same related topic can be used in many different ContentFilteredTopics.
- filter_expression** A logical expression on the contents on the Topic. If the expression evaluates to TRUE, a DDS sample is received; otherwise it is discarded. This parameter cannot be NULL. The notation for this expression depends on the filter that you are using (specified by the **filter_name** parameter). See [SQL Filter Expression Notation \(Section 5.4.6 on page 222\)](#) and [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#). The **filter_expression** can be changed with `set_expression()` ([Setting an Expression's Filter and Parameters \(Section 5.4.5.2 on page 220\)](#)).
- expression_parameters** A string sequence of filter expression parameters. Each parameter corresponds to a positional argument in the filter expression: element 0 corresponds to positional argument 0, element 1 to positional argument 1, and so forth.
- The **expression_parameters** can be changed with `set_expression_parameters()` or `set_expression()` ([Setting an Expression's Filter and Parameters \(Section 5.4.5.2 on page 220\)](#)), `append_to_expression_parameter()` ([Appending a String to an Expression Parameter \(Section 5.4.5.3 on page 220\)](#)) and `remove_from_expression_parameter()` ([Removing a String from an Expression Parameter \(Section 5.4.5.4 on page 221\)](#)).
- filter_name**

Name of the content filter to use for filtering. The filter must have been previously registered with the *DomainParticipant* (see [Registering a Custom Filter \(Section 5.4.8.2 on page 234\)](#)). There are two builtin filters, `DDS_SQLFILTER_NAME`¹ (the default filter) and `DDS_STRINGMATCHFILTER_NAME`—these are automatically registered.

To use the `STRINGMATCH` filter, call `create_contentfilteredtopic_with_filter()` with `"DDS_STRINGMATCHFILTER_NAME"` as the `filter_name`. `STRINGMATCH` filter expressions have the syntax:

`<field name> MATCH <string pattern>` (see [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#)).

If you run *RTI Code Generator* with `-notypecode`, you must use the `"with_filter"` version with a custom filter instead—do not use the builtin SQL filter or the `STRINGMATCH` filter with the `-notypecode` option because they require type codes.

To summarize:

- To use the builtin default SQL filter:
 - Do not use `-notypecode` when running *RTI Code Generator*
 - Call `create_contentfilteredtopic()`
 - See [SQL Filter Expression Notation \(Section 5.4.6 on page 222\)](#)
- To use the builtin `STRINGMATCH` filter:
 - Do not use `-notypecode` when running *RTI Code Generator*
 - Call `create_contentfilteredtopic_with_filter()`, setting the `filter_name` to `DDS_STRINGMATCHFILTER_NAME`
 - See [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#)
- To use a custom filter:
 - Call `create_contentfilteredtopic_with_filter()`, setting the `filter_name` to a registered custom filter
- To use *RTI Code Generator* with `-notypecode`:
 - Call `create_contentfilteredtopic_with_filter()`, setting the `filter_name` to a registered custom filter

¹ In the Java and C# APIs, you can access the names of the builtin filters by using `DomainParticipant.SQLFILTER_NAME` and `DomainParticipant.STRINGMATCHFILTER_NAME`.

² In the Java and C# APIs, you can access the names of the builtin filters by using `DomainParticipant.SQLFILTER_NAME` and `DomainParticipant.STRINGMATCHFILTER_NAME`.

Be careful with memory management of the string sequence in some of the ContentFilteredTopic APIs. See the **String Support** section in the API Reference HTML documentation (within the **Infrastructure** module) for details on sequences.

5.4.3.1 Creating ContentFilteredTopics for Built-in DDS Types

To create a ContentFilteredTopic for a built-in DDS type (see [Built-in Data Types \(Section 3.2 on page 30\)](#)), use the standard *DomainParticipant* operations, `create_contentfilteredtopic()` or `create_contentfilteredtopic_with_filter`.

The field names used in the filter expressions for the built-in SQL (see [SQL Filter Expression Notation \(Section 5.4.6 on page 222\)](#)) and StringMatch filters (see [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#)) must correspond to the names provided in the IDL description of the built-in DDS types.

ContentFilteredTopic Creation Examples:

For simplicity, error handling is not shown in the following examples.

C Example:

```
DDS_Topic * topic = NULL;
DDS_ContentFilteredTopic * contentFilteredTopic = NULL;
struct DDS_StringSeq parameters = DDS_SEQUENCE_INITIALIZER;
/* Create a string ContentFilteredTopic */
topic = DDS_DomainParticipant_create_topic(
    participant, "StringTopic",
    DDS_StringTypeSupport_get_type_name(),
    &DDS_TOPIC_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);
contentFilteredTopic =
    DDS_DomainParticipant_create_contentfilteredtopic(
        participant,
        "StringContentFilteredTopic",
        topic,
        "value = 'Hello World!'", &parameters);
```

C++ Example with Namespaces:

```
using namespace DDS;
...
/* Create a String ContentFilteredTopic */
Topic * topic = participant->create_topic(
    "StringTopic",
    StringTypeSupport::get_type_name(),
    TOPIC_QOS_DEFAULT,
    NULL, STATUS_MASK_NONE);
StringSeq parameters;
ContentFilteredTopic * contentFilteredTopic =
    participant->create_contentfilteredtopic(
        "StringContentFilteredTopic", topic,
        "value = 'Hello World!'", parameters);
```

C++/CLI Example:

```
using namespace DDS;
...
/* Create a String ContentFilteredTopic */
Topic^ topic = participant->create_topic(
    "StringTopic", StringTypeSupport::get_type_name(),
    DomainParticipant::TOPIC_QOS_DEFAULT,
    nullptr, StatusMask::STATUS_MASK_NONE);
StringSeq^ parameters = gcnew StringSeq();
ContentFilteredTopic^ contentFilteredTopic =
    participant->create_contentfilteredtopic(
        "StringContentFilteredTopic", topic,
        "value = 'Hello World!'", parameters);
```

C# Example:

```
using namespace DDS;
...
/* Create a String ContentFilteredTopic */
Topic topic = participant.create_topic(
    "StringTopic", StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusMask.STATUS_MASK_NONE);
StringSeq parameters = new StringSeq();
ContentFilteredTopic contentFilteredTopic =
    participant.create_contentfilteredtopic(
        "StringContentFilteredTopic", topic,
        "value = 'Hello World!'", parameters);
```

Java Example:

```
import com.rti.dds.type.builtin.*;
...
/* Create a String ContentFilteredTopic */
Topic topic = participant.create_topic(
    "StringTopic", StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusKind.STATUS_MASK_NONE);
StringSeq parameters = new StringSeq();
ContentFilteredTopic contentFilteredTopic =
    participant.create_contentfilteredtopic(
        "StringContentFilteredTopic", topic,
        "value = 'Hello World!'", parameters);
```

5.4.4 Deleting ContentFilteredTopics

To delete a ContentFilteredTopic, use the *DomainParticipant's* **delete_contentfilteredtopic()** operation:

Make sure no *DataReaders* are using the ContentFilteredTopic. (If this is not true, the operation returns **PRECONDITION_NOT_MET**.)

Delete the ContentFilteredTopic by using the *DomainParticipant's* **delete_contentfilteredtopic()** operation.

```
DDS_ReturnCode_t delete_contentfilteredtopic
(DDSContentFilteredTopic * a_contentfilteredtopic)
```

5.4.5 Using a ContentFilteredTopic

Once you've created a ContentFilteredTopic, you can use the operations listed in [Table 5.5 ContentFilteredTopic Operations](#).

Table 5.5 ContentFilteredTopic Operations

Operation	Description	Reference
append_to_expression_parameter	Concatenates a string value to the input expression parameter	Appending a String to an Expression Parameter (Section 5.4.5.3 on the facing page)
get_expression_parameters	Gets the expression parameters.	Getting the Current Expression Parameters (Section 5.4.5.1 below)
get_filter_expression	Gets the expression.	Getting the Filter Expression (Section 5.4.5.5 on page 221)
get_related_topic	Gets the related Topic.	Getting the Related Topic (Section 5.4.5.6 on page 221)
narrow	Casts a DDS_TopicDescription pointer to a ContentFilteredTopic pointer.	‘Narrowing’ a ContentFilteredTopic to a TopicDescription (Section 5.4.5.7 on page 222)
remove_from_expression_parameter	Removes a string value from the input expression parameter	Removing a String from an Expression Parameter (Section 5.4.5.4 on page 221)
set_expression	Changes the filter expression and parameters.	Setting an Expression’s Filter and Parameters (Section 5.4.5.2 on the facing page)
set_expression_parameters	Changes the expression parameters.	

5.4.5.1 Getting the Current Expression Parameters

To get the expression parameters, use the ContentFilteredTopic’s `get_expression_parameters()` operation:

```
DDS_ReturnCode_t get_expression_parameters(
    struct DDS_StringSeq & parameters)
```

Where:

parameters The filter expression parameters.

The memory for the strings in this sequence is managed as described in the **String Support** section of the API Reference HTML documentation (within the **Infrastructure** module). In particular, be careful to avoid a situation in which Connex DDS allocates a string on your behalf and you then reuse that string in such a way that Connex DDS believes it to have more memory allocated to it than it actually does. This parameter cannot be NULL.

This operation gives you the expression parameters that were specified on the last successful call to **set_expression_parameters()** or **set_expression()**, or if they were never called, the parameters specified when the ContentFilteredTopic was created.

5.4.5.2 Setting an Expression's Filter and Parameters

To change the filter expression and expression parameters associated with a ContentFilteredTopic:

```
DDS_ReturnCode set_expression(
    const char * expression,
    const struct DDS_StringSeq & parameters)
```

To change just the expression parameters (not the filter expression):

```
DDS_ReturnCode_t set_expression_parameters(
    const struct DDS_StringSeq & parameters)
```

Where:

- expression** The new expression to be set in the ContentFilteredTopic.
- parameters** The filter expression parameters. Each element in the parameter sequence corresponds to a positional parameter in the filter expression. When using the default DDS_SQLFILTER_NAME, parameter strings are automatically converted to the member type. For example, "4" is converted to the integer 4. This parameter cannot be NULL.

The ContentFilteredTopic's operations do not manage the sequences; you must ensure that the parameter sequences are valid. Please refer to the **String Support** section in the API Reference HTML documentation (within the Infrastructure module) for details on sequences.

5.4.5.3 Appending a String to an Expression Parameter

To concatenate a string to an expression parameter, use the ContentFilteredTopic's **append_to_expression_parameter()** operation:

```
DDS_ReturnCode_t append_to_expression_parameter(
    const DDS_Long index,
    const char* value);
```

When using the STRINGMATCH filter, **index** must be 0.

This function is only intended to be used with the builtin SQL and STRINGMATCH filters. This function can be used in expression parameters associated with MATCH operators (see [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 on page 228\)](#)) to add a pattern to the match pattern list. For example, if **filter_expression** is:

```
symbol MATCH 'IBM'
```

Then **append_to_expression_parameter(0, "MSFT")** would generate the expression:

```
symbol MATCH 'IBM,MSFT'
```

5.4.5.4 Removing a String from an Expression Parameter

To remove a string from an expression parameter use the ContentFilteredTopic's **remove_from_expression_parameter()** operation:

```
DDS_ReturnCode_t remove_from_expression_parameter(
    const DDS_Long index,    const char* value)
```

When using the STRINGMATCH filter, **index** must be 0.

This function is only intended to be used with the builtin SQL and STRINGMATCH filters. It can be used in expression parameters associated with MATCH operators (see [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 on page 228\)](#)) to remove a pattern from the match pattern list. For example, if **filter_expression** is:

```
symbol MATCH 'IBM,MSFT'
```

Then **remove_from_expression_parameter(0, "IBM")** would generate the expression:

```
symbol MATCH 'MSFT'
```

5.4.5.5 Getting the Filter Expression

To get the filter expression that was specified when the ContentFilteredTopic was created or when **set_expression()** was used:

```
const char* get_filter_expression ()
```

5.4.5.6 Getting the Related Topic

To get the related *Topic* that was specified when the ContentFilteredTopic was created:

```
DDS_Topic * get_related_topic ()
```

5.4.5.7 ‘Narrowing’ a ContentFilteredTopic to a TopicDescription

To safely cast a DDS_TopicDescription pointer to a ContentFilteredTopic pointer, use the ContentFilteredTopic’s **narrow()** operation:

```
DDS_TopicDescription* narrow ()
```

5.4.6 SQL Filter Expression Notation

A SQL filter expression is similar to the **WHERE** clause in SQL. The SQL expression format provided by Connex DDS also supports the **MATCH** operator as an extended operator (see [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 on page 228\)](#)).

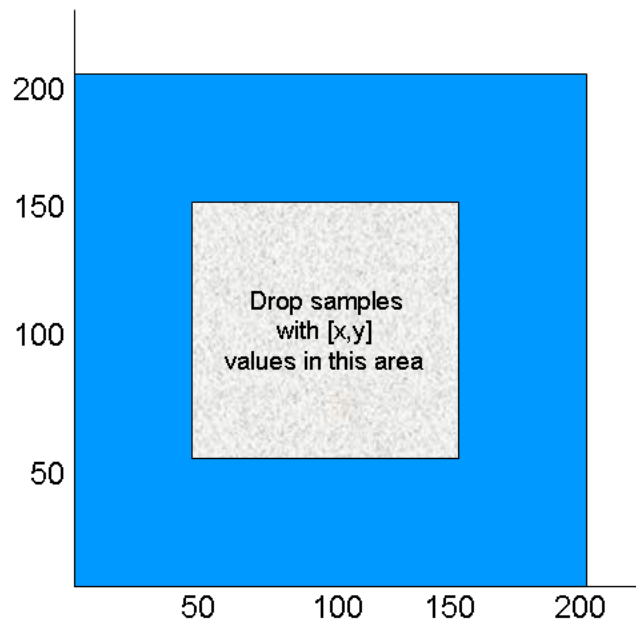
The following sections provide more information:

- [Example SQL Filter Expressions \(Section 5.4.6.1 below\)](#)
- [SQL Grammar \(Section 5.4.6.2 on page 224\)](#)
- [Token Expressions \(Section 5.4.6.3 on page 225\)](#)
- [Type Compatibility in the Predicate \(Section 5.4.6.4 on page 227\)](#)
- [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 on page 228\)](#)
- [Composite Members \(Section 5.4.6.6 on page 229\)](#)
- [Strings \(Section 5.4.6.7 on page 229\)](#)
- [Enumerations \(Section 5.4.6.8 on page 230\)](#)
- [Pointers \(Section 5.4.6.9 on page 230\)](#)
- [Arrays \(Section 5.4.6.10 on page 230\)](#)
- [Sequences \(Section 5.4.6.11 on page 231\)](#)

5.4.6.1 Example SQL Filter Expressions

Assume that you have a *Topic* with two floats, X and Y, which are the coordinates of an object moving inside a rectangle measuring 200 x 200 units. This object moves quite a bit, generating lots of DDS samples that you are not interested in. Instead you only want to receive DDS samples *outside* the middle of the rectangle, as seen in [Filtering Example \(Section Figure 5.5 on the next page\)](#). That is, you want to filter *out* data points in the gray box.

Figure 5.5 Filtering Example



The filter expression would look like this (remember the expression is written so that DDS samples that we *do* want will *pass*):

```
"(X < 50 or X > 150) and (Y < 50 or Y > 150)"
```

While this filter works, it cannot be changed after the `ContentFilteredTopic` has been created. Suppose you would like the ability to adjust the coordinates that are considered outside the acceptable range (changing the size of the gray box). You can achieve this by using filter parameters. An more flexible way to write the expression is this:

```
"(X < %0 or X > %1) and (Y < %2 or Y > %3)"
```

Recall that when you create a `ContentFilteredTopic` (see [Creating ContentFilteredTopics \(Section 5.4.3 on page 214\)](#)), you pass a `expression_parameters` string sequence as one of the parameters. Each element in the string sequence corresponds to one argument.

See the **String** and **Sequence Support** sections of the API Reference HTML documentation (from the **Modules** page, select **RTI Connex DDS API Reference, Infrastructure Module**).

In C++, the filter parameters could be assigned like this:

```
FilterParameter[0] = "50";
FilterParameter[1] = "150";
FilterParameter[2] = "50";
FilterParameter[3] = "150";
```

With these parameters, the filter expression is identical to the first approach. However, it is now possible to change the parameters by calling `set_expression_parameters()`. For example, perhaps you decide that you only want to see data points where $X < 10$ or $X > 190$. To make this change:

```
FilterParameter[0] = 10
FilterParameter[1] = 190
set_expression_parameters(...)
```

The new filter parameters will affect all *DataReaders* that have been created with this *ContentFilteredTopic*.

5.4.6.2 SQL Grammar

This section describes the subset of SQL syntax, in Backus–Naur Form (BNF), that you can use to form filter expressions.

The following notational conventions are used:

NonTerminals are typeset in italics.

'Terminals' are quoted and typeset in a fixed-width font. They are written in upper case in most cases in the BNF-grammar below, but should be case insensitive.

TOKENS are typeset in bold.

The notation (element // ',') represents a non-empty, comma-separated list of elements.

```
Expression ::= FilterExpression
            | TopicExpression
            | QueryExpression
            .
FilterExpression ::= Condition
TopicExpression ::= SelectFrom { Where } ';'
QueryExpression ::= { Condition }{ 'ORDER BY' (FIELDNAME // ',') }
                .
SelectFrom      ::= 'SELECT' Aggregation 'FROM' Selection
                .
Aggregation     ::= '*'
                | (SubjectFieldSpec // ',')
                .
SubjectFieldSpec ::= FIELDNAME
                | FIELDNAME 'AS' IDENTIFIER
                | FIELDNAME IDENTIFIER
                .
Selection       ::= TOPICNAME
                | TOPICNAME NaturalJoin JoinItem
                .
JoinItem        ::= TOPICNAME
                | TOPICNAME NaturalJoin JoinItem
                | '(' TOPICNAME NaturalJoin JoinItem ')'
                .
NaturalJoin     ::= 'INNER JOIN'
```

```

| 'INNER NATURAL JOIN'
| 'NATURAL JOIN'
| 'NATURAL INNER JOIN'
.
Where ::= 'WHERE' Condition
.
Condition ::= Predicate
| Condition 'AND' Condition
| Condition 'OR' Condition
| 'NOT' Condition
| '(' Condition ')'
.
Predicate ::= ComparisonPredicate
| BetweenPredicate
.
ComparisonPredicate ::= ComparisonTerm RelOp ComparisonTerm
.
ComparisonTerm ::= FieldIdentifier
| Parameter
.
BetweenPredicate ::= FieldIdentifier 'BETWEEN' Range
| FieldIdentifier 'NOT BETWEEN' Range
.
FieldIdentifier ::= FIELDNAME
| IDENTIFIER
.
RelOp ::= '=' | '>' | '>=' | '<' | '<=' | '<>' | 'LIKE' | 'MATCH'
.
Range ::= Parameter 'AND' Parameter
.
Parameter ::= INTEGERVALUE
| CHARVALUE
| FLOATVALUE
| STRING
| ENUMERATEDVALUE
| BOOLEANVALUE
| PARAMETER

```

INNER JOIN, INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. They are all supported because they all are part of the SQL standard.

5.4.6.3 Token Expressions

The syntax and meaning of the tokens used in SQL grammar is described as follows:

IDENTIFIER—An identifier for a FIELDNAME, defined as any series of characters 'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '_' but may *not* start with a digit.

```
IDENTIFIER: LETTER (PART_LETTER)*
```

where LETTER: ["A"-"Z", "_", "a"-"z"] PART_LETTER: ["A"-"Z", "_", "a"-"z", "0"-"9"]

FIELDNAME—A reference to a field in the data structure. A dot '.' is used to navigate through nested structures. The number of dots that may be used in a FIELDNAME is unlimited. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure, which may or may not match the fieldnames that appear on the language-specific (e.g., C/C++, Java) mapping of the structure. To reference the $n+1$ element in an array or sequence, use the notation '`[n]`', where n is a natural number (zero included). FIELDNAME must resolve to a primitive IDL type; that is either boolean, octet, (unsigned) short, (unsigned) long, (unsigned) long long, float double, char, wchar, string, wstring, or enum.

```
FIELDNAME: FieldNamePart ( "." FieldNamePart ) *
```

where FieldNamePart : IDENTIFIER ("[" Index "]") * Index > : (["0"-"9"])+ | ["0x","0X"](["0"-"9", "A"-"F", "a"-"f"])+

Primitive IDL types referenced by FIELDNAME are treated as different types in Predicate according to the following table:

Predicate Data Type	IDL Type
BOOLEANVALUE	boolean
INTEGERVALUE	octet, (unsigned) short, (unsigned) long, (unsigned) long long
FLOATVALUE	float, double
CHARVALUE	char, wchar
STRING	string, wstring
ENUMERATEDVALUE	enum

TOPICNAME—An identifier for a topic, and is defined as any series of characters 'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '_' but may *not* start with a digit.

```
TOPICNAME : IDENTIFIER
```

INTEGERVALUE—Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. 'L' or 'l' must be used for long long, otherwise long is assumed. A hexadecimal number is preceded by **0x** and must be a valid hexadecimal expression.

```
INTEGERVALUE : (["+", "-"])? (["0"-"9"])+ [("L", "l")]?
| (["+", "-"])? ["0x", "0X"] (["0"-"9",
"A"-"F", "a"-"f"])+ [("L", "l")]?
```

CHARVALUE—A single character enclosed between single quotes.

```
CHARVALUE : "'" (~["'"])? "'"
```

FLOATVALUE—Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point ('.'). 'F' or 'f' must be used for float, otherwise double is assumed. A power-of-ten expression may be postfixed, which has the syntax *en* or *En*, where *n* is a number, optionally preceded by a plus or minus sign.

```
FLOATVALUE : ([ "+", "-" ]? ([ "0"-"9" ]* ( "." )? ([ "0"-"9" ]+
(EXPONENT)? [ ("F", 'f') ])?
```

where EXPONENT: ["e", "E"] (["+", "-"])? (["0"-"9"])+

STRING—Any series of characters encapsulated in single quotes, except the single quote itself.

```
STRING : "'" (~["'"]) * "'"
```

ENUMERATEDVALUE—A reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.

```
ENUMERATEDVALUE : "'" ["A" - "Z", "a" - "z"]
["A" - "Z", "a" - "z", "_", "0" - "9"] * "'"
```

BOOLEANVALUE—Can either be **TRUE** or **FALSE**, and is case insensitive.

```
BOOLEANVALUE : ["TRUE", "FALSE"]
```

PARAMETER—Takes the form *%n*, where *n* represents a natural number (zero included) smaller than 100. It refers to the (*n* + 1)th argument in the given context. This argument can only be in primitive type value format. It cannot be a **FIELDNAME**.

```
PARAMETER : "%" ([ "0"-"9" ])+
```

5.4.6.4 Type Compatibility in the Predicate

As seen in [Table 5.6 Valid Type Comparisons](#), only certain combinations of type comparisons are valid in the Predicate.

Table 5.6 Valid Type Comparisons

	BOOLEAN VALUE	INTEGER VALUE	FLOAT VALUE	CHAR VALUE	STRING	ENUMERATED VALUE
BOOLEAN	YES					
INTEGERVALUE		YES	YES			
FLOATVALUE		YES	YES			

Table 5.6 Valid Type Comparisons

	BOOLEAN VALUE	INTEGER VALUE	FLOAT VALUE	CHAR VALUE	STRING	ENUMERATED VALUE
CHARVALUE				YES	YES	YES
STRING				YES	YES ¹	YES
ENUMERATED VALUE		YES		YES ²	YES ³	YES ⁴

5.4.6.5 SQL Extension: Regular Expression Matching

The relational operator **MATCH** may only be used with string fields. The right-hand operator is a string pattern. A string pattern specifies a template that the left-hand field must match.

MATCH is case-sensitive. These characters have special meaning: `, / ? * [] - ^ ! \ %`

The pattern allows limited "wild card" matching under the rules in [Table 5.7 Wild Card Matching](#).

The syntax is similar to the POSIX® **fnmatch** syntax⁵. The **MATCH** syntax is also similar to the 'subject' strings of TIBCO Rendezvous®. Some example expressions include:

```
"symbol MATCH 'NASDAQ/[A-G]*'"
"symbol MATCH 'NASDAQ/GOOG,NASDAQ/MSFT'"
```

Table 5.7 Wild Card Matching

Character	Meaning
,	A , separates a list of alternate patterns. The field string is matched if it matches one or more of the patterns.

^aSee [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 below\)](#).

²Because of the formal notation of the Enumeration values, they are compatible with string and char literals, but they are not compatible with string or char variables, i.e., "MyEnum=EnumValue" is correct, but "MyEnum=MyString" is not allowed.

³Because of the formal notation of the Enumeration values, they are compatible with string and char literals, but they are not compatible with string or char variables, i.e., "MyEnum=EnumValue" is correct, but "MyEnum=MyString" is not allowed.

⁴Only for same-type Enums.

⁵See <http://www.opengroup.org/onlinepubs/000095399/functions/fnmatch.html>.

Table 5.7 Wild Card Matching

Character	Meaning
/	A / in the pattern string matches a / in the field string. It separates a sequence of mandatory substrings.
?	A ? in the pattern string matches any single non-special characters in the field string.
*	A * in the pattern string matches 0 or more non-special characters in field string.
%	This special character is used to designate filter expression parameters.
\	<i>(Not supported)</i> Escape character for special characters.
[charlist]	Matches any one of the characters in charlist.
[!charlist] or [^charlist]	<i>(Not supported)</i> Matches any one of the characters <i>not</i> in charlist.
[s-e]	Matches any character from s to e , inclusive.
[!s-e] or [^s-e]	<i>(Not supported)</i> Matches any character <i>not</i> in the interval s to e .

5.4.6.6 Composite Members

Any member can be used in the filter expression, with the following exceptions:

- 128-bit floating point numbers (long doubles) are not supported
- bitfields are not supported
- **LIKE** is not supported

Composite members are accessed using the familiar dot notation, such as "**x.y.z > 5**". For unions, the notation is special due to the nature of the IDL union type.

On the publishing side, you can access the union discriminator with **myunion._d** and the actual member with **myunion._u.mymember**. If you want to use a ContentFilteredTopic on the subscriber side and filter a DDS sample with a top-level union, you can access the union discriminator directly with **_d** and the actual member with **mymember** in the filter expression.

5.4.6.7 Strings

The filter expression and parameters can use IDL strings. String constants must appear between single quotation marks ('').

For example:

```
" fish = 'salmon' "
```

Strings used as parameter values must contain the enclosing quotation marks (') within the parameter value; do not place the quotation marks within the expression statement. For example, the expression " symbol MATCH %0 " with parameter 0 set to " 'IBM' " is legal, whereas the expression " symbol MATCH '%0' " with parameter 0 set to " IBM " will not compile.

5.4.6.8 Enumerations

A filter expression can use enumeration values, such as GREEN, instead of the numerical value. For example, if *x* is an enumeration of GREEN, YELLOW and RED, the following expressions are valid:

```
"x = 'GREEN'"
"x < 'RED'"
```

5.4.6.9 Pointers

Pointers can be used in filter expressions and are automatically dereferenced to the correct value.

For example:

```
struct Point {
    long x;
    long y;
};
struct Rectangle {
    Point *u_l;
    Point *l_r;
};
```

The following expression is valid on a *Topic* of type Rectangle:

```
"u_l.x > l_r.x"
```

5.4.6.10 Arrays

Arrays are accessed with the familiar [] notation.

For example:

```
struct ArrayType {
    long value[255][5];
};
```

The following expression is valid on a *Topic* of type ArrayType:

```
"value[244][2] = 5"
```

In order to compare an array of bytes (octets in IDL), instead of comparing each individual element of the array using [] notation, Connex DDS provides a helper function, **hex()**. The **hex()** function can be used to represent an array of bytes (octets in IDL). To use the **hex()** function, use the notation **&hex()** and pass the byte array as a sequence of hexadecimal values.

For example:

```
&hex (07 08 09 0A 0B 0c 0D 0E 0F 10 11 12 13 14 15 16)
```

Here the leftmost-pair represents the byte and index 0.

Note: If the length of the octet array represented by the **hex()** function does not match the length of the field being compared, it will result in a compilation error.

For example:

```
struct ArrayType {
    octet value[2];
};
```

The following expression is valid:

```
"value = &hex(12 0A)"
```

5.4.6.11 Sequences

Sequence elements can be accessed using the () or [] notation.

For example:

```
struct SequenceType {
    sequence<long> s;
};
```

The following expressions are valid on a *Topic* of type SequenceType:

```
"s(1) = 5"
"s[1] = 5"
```

5.4.7 STRINGMATCH Filter Expression Notation

The STRINGMATCH Filter is a subset of the SQL filter; it only supports the MATCH relational operator on a single string field. It is introduced mainly for the use case of partitioning data according to channels in the *DataWriter's* [MULTI_CHANNEL QosPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386) in Market Data applications.

A STRINGMATCH filter expression has the following syntax:

```
<field name> MATCH <string pattern>
```

The STRINGMATCH filter is provided to support the narrow use case of filtering a single string field of the DDS sample against a comma-separated list of matching string values. It is intended to be used in conjunction with ContentFilteredTopic helper routines `append_to_expression_parameter()` ([Appending a String to an Expression Parameter \(Section 5.4.5.3 on page 220\)](#)) and `remove_from_expression_parameter()` ([Removing a String from an Expression Parameter \(Section 5.4.5.4 on page 221\)](#)), which allow you to easily append and remove individual string values from the comma-separated list of string values.

The STRINGMATCH filter must contain only one `<field name>`, and a single occurrence of the MATCH operator. The `<string pattern>` must be either the single parameter `%0`, or a single, comma-separated list of strings without intervening spaces.

During creation of a STRINGMATCH filter, the `<string pattern>` is automatically parameterized. That is, during creation, if the `<string pattern>` specified in the filter expression is not the parameter `%0`, then the comma-separated list of strings is copied to the initial contents of parameter 0 and the `<string pattern>` in the filter expression is replaced with the parameter `%0`.

The initial matching string list is converted to an explicit parameter value so that subsequent additions and deletions of string values to and from the list of matching strings may be performed with the `append_to_expression_parameter()` and `remove_from_expression_parameter()` operations mentioned above.

5.4.7.1 Example STRINGMATCH Filter Expressions

This expression evaluates to TRUE if the value of `symbol` is equal to `NASDAQ/MSFT`:

```
symbol MATCH 'NASDAQ/MSFT'
```

This expression evaluates to TRUE if the value of `symbol` is equal to `NASDAQ/IBM` or `NASDAQ/MSFT`:

```
symbol MATCH 'NASDAQ/IBM,NASDAQ/MSFT'
```

This expression evaluates to TRUE if the value of `symbol` corresponds to `NASDAQ` and starts with a letter between M and Y:

```
symbol MATCH 'NASDAQ/[M-Y]*'
```

5.4.7.2 STRINGMATCH Filter Expression Parameters

In the builtin STRINGMATCH filter, there is one, and only one, parameter: parameter 0. (If you want to add more parameters, see [Appending a String to an Expression Parameter \(Section 5.4.5.3 on page 220\)](#).) The parameter can be specified explicitly using the same syntax as the SQL filter or implicitly by using a constant string pattern. For example:

```
symbol MATCH %0 (Explicit parameter)
```

```
symbol MATCH 'IBM' (Implicit parameter initialized to IBM)
```

Strings used as parameter values must contain the enclosing quotation marks (') within the parameter value; do not place the quotation marks within the expression statement. For example, the expression " symbol MATCH %0 " with parameter 0 set to " 'IBM' " is legal, whereas the expression " symbol MATCH '%0' " with parameter 0 set to " IBM " will not compile.

5.4.8 Custom Content Filters

By default, a `ContentFilteredTopic` will use a SQL-like content filter, `DDS_SQLFILTER_NAME` (see [SQL Filter Expression Notation \(Section 5.4.6 on page 222\)](#)), which implements a superset of the content filter. There is another builtin filter, `DDS_STRINGMATCHFILTER_NAME` (see [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#)). Both of these are automatically registered.

If you want to use a different filter, **you must register it first**, then create the `ContentFilteredTopic` using `create_contentfilteredtopic_with_filter()` (see [Creating ContentFilteredTopics \(Section 5.4.3 on page 214\)](#)).

One reason to use a custom filter is that the default filter can only filter based on relational operations between topic members, not on a computation involving topic members. For example, if you want to filter based on the sum of the members, you must create your own filter.

Notes:

- The API for using a custom content filter is subject to change in a future release.
- Custom content filters are not supported when using the .NET APIs

5.4.8.1 Filtering on the Writer Side with Custom Filters

There are two approaches for performing writer-side filtering. The first approach is to evaluate each written DDS sample against filters of all the readers that have content filter specified and identify the readers whose filter passes the DDS sample.

The second approach is to evaluate the written DDS sample once for the writer and then rely on the filter implementation to provide a set of readers whose filter passes the DDS sample. This approach allows the filter implementation to cache the result of filtering, if possible. For example, consider a scenario where the data is described by the struct shown below, where $10 < x < 20$:

```
struct MyData {
    int x;
    int y;
};
```

If the filter expression is based only on the `x` field, the filter implementation can maintain a hash map for all the different values of `x` and cache the filtering results in the hash map. Then any future evaluations will only be $O(1)$, because it only requires a lookup in the hash map.

But if in the same example, a reader has a content filter that is based on both x and y , or just y , the filter implementation cannot cache the result—because the filter was only maintaining a hash map for x . In this case, the filter implementation can inform Connex DDS that it will not be caching the result for those *DataReaders*. The filter can use `DDS_ExpressionProperty` to indicate to the middleware whether or not it will cache the results for *DataReader*. [Table 5.8 DDS_ExpressionProperty](#) describes `DDS_ExpressionProperty`.

Table 5.8 DDS_ExpressionProperty

Type	Field Name	Description
DDS_Boolean	key_only_filter	Indicates if the filter expression is based only on key fields. In this case, Connex DDS itself can cache the filtering results.
DDS_Boolean	writer_side_filter_optimization	Indicates if the filter implementation can cache the filtering result for the expression provided. If this is true then Connex DDS will do no caching or explicit filter evaluation for the associated <i>DataReader</i> . It will instead rely on the filter implementation to provide appropriate results.

5.4.8.2 Registering a Custom Filter

To use a custom filter, it must be registered in the following places:

- Register the custom filter in any subscribing application in which the filter is used to create a `ContentFilteredTopic` and corresponding *DataReader*.
- In each publishing application, you only need to register the custom filter if you want to perform writer-side filtering. A *DataWriter* created with an associated filter will use that filter if it discovers a matched *DataReader* that uses the same filter.

For example, suppose Application A on the subscription side creates a *Topic* named **X** and a `ContentFilteredTopic` named **filteredX** (and a corresponding *DataReader*), using a previously registered content filter, **myFilter**. With only that, you will have filtering on the subscription side. If you also want to perform filtering in any application that publishes *Topic X*, then you also need to register the same definition of the `ContentFilter` **myFilter** in that application.

To register a new filter, use the *DomainParticipant's* `register_contentfilter()` operation¹:

```
DDS_ReturnCode_t register_contentfilter(
    const char * filter_name,
    const DDSContentFilter * contentfilter)
```

¹This operation is an extension to the DDS standard.

- **filter_name**

The name of the filter. The name must be unique within the *DomainParticipant*. The **filter_name** cannot have a length of 0. The same filtering functions and handle can be registered under different names.

- **content_filter**

This class specifies the functions that will be used to process the filter.

You must derive from the *DDSContentFilter* base class and implement the virtual [compile \(Section below\)](#), [evaluate \(Section below\)](#), and [finalize \(Section below\)](#) functions described below.

Optionally, you can derive from the *DDSWriterContentFilter* base class instead, to implement additional filtering operations that will be used by the *DataWriter*. When performing writer-side filtering, these operations allow a DDS sample to be evaluated once for the *DataWriter*, instead of evaluating the DDS sample for every *DataReader* that is matched with the *DataWriter*. An instance of the derived class is then used as an argument when calling **register_contentfilter()**.

- **compile**

The function that will be used to compile a filter expression and parameters. Connex DDS will call this function when a *ContentFilteredTopic* is created and when the filter parameters are changed. This parameter cannot be NULL. See [Compile Function \(Section 5.4.8.5 on page 237\)](#). This is a member of *DDSContentFilter* and *DDSWriterContentFilter*.

- **evaluate**

The function that will be called by Connex DDS each time a DDS sample is received. Its purpose is to evaluate the DDS sample based on the filter. This parameter cannot be NULL. See [Evaluate Function \(Section 5.4.8.6 on page 238\)](#). This is a member of *DDSContentFilter* and *DDSWriterContentFilter*.

- **finalize**

The function that will be called by Connex DDS when an instance of the custom content filter is no longer needed. This parameter may be NULL. See [Finalize Function \(Section 5.4.8.7 on page 239\)](#). This is a member of *DDSContentFilter* and *DDSWriterContentFilter*.

- **writer_attach**

The function that will be used to create some state required to perform filtering on the writer side using the operations provided in *DDSWriterContentFilter*. Connex DDS will call this function for every *DataWriter*; it will be called only the *first time* the *DataWriter* matches a *DataReader* using the specified filter. This function will not be called for any subsequent *DataReaders* that match the *DataWriter* and are using the same filter. See [Writer Attach Function \(Section 5.4.8.8 on page 239\)](#). This is a member of *DDSWriterContentFilter*.

- **writer_detach**

The function that will be used to delete any state created using the `writer_attach` function. Connex DDS will call this function when the *DataWriter* is deleted. See [Writer Detach Function \(Section 5.4.8.9 on page 239\)](#). This is a member of `DDSWriterContentFilter`.

- **writer_compile**

The function that will be used by the *DataWriter* to compile filter expression and parameters provided by the reader. Connex DDS will call this function when the *DataWriter* discovers a *DataReader* with a `ContentFilteredTopic` or when a *DataWriter* is notified of a change in *DataReader*'s filter parameter. This function will receive as an input a `DDS_Cookie_t` which uniquely identifies the *DataReader* for which the function was invoked. See [Writer Compile Function \(Section 5.4.8.10 on page 239\)](#). This is a member of `DDSWriterContentFilter`.

- **writer_evaluate**

The function that will be called by Connex DDS every time a *DataWriter* writes a new DDS sample. Its purpose is to evaluate the DDS sample for all the readers for which the *DataWriter* is performing writer-side filtering and return the list of `DDS_Cookie_t` associated with the *DataReaders* whose filter pass the DDS sample. See [Writer Evaluate Function \(Section 5.4.8.11 on page 240\)](#).

- **writer_return_loan**

The function that will be called by Connex DDS to return the loan on a sequence of `DDS_Cookie_t` provided by the `writer_evaluate` function. See [Writer Return Loan Function \(Section 5.4.8.12 on page 241\)](#). This is a member of `DDSWriterContentFilter`.

- **writer_finalize**

The function that will be called by Connex DDS to notify the filter implementation that the *DataWriter* is no longer matching with a *DataReader* for which it was previously performing writer-side filtering. This will allow the filter to purge any state it was maintaining for the *DataReader*. See [Writer Finalize Function \(Section 5.4.8.13 on page 241\)](#). This is a member of `DDSWriterContentFilter`.

5.4.8.3 Unregistering a Custom Filter

To unregister a filter, use the *DomainParticipant*'s `unregister_contentfilter()` operation¹, which is useful if you want to reuse a particular filter name. (Note: You do not have to unregister the filter before deleting the parent *DomainParticipant*. If you do not need to reuse the filter name to register another filter, there is no reason to unregister the filter.)

```
DDS_ReturnCode_t unregister_contentfilter(const char * filter_name)
```

¹This operation is an extension to the DDS standard.

filter_name The name of the previously registered filter. The name must be unique within the *DomainParticipant*. The **filter_name** cannot have a length of 0.

If you attempt to unregister a filter that is still being used by a *ContentFilteredTopic*, **unregister_contentfilter()** will return **PRECONDITION_NOT_MET**.

If there are still existing discovered *DataReaders* with the same **filter_name** and the filter's **compile** function has previously been called on the discovered *DataReaders*, the filter's **finalize** function will be called on those discovered *DataReaders* before the content filter is unregistered. This means filtering will be performed on the application that is creating the *DataReader*.

5.4.8.4 Retrieving a ContentFilter

If you know the name of a *ContentFilter*, you can get a pointer to its structure. If the *ContentFilter* has not already been registered, this operation will return **NULL**.

```
DDS_ContentFilter *lookup_contentfilter (const char * filter_name)
```

5.4.8.5 Compile Function

The **compile** function specified in the *ContentFilter* will be used to compile a filter expression and parameters. Please note that the term ‘compile’ is intentionally defined very broadly. It is entirely up to you, as the user, to decide what this function should do. The only requirement is that the **error_code** parameter passed to the compile function must return **OK** on successful execution. For example:

```
DDS_ReturnCode_t sample_compile_function(
    void ** new_compile_data, const char * expression,
    const DDS_StringSeq & parameters,
    const DDS_TypeCode * type_code,
    const char * type_class_name,
    void * old_compile_data)
{
    *new_compile_data = (void*)DDS_String_dup(parameters[0]);
    return DDS_RETCODE_OK;
}
```

Where:

new_compile_data A user-specified opaque pointer of this instance of the content filter. This value is passed to the **evaluate** and **finalize** functions

expression An ASCIIZ string with the filter expression the *ContentFilteredTopic* was created with. Note that the memory used by the parameter pointer is owned by Connex DDS. If you want to manipulate this string, you *must* make a copy of it first. Do not free the memory for this string.

- parameters** A string sequence of expression parameters used to create the ContentFilteredTopic. The string sequence is equal (but not identical) to the string sequence passed to **create_contentfilteredtopic()** (see **expression_parameters** in [Creating ContentFilteredTopics \(Section 5.4.3 on page 214\)](#)).
- The sequence passed to the **compile** function is owned by Connex DDS and must not be referred to outside the **compile** function.
- type_code** A pointer to the type code of the related *Topic*. A type code is a description of the topic members, such as their type (long, octet, etc.), but does not contain any information with respect to the memory layout of the structures. The type code can be used to write filters that can be used with any type. See [Using Generated Types without Connex DDS \(Standalone\) \(Section 3.7 on page 139\)](#). [Note: If you are using the Java API, this parameter will always be NULL.]
- type_class_name** Fully qualified class name of the related *Topic*.
- old_compile_data** The **new_compile_data** value from a *previous* call to this instance of a content filter. If **compile** is called more than once for an instance of a ContentFilteredTopic (such as if the expression parameters are changed), then the **new_compile_data** value returned by the previous invocation is passed in the **old_compile_data** parameter (which can be NULL). If this is a new instance of the filter, NULL is passed. This parameter is useful for freeing or reusing previously allocated resources.

5.4.8.6 Evaluate Function

The **evaluate** function specified in the ContentFilter will be called each time a DDS sample is received. This function's purpose is to determine if a DDS sample should be filtered out (not put in the receive queue).

For example:

```
DDS_Boolean sample_evaluate_function(
    void* compile_data,
    const void* sample,
    struct DDS_FilterSampleInfo * meta_data) {
    char *parameter = (char*)compile_data;
    DDS_Long x;
    Foo *foo_sample = (Foo*)sample;
    sscanf(parameter, "%d", &x);
    return (foo_sample->x > x ? DDS_BOOLEAN_FALSE : DDS_BOOLEAN_TRUE);
}
```

The function may use the following parameters:

- compile_data** The last return value from the **compile** function for this instance of the content filter. Can be NULL.
- sample** A pointer to a C structure with the data to filter. Note that the **evaluate** function always receives *deserialized* data.
- meta_data** A pointer to the meta data associated with the DDS sample.

Note: Currently the `meta_data` field only supports `related_sample_identity` (described in [Table 6.16 DDS_WriteParams_t](#)).

5.4.8.7 Finalize Function

The **finalize** function specified in the `ContentFilter` will be called when an instance of the custom content filter is no longer needed. When this function is called, it is safe to free all resources used by this particular instance of the custom content filter.

For example:

```
void sample_finalize_function ( void* compile_data) {
    /* free parameter string from compile function */
    DDS_String_free((char *)compile_data);
}
```

The **finalize** function may use the following optional parameters:

- system_key** See [Compile Function \(Section 5.4.8.5 on page 237\)](#).
- handle** This is the opaque returned by the last call to the **compile** function.

5.4.8.8 Writer Attach Function

The **writer_attach** function specified in the `WriterContentFilter` will be used to create some state that can be used by the filter to perform writer-side filtering more efficiently. It is entirely up to you, as the implementer of the filter, to decide if the filter requires this state.

The function has the following parameter:

- writer_filter_data** A user-specified opaque pointer to some state created on the writer side that will help perform writer-side filtering efficiently.

5.4.8.9 Writer Detach Function

The **writer_detach** function specified in the `WriterContentFilter` will be used to free up any state that was created using the **writer_attach** function.

The function has the following parameter:

- `writer_filter_data` A pointer to the state created using the **writer_attach** function.

5.4.8.10 Writer Compile Function

The **writer_compile** function specified in the `WriterContentFilter` will be used by a *DataWriter* to compile a filter expression and parameters associated with a *DataReader* for which the *DataWriter* is performing fil-

tering. The function will receive as input a **DDS_Cookie_t** that uniquely identifies the *DataReader* for which the function was invoked.

The function has the following parameters:

writer_filter_data	A pointer to the state created using the writer_attach function.
prop	A pointer to <code>DDS_ExpressionProperty</code> . This is an output parameter. It allows you to indicate to Connex DDS if a filter expression can be optimized (as described in Filtering on the Writer Side with Custom Filters (Section 5.4.8.1 on page 233)).
expression	An ASCIIZ string with the filter expression the <code>ContentFilteredTopic</code> was created with. Note that the memory used by the parameter pointer is owned by Connex DDS. If you want to manipulate this string, you must make a copy of it first. Do not free the memory for this string.
parameters	A string sequence of expression parameters used to create the <code>ContentFilteredTopic</code> . The string sequence is equal (but not identical) to the string sequence passed to create_contentfilteredtopic() (see expression_parameters in Creating ContentFilteredTopics (Section 5.4.3 on page 214)). The sequence passed to the compile function is owned by Connex DDS and must not be referred to outside the writer_compile function.
type_code	A pointer to the type code of the related <code>Topic</code> . A type code is a description of the topic members, such as their type (long, octet, etc.), but does not contain any information with respect to the memory layout of the structures. The type code can be used to write filters that can be used with any type. See Using Generated Types without Connex DDS (Standalone) (Section 3.7 on page 139) . [Note: If you are using the Java API, this parameter will always be NULL.]
type_class_name	The fully qualified class name of the related <code>Topic</code> .
cookie	A DDS_Cookie_t to uniquely identify the <i>DataReader</i> for which the writer_compile function was called.

5.4.8.11 Writer Evaluate Function

The **writer_evaluate** function specified in the `WriterContentFilter` will be used by a *DataWriter* to retrieve the list of *DataReaders* whose filter passed the DDS sample. The **writer_evaluate** function returns a sequence of cookies which identifies the set of *DataReaders* whose filter passes the DDS sample.

The function has the following parameters:

writer_filter_data	A pointer to the state created using the writer_attach function.
sample	A pointer to the data to be filtered. Note that the writer_evaluate function always receives <i>deserialized</i> data.

meta_data A pointer to the meta-data associated with the DDS sample.

Note: Currently the **meta_data** field only supports **related_sample_identity** (described in [Table 6.16 DDS_WriteParams_t](#)).

5.4.8.12 Writer Return Loan Function

Connex DDS uses the **writer_return_loan** function specified in the `WriterContentFilter` to indicate to the filter implementation that it has finished using the sequence of cookies returned by the filter's **writer_evaluate** function. Your filter implementation should *not* free the memory associated with the cookie sequence before the **writer_return_loan** function is called.

The function has the following parameters:

writer_filter_data A pointer to the state created using the **writer_attach** function.

cookies The sequence of cookies for which the **writer_return_loan** function was called.

5.4.8.13 Writer Finalize Function

The **writer_finalize** function specified in the `WriterContentFilter` will be called when the *DataWriter* no longer matches with a *DataReader* that was created with `ContentFilteredTopic`. This will allow the filter implementation to delete any state it was maintaining for the *DataReader*.

The function has the following parameters:

writer_filter_data A pointer to the state created using the **writer_attach** function.

cookie A `DDS_Cookie_t` to uniquely identify the *DataReader* for which the `writer_finalize` was called.

Chapter 6 Sending Data

This section discusses how to create, configure, and use *Publishers* and *DataWriters* to send data. It describes how these *Entities* interact, as well as the types of operations that are available for them.

This section includes the following sections:

The goal of this section is to help you become familiar with the *Entities* you need for sending data. For up-to-date details such as formal parameters and return codes on any mentioned operations, please see the API Reference HTML documentation.

6.1 Preview: Steps to Sending Data

To send DDS samples of a data instance:

1. Create and configure the required *Entities*:
 - a. Create a *DomainParticipant* (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#)).
 - b. Register user data types¹ with the *DomainParticipant*. For example, the ‘**FooDataType**’. (This step is not necessary in the Modern C++ API--the Topic instantiation automatically registers the type)
 - c. Use the *DomainParticipant* to create a *Topic* with the registered data type.
 - d. Optionally², use the *DomainParticipant* to create a *Publisher*.

¹Type registration is not required for built-in types (see [Registering Built-in Types \(Section 3.2.1 on page 30\)](#)).

²You are not required to explicitly create a *Publisher*; instead, you can use the 'implicit *Publisher*' created from the *DomainParticipant*. See [Creating Publishers Explicitly vs. Implicitly \(Section 6.2.1 on page 248\)](#).

- e. Use the *Publisher* or *DomainParticipant* to create a *DataWriter* for the *Topic*.
 - f. Use a type-safe method to cast the generic *DataWriter* created by the *Publisher* to a type-specific *DataWriter*. For example, ‘**FooDataWriter**’. (This step doesn't apply to the Modern C++ API where you directly instantiate a type-safe ‘**DataWriter<Foo>**’.)
 - g. Optionally, register data instances with the *DataWriter*. If the *Topic*'s user data type contain *key* fields, then registering a data *instance* (data with a specific key value) will improve performance when repeatedly sending data with the same key. You may register many different data instances; each registration will return an *instance handle* corresponding to the specific key value. For non-keyed data types, instance registration has no effect. See [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#) for more information on keyed data types and instances.
2. Every time there is changed data to be published:
 - a. Store the data in a variable of the correct data type (for instance, variable ‘**Foo**’ of the type ‘**FooDataType**’).
 - b. Call the **FooDataWriter**'s **write()** operation, passing it a reference to the variable ‘**Foo**’.
 - For non-keyed data types or for non-registered instances, also pass in **DDS_HANDLE_NIL**.
 - For keyed data types, pass in the instance handle corresponding to the instance stored in ‘**Foo**’, if you have registered the instance previously. This means that the data stored in ‘**Foo**’ has the same key value that was used to create instance handle.
 - c. The **write()** function will take a snapshot of the contents of ‘**Foo**’ and store it in Connex DDS internal buffers from where the DDS data sample is sent under the criteria set by the *Publisher's* and *DataWriter's* QosPolicies. If there are matched *DataReaders*, then the DDS data sample will have been passed to the physical transport plug-in/device driver by the time that **write()** returns.

6.2 Publishers

An application that intends to publish information needs the following *Entities*: *DomainParticipant*, *Topic*, *Publisher*, and *DataWriter*. All *Entities* have a corresponding specialized *Listener* and a set of QosPolicies. A *Listener* is how Connex DDS notifies your application of status changes relevant to the Entity. The QosPolicies allow your application to configure the behavior and resources of the Entity.

- A *DomainParticipant* defines the DDS domain in which the information will be made available.
- A *Topic* defines the name under which the data will be published, as well as the type (format) of the data itself.
- An application writes data using a *DataWriter*. The *DataWriter* is bound at creation time to a *Topic*, thus specifying the name under which the *DataWriter* will publish the data and the type associated

with the data. The application uses the *DataWriter*'s **write()** operation to indicate that a new value of the data is available for dissemination.

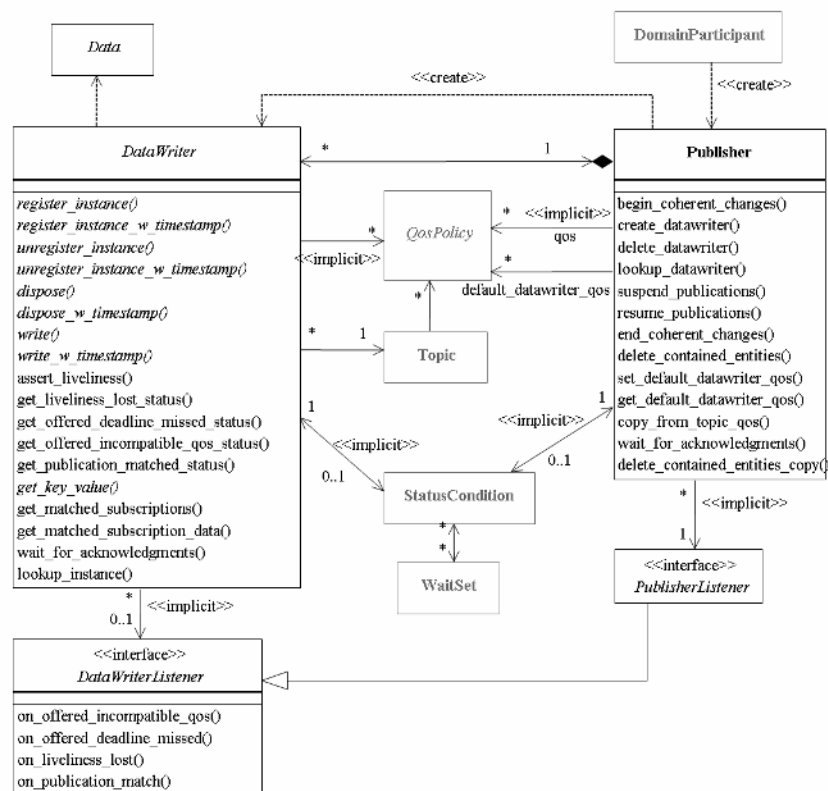
- A *Publisher* manages the activities of several *DataWriters*. The *Publisher* determines when the data is actually sent to other applications. Depending on the settings of various QoS Policies of the *Publisher* and *DataWriter*, data may be buffered to be sent with the data of other *DataWriters* or not sent at all. By default, the data is sent as soon as the *DataWriter*'s **write()** function is called.

You may have multiple *Publishers*, each managing a different set of *DataWriters*, or you may choose to use one *Publisher* for all your *DataWriters*.

For more information, see [Creating Publishers Explicitly vs. Implicitly \(Section 6.2.1 on page 248\)](#).

Figure 6.1 Publication Module below shows how these *Entities* are related, as well as the methods defined for each *Entity*.

Figure 6.1 Publication Module



Publishers are used to perform the operations listed in [Table 6.1 Publisher Operations on the next page](#). You can find more information about the operations by looking in the section listed under the **Reference**

column. For details such as formal parameters and return codes, please see the API Reference HTML documentation.

Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

Table 6.1 Publisher Operations

Working with ...	Operation	Description	Reference
DataWriters	begin_coherent_changes	Indicates that the application will begin a coherent set of modifications.	Writing Coherent Sets of DDS Data Samples (Section 6.3.10 on page 287)
	create_datawriter	Creates a <i>DataWriter</i> that will belong to the <i>Publisher</i> .	Creating DataWriters (Section 6.3.1 on page 266)
	create_datawriter_with_profile	Sets the <i>DataWriter</i> 's QoS based on a specified QoS profile.	
	copy_from_topic_qos	Copies relevant QoS policies from a <i>Topic</i> into a <i>DataWriterQoS</i> structure.	Other Publisher QoS-Related Operations (Section 6.2.4.6 on page 257)
DataWriters cont'd	delete_contained_entities	Deletes all of the <i>DataWriters</i> that were created by the <i>Publisher</i> .	Deleting Contained DataWriters (Section 6.2.3.1 on page 251)
	delete_datawriter	Deletes a <i>DataWriter</i> that belongs to the <i>Publisher</i> .	Deleting DataWriters (Section 6.3.3 on page 268)
	end_coherent_changes	Ends the coherent set initiated by <code>begin_coherent_changes()</code> .	Writing Coherent Sets of DDS Data Samples (Section 6.3.10 on page 287)

Table 6.1 Publisher Operations

Working with ...	Operation	Description	Reference
DataWriters cont'd	get_all_datawriters	Retrieves all the <i>DataWriters</i> created from this <i>Publisher</i> .	Getting All DataWriters (Section 6.3.2 on page 268)
	get_default_datawriter_qos	Copies the <i>Publisher</i> 's default <i>DataWriterQoS</i> values into a <i>DataWriterQoS</i> structure.	Setting DataWriter QoS Policies (Section 6.3.15 on page 300)
	get_status_changes	Will always return 0 since there are no <i>Statuses</i> currently defined for <i>Publishers</i> .	Getting Status and Status Changes (Section 4.1.4 on page 157)
	lookup_datawriter	Retrieves a <i>DataWriter</i> previously created for a specific <i>Topic</i> .	Finding a Publisher's Related DDS Entities (Section 6.2.6 on page 259)
DataWriters cont'd	set_default_datawriter_qos	Sets or changes the default <i>DataWriterQoS</i> values.	Getting and Setting Default QoS for DataWriters (Section 6.2.4.5 on page 256)
	set_default_datawriter_qos_with_profile	Sets or changes the default <i>DataWriterQoS</i> values based on a <i>QoS</i> profile.	
	wait_for_acknowledgments	Blocks until all data written by the <i>Publisher</i> 's reliable <i>DataWriters</i> are acknowledged by all matched reliable <i>DataReaders</i> , or until the a specified timeout duration, <i>max_wait</i> , elapses.	Waiting for Acknowledgments in a Publisher (Section 6.2.7 on page 260)
	is_sample_app_acknowledged	Indicates if a sample has been application-acknowledged by all the matching <i>DataReaders</i> that were alive when the sample was written. If a <i>DataReader</i> does not enable application acknowledgment (by setting the <i>ReliabilityQoSPolicy</i> 's <i>acknowledgment_kind</i> to a value other than <i>DDS_PROTOCOL_ACKNOWLEDGMENT_MODE</i>), the sample is considered application-acknowledged for that <i>DataReader</i> .	Application Acknowledgment (Section 6.3.12 on page 288)

Table 6.1 Publisher Operations

Working with ...	Operation	Description	Reference
Libraries and Profiles	get_default_library	Gets the <i>Publisher's</i> default QoS profile library.	Getting and Setting the Publisher's Default QoS Profile and Library (Section 6.2.4.4 on page 255)
	get_default_profile	Gets the <i>Publisher's</i> default QoS profile.	
	get_default_profile_library	Gets the library that contains the <i>Publisher's</i> default QoS profile.	
	set_default_library	Sets the default library for a <i>Publisher</i> .	
	set_default_profile	Sets the default profile for a <i>Publisher</i> .	
Participants	get_participant	Gets the <i>DomainParticipant</i> that was used to create the <i>Publisher</i> .	Finding a Publisher's Related DDS Entities (Section 6.2.6 on page 259)
Publishers	enable	Enables the <i>Publisher</i> .	Enabling DDS Entities (Section 4.1.2 on page 154)
	equals	Compares two <i>Publisher's</i> QoS structures for equality.	Comparing QoS Values (Section 6.2.4.2 on page 254)
	get_qos	Gets the <i>Publisher's</i> current QoSPolicy settings. This is most often used in preparation for calling set_qos().	Setting Publisher QoS Policies (Section 6.2.4 on page 251)
	set_qos	Sets the <i>Publisher's</i> QoS. You can use this operation to change the values for the <i>Publisher's</i> QoS Policies. Note, however, that not all QoS Policies can be changed after the <i>Publisher</i> has been created.	
	set_qos_with_profile	Sets the <i>Publisher's</i> QoS based on a specified QoS profile.	

Table 6.1 Publisher Operations

Working with ...	Operation	Description	Reference
Publishers cont'd	get_listener	Gets the currently installed Listener.	Setting Up PublisherListeners (Section 6.2.5 on page 257)
	set_listener	Sets the <i>Publisher's</i> Listener. If you created the <i>Publisher</i> without a Listener, you can use this operation to add one later.	
	suspend_publications	Provides a <i>hint</i> that multiple data-objects within the <i>Publisher</i> are about to be written. Connex DDS does not currently use this hint.	Suspending and Resuming Publications (Section 6.2.9 on page 261)
	resume_publications	Reverses the action of <code>suspend_publications()</code> .	

6.2.1 Creating Publishers Explicitly vs. Implicitly

To send data, your application must have a Publisher. However, you are not required to explicitly create one. If you do not create one, the middleware will implicitly create a *Publisher* the first time you create a *DataWriter* using the *DomainParticipant's* operations. It will be created with default QoS (DDS_PUBLISHER_QOS_DEFAULT) and no Listener.

A *Publisher* (implicit or explicit) gets its own default QoS and the default QoS for its child *DataWriters* from the *DomainParticipant*. These default QoS are set when the *Publisher* is created. (This is true for *Subscribers* and *DataReaders*, too.)

The 'implicit *Publisher*' can be accessed using the *DomainParticipant's* `get_implicit_publisher()` operation (see [Getting the Implicit Publisher or Subscriber \(Section 8.3.9 on page 569\)](#)). You can use this 'implicit *Publisher*' just like any other *Publisher* (it has the same operations, QoS Policies, etc.). So you can change the mutable QoS and set a Listener if desired.

DataWriters are created by calling `create_datawriter()` or `create_datawriter_with_profile()`—these operations exist for *DomainParticipants* and *Publishers*. If you use the *DomainParticipant* to create a *DataWriter*, it will belong to the implicit *Publisher*. If you use a *Publisher* to create a *DataWriter*, it will belong to that *Publisher*.

The middleware will use the same implicit *Publisher* for all *DataWriters* that are created using the *DomainParticipant's* operations.

Having the middleware implicitly create a *Publisher* allows you to skip the step of creating a *Publisher*. However, having all your *DataWriters* belong to the same *Publisher* can reduce the concurrency of the system because all the write operations will be serialized.

6.2.2 Creating Publishers

Before you can explicitly create a *Publisher*, you need a *DomainParticipant* (see [DomainParticipants \(Section 8.3 on page 547\)](#)). To create a *Publisher*, use the *DomainParticipant*'s `create_publisher()` or `create_publisher_with_profile()` operations.

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Note: The Modern C++ API Publishers provide constructors whose first and only required argument is the *DomainParticipant*.

```
DDSPublisher * create_publisher (
    const DDS_PublisherQos &qos,
    DDSPublisherListener *listener,
    DDS_StatusMask mask)
DDSPublisher * create_publisher_with_profile (
    const char *library_name,
    const char *profile_name,
    DDSPublisherListener *listener,
    DDS_StatusMask mask)
```

Where:

- qos** If you want the default QoS settings (described in the API Reference HTML documentation), use `DDS_PUBLISHER_QOS_DEFAULT` for this parameter (see [Creating a Publisher with Default QoS Policies \(Section Figure 6.2 on the facing page\)](#)).
- If you want to customize any of the QoS Policies, supply a QoS structure (see [Creating a Publisher with Non-Default QoS Policies \(not from a profile\) \(Section Figure 6.3 on page 253\)](#)). The QoS structure for a *Publisher* is described in [Publisher/Subscriber QoS Policies \(Section 6.4 on page 312\)](#).
- Note:** If you use `DDS_PUBLISHER_QOS_DEFAULT`, it is not safe to create the *Publisher* while another thread may be simultaneously calling `set_default_publisher_qos()`.
- listener** *Listeners* are callback routines. Connex DDS uses them to notify your application when specific events (status changes) occur with respect to the *Publisher* or the *DataWriters* created by the *Publisher*.
- The *listener* parameter may be set to NULL if you do not want to install a *Listener*. If you use NULL, the *Listener* of the *DomainParticipant* to which the *Publisher* belongs will be used instead (if it is set). For more information on *PublisherListeners*, see [Setting Up PublisherListeners \(Section 6.2.5 on page 257\)](#).
- mask** This bit-mask indicates which status changes will cause the *Publisher*'s *Listener* to be invoked. The bits set in the mask must have corresponding callbacks implemented in the *Listener*.

If you use NULL for the *Listener*, use `DDS_STATUS_MASK_NONE` for this parameter. If the *Listener* implements all callbacks, use `DDS_STATUS_MASK_ALL`. For information on statuses, see [Listeners \(Section 4.4 on page 177\)](#).

library_name A QoS Library is a named set of QoS profiles. See [URL Groups \(Section 17.8 on page 814\)](#). If NULL is used for **library_name**, the *DomainParticipant's* default library is assumed (see [Getting and Setting the Publisher's Default QoS Profile and Library \(Section 6.2.4.4 on page 255\)](#)).

profile_name A QoS profile groups a set of related QoS, usually one per entity. See [URL Groups \(Section 17.8 on page 814\)](#). If NULL is used for **profile_name**, the *DomainParticipant's* default profile is assumed and **library_name** is ignored

Figure 6.2 Creating a Publisher with Default QoS Policies

```
// create the publisher
DDSPublisher* publisher =
    participant->create_publisher(
        DDS_PUBLISHER_QOS_DEFAULT,
        NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
};
```

For more examples, see [Configuring QoS Settings when the Publisher is Created \(Section 6.2.4.1 on page 252\)](#).

After you create a *Publisher*, the next step is to use the *Publisher* to create a *DataWriter* for each *Topic*, see [Creating DataWriters \(Section 6.3.1 on page 266\)](#). For a list of operations you can perform with a *Publisher*, see [Table 6.1 Publisher Operations](#).

6.2.3 Deleting Publishers

(Note: in the Modern C++ API, *Entities* are automatically destroyed, see [Creating and Deleting DDS Entities \(Section 4.1.1 on page 153\)](#))

This section applies to both implicitly and explicitly created *Publishers*.

To delete a *Publisher*:

1. You must first delete all *DataWriters* that were created with the *Publisher*. Use the *Publisher's* `delete_datawriter()` operation to delete them one at a time, or use the `delete_contained_entities()` operation ([Deleting Contained DataWriters \(Section 6.2.3.1 on the next page\)](#)) to delete them all at the same time.

```
DDS_ReturnCode_t delete_datawriter (DDSDataWriter *a_datawriter)
```

2. Delete the *Publisher* by using the *DomainParticipant*'s `delete_publisher()` operation.

```
DDS_ReturnCode_t delete_publisher (DDSPublisher *p)
```

Note: A *Publisher* cannot be deleted within a *Listener* callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

6.2.3.1 Deleting Contained DataWriters

The *Publisher*'s `delete_contained_entities()` operation deletes all the *DataWriters* that were created by the *Publisher*.

```
DDS_ReturnCode_t delete_contained_entities ()
```

After this operation returns successfully, the application may delete the *Publisher* (see [Deleting Publishers \(Section 6.2.3 on the previous page\)](#)).

6.2.4 Setting Publisher QoS Policies

A *Publisher*'s QoS Policies control its behavior. Think of the policies as the configuration and behavior 'properties' of the *Publisher*. The **DDS_PublisherQos** structure has the following format:

```
DDS_PublisherQos struct {
    DDS_PresentationQosPolicy presentation;
    DDS_PartitionQosPolicy partition;
    DDS_GroupDataQosPolicy group_data;
    DDS_EntityFactoryQosPolicy entity_factory;
    DDS_AsynchronousPublisherQosPolicy asynchronous_publisher;
    DDS_ExclusiveAreaQosPolicy exclusive_area;
    DDS_EntityNameQosPolicy publisher_name;
} DDS_PublisherQos;
```

Note: `set_qos()` cannot always be used in a listener callback; see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

[Table 6.2 Publisher QoS Policies](#) summarizes the meaning of each policy. (They appear alphabetically in the table.) For information on *why* you would want to change a particular QoS Policy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation for each policy.

Table 6.2 Publisher QoS Policies

QoS Policy	Description
ASYNCHRONOUS_PUBLISHER QoS Policy (DDS Extension) (Section 6.4.1 on page 313)	Configures the mechanism that sends user data in an external middleware thread.
ENTITYFACTORY QoS Policy (Section 6.4.2 on page 315)	Controls whether or not child <i>Entities</i> are created in the enabled state.
ENTITY_NAME QoS Policy (DDS Extension) (Section 6.5.9 on page 374)	Assigns a name and role_name to a <i>Publisher</i> .
EXCLUSIVE_AREA QoS Policy (DDS Extension) (Section 6.4.3 on page 318)	Configures multi-thread concurrency and deadlock prevention capabilities.
GROUP_DATA QoS Policy (Section 6.4.4 on page 320)	Along with TOPIC_DATA QoS Policy (Section 5.2.1 on page 209) and USER_DATA QoS Policy (Section 6.5.26 on page 417) , this QoS Policy is used to attach a buffer of bytes to Connex DDS's discovery meta-data.
PARTITION QoS Policy (Section 6.4.5 on page 323)	Adds string identifiers that are used for matching <i>DataReaders</i> and <i>DataWriters</i> for the same <i>Topic</i> .
PRESENTATION QoS Policy (Section 6.4.6 on page 330)	Controls how Connex DDS presents data received by an application to the <i>DataReaders</i> of the data.

6.2.4.1 Configuring QoS Settings when the Publisher is Created

As described in [Creating Publishers \(Section 6.2.2 on page 249\)](#), there are different ways to create a *Publisher*, depending on how you want to specify its QoS (with or without a QoS Profile).

- In [Creating a Publisher with Default QoS Policies \(Section Figure 6.2 on page 250\)](#) we saw an example of how to explicitly create a *Publisher* with default QoS Policies. It used the special constant, **DDS_PUBLISHER_QOS_DEFAULT**, which indicates that the default QoS values for a *Publisher* should be used. Default *Publisher* QoS Policies are configured in the *DomainParticipant*; you can change them with the *DomainParticipant*'s **set_default_publisher_qos()** or **set_default_publisher_qos_with_profile()** operation (see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on page 568\)](#)).
- To create a *Publisher* with non-default QoS settings, without using a QoS profile, see [Figure 6.3 Creating a Publisher with Non-Default QoS Policies \(not from a profile\) on the next page](#). It uses the *DomainParticipant*'s **get_default_publisher_qos()** method to initialize a **DDS_PublisherQos** structure. Then the policies are modified from their default values before the QoS structure is passed to **create_publisher()**.

- You can also create a *Publisher* and specify its QoS settings via a QoS Profile. To do so, call **create_publisher_with_profile()**, as seen in [Figure 6.4 Creating a Publisher with a QoS Profile on the next page](#).
- If you want to use a QoS profile, but then make some changes to the QoS before creating the *Publisher*, call the DomainParticipantFactory's **get_publisher_qos_from_profile()**, modify the QoS and use the modified QoS structure when calling **create_publisher()**, as seen in [Figure 6.5 Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values on the facing page](#).

For more information, see [Creating Publishers \(Section 6.2.2 on page 249\)](#) and [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 6.3 Creating a Publisher with Non-Default QoS Policies (not from a profile)

```
DDS_PublisherQos publisher_qos;1
// get defaults
if (participant->get_default_publisher_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
// make QoS changes here
// for example, this changes the ENTITY_FACTORY QoS
publisher_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
// create the publisher
DDS_Publisher* publisher = participant->create_publisher(publisher_qos,
    NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}
```

Figure 6.4 Creating a Publisher with a QoS Profile

```
// create the publisher with QoS profile
DDSPublisher* publisher = participant->create_publisher_with_profile(
    "MyPublisherLibrary", "MyPublisherProfile",
    NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}
```

¹For the C API, you need to use `DDS_PublisherQos_INITIALIZER` or `DDS_PublisherQos_initialize()`. See [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#)

Figure 6.5 Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values

```

DDS_PublisherQos publisher_qos;1
// Get publisher QoS from profile
retcode = factory->get_publisher_qos_from_profile(publisher_qos,
    "PublisherLibrary", "PublisherProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
// New entity_factory autoenable_created_entities will be true
publisher_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_TRUE;
// create the publisher with modified QoS
DDSPublisher* publisher = participant->create_publisher(
    "Example Foo", type_name, publisher_qos,
    NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}

```

6.2.4.2 Comparing QoS Values

The `equals()` operation compares two *Publisher's* `DDS_PublisherQoS` structures for equality. It takes two parameters for the two *Publisher's* QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

6.2.4.3 Changing QoS Settings After the Publisher Has Been Created

There are 2 ways to change an existing *Publisher's* QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

- To change an existing *Publisher's* QoS programmatically (that is, without using a QoS profile): `get_qos()` and `set_qos()`. See the example code in [Figure 6.6 Changing the QoS of an Existing Publisher on the next page](#). It retrieves the current values by calling the *Publisher's* `get_qos()` operation. Then it modify the value and call `set_qos()` to apply the new value. Note, however, that some QoS Policies cannot be changed after the *Publisher* has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.
- You can also change a *Publisher's* (and all other *Entities'*) QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see [Figure 6.7 Changing the QoS of an Existing Publisher](#)

¹For the C API, you need to use `DDS_PublisherQos_INITIALIZER` or `DDS_PublisherQos_initialize()`. See [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#)

with a QoS Profile on the next page. For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 6.6 Changing the QoS of an Existing Publisher

```
DDS_PublisherQos publisher_qos;1
// Get current QoS. publisher points to an existing DDSPublisher.
if (publisher->get_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
// make changes
// New entity_factory autoenable_created_entities will be true
publisher_qos.entity_factory.autoenable_created_entities =DDS_BOOLEAN_TRUE;
// Set the new QoS
if (publisher->set_qos(publisher_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 6.7 Changing the QoS of an Existing Publisher with a QoS Profile

```
retcode = publisher->set_qos_with_profile(
    "PublisherProfileLibrary","PublisherProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

6.2.4.4 Getting and Setting the Publisher's Default QoS Profile and Library

You can retrieve the default QoS profile used to create *Publishers* with the `get_default_profile()` operation.

You can also get the default library for *Publishers*, as well as the library that contains the *Publisher's* default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

```
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()
```

There are also operations for setting the *Publisher's* default library and profile:

¹For the C API, you need to use `DDS_PublisherQos_INITIALIZER` or `DDS_PublisherQos_initialize()`. See [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#)

```
DDS_ReturnCode_t set_default_library (const char * library_name)
DDS_ReturnCode_t set_default_profile (const char * library_name,
                                     const char * profile_name)
```

These operations only affect which library/profile will be used as the default the next time a default *Publisher* library/profile is needed during a call to one of this *Publisher's* operations.

When calling a *Publisher* operation that requires a **profile_name** parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.) If the default library/profile is not set, the *Publisher* inherits the default from the *DomainParticipant*.

set_default_profile() does not set the default QoS for *DataWriters* created by the *Publisher*; for this functionality, use the *Publisher's* **set_default_datawriter_qos_with_profile()**, see [Getting and Setting Default QoS for DataWriters \(Section 6.2.4.5 below\)](#) (you may pass in NULL aftercalling the *Publisher's* **set_default_profile()**).

set_default_profile() does not set the default QoS for newly created *Publishers*; for this functionality, use the *DomainParticipant's* **set_default_publisher_qos_with_profile()** operation, see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on page 568\)](#).

6.2.4.5 Getting and Setting Default QoS for DataWriters

These operations set the default QoS that will be used for new *DataWriters* if **create_datawriter()** is called with DDS_DATAWRITER_QOS_DEFAULT as the **qos** parameter:

```
DDS_ReturnCode_t set_default_datawriter_qos (const DDS_DataWriterQos &qos)
DDS_ReturnCode_t set_default_datawriter_qos_with_profile (
    const char *library_name,
    const char *profile_name)
```

The above operations may potentially allocate memory, depending on the sequences contained in some QoS policies.

To get the default QoS that will be used for creating *DataWriters* if **create_datawriter()** is called with DDS_PARTICIPANT_QOS_DEFAULT as the **qos** parameter:

```
DDS_ReturnCode_t get_default_datawriter_qos (DDS_DataWriterQos & qos)
```

This operation gets the QoS settings that were specified on the last successful call to **set_default_datawriter_qos()** or **set_default_datawriter_qos_with_profile()**, or if the call was never made, the default values listed in DDS_DataWriterQos.

Note: It is not safe to set the default *DataWriter* QoS values while another thread may be simultaneously calling **get_default_datawriter_qos()**, **set_default_datawriter_qos()**, or **create_datawriter()** with DDS_DATAWRITER_QOS_DEFAULT as the **qos** parameter. It is also not safe to get the default *DataWriter* QoS values while another thread may be simultaneously calling **set_default_datawriter_qos()**.

6.2.4.6 Other Publisher QoS-Related Operations

- **Copying a Topic’s QoS into a DataWriter’s QoS**

This method is provided as a convenience for setting the values in a *DataWriterQos* structure before using that structure to create a *DataWriter*. As explained in [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#), most of the policies in a *TopicQos* structure do not apply directly to the *Topic* itself, but to the associated *DataWriters* and *DataReaders* of that *Topic*. The *TopicQos* serves as a single container where the values of QoS Policies that must be set compatibly across matching *DataWriters* and *DataReaders* can be stored.

Thus instead of setting the values of the individual QoS Policies that make up a *DataWriterQos* structure every time you need to create a *DataWriter* for a *Topic*, you can use the *Publisher’s* **copy_from_topic_qos()** operation to “import” the *Topic’s* QoS Policies into a *DataWriterQos* structure. This operation copies the relevant policies in the *TopicQos* to the corresponding policies in the *DataWriterQos*.

This copy operation will often be used in combination with the *Publisher’s* **get_default_datawriter_qos()** and the *Topic’s* **get_qos()** operations. The *Topic’s* QoS values are merged on top of the *Publisher’s* default *DataWriter* QoS Policies with the result used to create a new *DataWriter*, or to set the QoS of an existing one (see [Setting DataWriter QoS Policies \(Section 6.3.15 on page 300\)](#)).

- **Copying a Publisher’s QoS**

C API users should use the **DDS_PublisherQos_copy()** operation rather than using structure assignment when copying between two QoS structures. The **copy()** operation will perform a deep copy so that policies that allocate heap memory such as sequences are copied correctly. In C++, C++/CLI, C# and Java, a copy constructor is provided to take care of sequences automatically.

- **Clearing QoS-Related Memory**

Some QoS Policies contain sequences that allocate memory dynamically as they grow or shrink. The C API’s **DDS_PublisherQos_finalize()** operation frees the memory used by sequences but otherwise leaves the QoS unchanged. C API users should call **finalize()** on all **DDS_PublisherQos** objects before they are freed, or for QoS structures allocated on the stack, before they go out of scope. In C++, C++/CLI, C# and Java, the memory used by sequences is freed in the destructor.

6.2.5 Setting Up PublisherListeners

Like all *Entities*, *Publishers* may optionally have *Listeners*. *Listeners* are user-defined objects that implement a DDS-defined interface (i.e. a pre-defined set of callback functions). *Listeners* provide the means for Connex DDS to notify applications of any changes in *Statuses* (events) that may be relevant to it. By writing the callback functions in the *Listener* and installing the *Listener* into the *Publisher*, applications can be notified to handle the events of interest. For more general information on *Listeners* and *Statuses*, see [Listeners \(Section 4.4 on page 177\)](#).

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

As illustrated in [Publication Module \(Section Figure 6.1 on page 244\)](#), the *PublisherListener* interface extends the *DataWriterListener* interface. In other words, the *PublisherListener* interface contains all the functions in the *DataWriterListener* interface. There are no *Publisher*-specific *statuses*, and thus there are no *Publisher*-specific functions.

Instead, the methods of a *PublisherListener* will be called back for changes in the *Statuses* of any of the *DataWriters* that the *Publisher* has created. This is only true if the *DataWriter* itself does not have a *DataWriterListener* installed, see [Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#). If a *DataWriterListener* has been installed and has been enabled to handle a *Status* change for the *DataWriter*, then Connex DDS will call the method of the *DataWriterListener* instead.

If you want a *Publisher* to handle status events for its *DataWriters*, you can set up a *PublisherListener* during the *Publisher*'s creation or use the `set_listener()` method after the *Publisher* is created. The last parameter is a bit-mask with which you should set which *Status* events that the *PublisherListener* will handle. For example,

```
DDS_StatusMask mask = DDS_OFFERED_DEADLINE_MISSED_STATUS |
                      DDS_OFFERED_INCOMPATIBLE_QOS_STATUS;
publisher = participant->create_publisher(
    DDS_PUBLISHER_QOS_DEFAULT, listener, mask);
```

or

```
DDS_StatusMask mask = DDS_OFFERED_DEADLINE_MISSED_STATUS |
                      DDS_OFFERED_INCOMPATIBLE_QOS_STATUS;
publisher->set_listener(listener, mask);
```

As previously mentioned, the callbacks in the *PublisherListener* act as ‘default’ callbacks for all the *DataWriters* contained within. When Connex DDS wants to notify a *DataWriter* of a relevant *Status* change (for example, `PUBLICATION_MATCHED`), it first checks to see if the *DataWriter* has the corresponding *DataWriterListener* callback enabled (such as the `on_publication_matched()` operation). If so, Connex DDS dispatches the event to the *DataWriterListener* callback. Otherwise, Connex DDS dispatches the event to the corresponding *PublisherListener* callback.

A particular callback in a *DataWriter* is *not* enabled if either:

- The application installed a NULL *DataWriterListener* (meaning there are *no* callbacks for the *DataWriter* at all).
- The application has disabled the callback for a *DataWriterListener*. This is done by turning off the associated status bit in the *mask* parameter passed to the `set_listener()` or `create_datawriter()` call when installing the *DataWriterListener* on the *DataWriter*. For more information on *DataWriterListeners*, see [Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#).

Similarly, the callbacks in the *DomainParticipantListener* act as ‘default’ callbacks for all the *Publishers* that belong to it. For more information on *DomainParticipantListeners*, see [Setting Up DomainParticipantListeners \(Section 8.3.5 on page 560\)](#).

For example, [Example Code to Create a Publisher with a Simple Listener \(Section Figure 6.8 below\)](#) shows how to create a *Publisher* with a *Listener* that simply prints the events it receives.

Figure 6.8 Example Code to Create a Publisher with a Simple Listener

```
class MyPublisherListener : public DDSPublisherListener {
public:
    virtual void on_offered_deadline_missed(
        DDSDataWriter* writer,
        const DDS_OfferedDeadlineMissedStatus& status);
    virtual void on_liveliness_lost(
        DDSDataWriter* writer,
        const DDS_LivelinessLostStatus& status);
    virtual void on_offered_incompatible_qos(
        DDSDataWriter* writer,
        const DDS_OfferedIncompatibleQosStatus& status);
    virtual void on_publication_matched(
        DDSDataWriter* writer,
        const DDS_PublicationMatchedStatus& status);
    virtual void on_reliable_writer_cache_changed(
        DDSDataWriter* writer,
        const DDS_ReliableWriterCacheChangedStatus& status);
    virtual void on_reliable_reader_activity_changed (
        DDSDataWriter* writer,
        const DDS_ReliableReaderActivityChangedStatus& status);
};

void MyPublisherListener::on_offered_deadline_missed(
    DDSDataWriter* writer,
    const DDS_OfferedDeadlineMissedStatus& status)
{
    printf("on_offered_deadline_missed\n");
}

// ...Implement all remaining listeners in a similar manner...
DDSPublisherListener *myPubListener = new MyPublisherListener();
DDSPublisher* publisher =
    participant->create_publisher(DDS_PUBLISHER_QOS_DEFAULT,
    myPubListener, DDS_STATUS_MASK_ALL);
```

6.2.6 Finding a Publisher's Related DDS Entities

These *Publisher* operations are useful for obtaining a handle to related *Entities*:

- **get_participant()**: Gets the *DomainParticipant* with which a *Publisher* was created.
- **lookup_datawriter()**: Finds a *DataWriter* created by the *Publisher* with a *Topic* of a particular name. Note that in the event that multiple *DataWriters* were created by the same *Publisher* with the

same *Topic*, any one of them may be returned by this method. (In the Modern C++ API this method is a freestanding function, `dds::pub::find()`)

- **DDS_Publisher_as_Entity()**: This method is provided for C applications and is necessary when invoking the parent class *Entity* methods on *Publishers*. For example, to call the *Entity* method `get_status_changes()` on a *Publisher*, `my_pub`, do the following:

```
DDS_Entity_get_status_changes(DDS_Publisher_as_Entity(my_pub))
```

DDS_Publisher_as_Entity() is not provided in the C++, C++/CLI, C# and Java APIs because the object-oriented features of those languages make it unnecessary.

6.2.7 Waiting for Acknowledgments in a Publisher

The *Publisher's* `wait_for_acknowledgments()` operation blocks the calling thread until either all data written by the *Publisher's* reliable *DataWriters* is acknowledged or the duration specified by the `max_wait` parameter elapses, whichever happens first.

Note that if a thread is blocked in the call to `wait_for_acknowledgments()` on a *Publisher* and a different thread writes new DDS samples on any of the *Publisher's* reliable *DataWriters*, the new DDS samples must be acknowledged before unblocking the thread that is waiting on `wait_for_acknowledgments()`.

```
DDS_ReturnCode_t wait_for_acknowledgments (const DDS_Duration_t & max_wait)
```

This operation returns **DDS_RETCODE_OK** if all the DDS samples were acknowledged, or **DDS_RETCODE_TIMEOUT** if the `max_wait` duration expired first.

There is a similar operation available for individual *DataWriters*, see [Waiting for Acknowledgments in a DataWriter \(Section 6.3.11 on page 288\)](#).

The reliability protocol used by Connex DDS is discussed in [Reliable Communications \(Section Chapter 10 on page 629\)](#).

6.2.8 Statuses for Publishers

There are no statuses specific to the Publisher itself. The following statuses can be monitored by the *PublisherListener* for the *Publisher's* *DataWriters*.

- [OFFERED_DEADLINE_MISSED Status \(Section 6.3.6.5 on page 277\)](#)
- [LIVELINESS_LOST Status \(Section 6.3.6.4 on page 276\)](#)
- [OFFERED_INCOMPATIBLE_QOS Status \(Section 6.3.6.6 on page 277\)](#)
- [PUBLICATION_MATCHED Status \(Section 6.3.6.7 on page 278\)](#)
- [RELIABLE_WRITER_CACHE_CHANGED Status \(DDS Extension\) \(Section 6.3.6.8 on page 279\)](#)

- [RELIABLE_READER_ACTIVITY_CHANGED Status \(DDS Extension\) \(Section 6.3.6.9 on page 281\)](#)

6.2.9 Suspending and Resuming Publications

The operations `suspend_publications()` and `resume_publications()` provide a *hint* to Connex DDS that multiple data-objects within the Publisher are about to be written. Connex DDS does not currently use this hint.

6.3 DataWriters

To create a *DataWriter*, you need a *DomainParticipant* and a *Topic*.

You need a *DataWriter* for each *Topic* that you want to publish. Once you have a *DataWriter*, you can use it to perform the operations listed in [Table 6.3 DataWriter Operations](#). The most important operation is `write()`, described in [Writing Data \(Section 6.3.8 on page 283\)](#). For more details on all operations, see the API Reference HTML documentation.

DataWriters are created by using operations on a *DomainParticipant* or a *Publisher*, as described in [Creating DataWriters \(Section 6.3.1 on page 266\)](#). If you use the *DomainParticipant*'s operations, the *DataWriter* will belong to an implicit *Publisher* that is automatically created by the middleware. If you use a *Publisher*'s operations, the *DataWriter* will belong to that *Publisher*. So either way, the *DataWriter* belongs to a *Publisher*.

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

Table 6.3 DataWriter Operations

Working with ...	Operation	Description	Reference
DataWriters	assert_liveliness	Manually asserts the liveliness of the <i>DataWriter</i> .	Asserting Liveliness (Section 6.3.17 on page 311)
	enable	Enables the <i>DataWriter</i> .	Enabling DDS Entities (Section 4.1.2 on page 154)
	equals	Compares two <i>DataWriter</i> 's QoS structures for equality.	Comparing QoS Values (Section 6.3.15.2 on page 305)
	get_qos	Gets the QoS.	Setting DataWriter QoS Policies (Section 6.3.15 on page 300)
	lookup_instance	Gets a handle, given an instance. (Useful for keyed data types only.)	Looking Up an Instance Handle (Section 6.3.14.3 on page 299)
	set_qos	Modifies the QoS.	Setting DataWriter QoS Policies (Section 6.3.15 on page 300)
	set_qos_with_profile	Modifies the QoS based on a QoS profile.	Setting DataWriter QoS Policies (Section 6.3.15 on page 300)
	get_listener	Gets the currently installed Listener.	Setting Up DataWriterListeners (Section 6.3.4 on page 269)
	set_listener	Replaces the Listener.	

Table 6.3 DataWriter Operations

Working with ...	Operation	Description	Reference
FooDataWriter (See Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.7 on page 281))	dispose	States that the instance no longer exists. (Useful for keyed data types only.)	Disposing of Data (Section 6.3.14.2 on page 299)
	dispose_w_timestamp	Same as dispose, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)	
	flush	Makes the batch available to be sent on the network.	Flushing Batches of DDS Data Samples (Section 6.3.9 on page 287)
	get_key_value	Maps an instance_handle to the corresponding key.	Getting the Key Value for an Instance (Section 6.3.14.4 on page 299)
	narrow	A type-safe way to cast a pointer. This takes a DDSDataWriter pointer and 'narrows' it to a 'FooDataWriter' where 'Foo' is the related data type.	Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.7 on page 281)
	register_instance	States the intent of the <i>DataWriter</i> to write values of the data-instance that matches a specified key. Improves the performance of subsequent writes to the instance. (Useful for keyed data types only.)	Registering and Unregistering Instances (Section 6.3.14.1 on page 297)
	register_instance_w_timestamp	Like register_instance, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)	
	unregister_instance	Reverses register_instance. Relinquishes the ownership of the instance. (Useful for keyed data types only.)	
	unregister_instance_w_timestamp	Like unregister_instance, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)	
	write	Writes a new value for a data-instance.	Writing Data (Section 6.3.8 on page 283)
write_w_timestamp	Same as write, but allows the application to override the automatic source_timestamp.		

Table 6.3 DataWriter Operations

Working with ...	Operation	Description	Reference
FooDataWriter (See Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.7 on page 281))	write_w_params	Same as write, but allows the application to specify parameters such as source timestamp and instance handle.	Writing Data (Section 6.3.8 on page 283)
	dispose_w_params	Same as dispose, but allows the application to specify parameters such as source timestamp and instance handle..	Disposing of Data (Section 6.3.14.2 on page 299)
	register_w_params	Same as register, but allows the application to specify parameters such as source timestamp, instance handle.	Registering and Unregistering Instances (Section 6.3.14.1 on page 297)
	unregister_w_params	Same as unregister, but allows the application to specify parameters such as source timestamp, and instance handle.	
Matched Subscriptions	get_matched_subscriptions	Gets a list of subscriptions that have a matching <i>Topic</i> and compatible QoS. These are the subscriptions currently associated with the <i>DataWriter</i> .	Finding Matching Subscriptions (Section 6.3.16.1 on page 309)
	get_matched_subscription_data	Gets information on a subscription with a matching <i>Topic</i> and compatible QoS.	
	get_matched_subscription_locators	Gets a list of locators for subscriptions that have a matching <i>Topic</i> and compatible QoS. These are the subscriptions currently associated with the <i>DataWriter</i> .	
	get_matched_subscription_participant_data	Gets information about the <i>DomainParticipant</i> of a matching subscription.	Finding the Matching Subscription's ParticipantBuiltinTopicData (Section 6.3.16.2 on page 311)
Status	get_status_changes	Gets a list of statuses that have changed since the last time the application read the status or the listeners were called.	Getting Status and Status Changes (Section 4.1.4 on page 157)

Table 6.3 DataWriter Operations

Working with ...	Operation	Description	Reference
	get_liveliness_ lost_status	Gets LIVELINESS_LOST status.	
	get_offered_ deadline_ missed_status	Gets OFFERED_DEADLINE_MISSED status.	
	get_offered_ incompatible_qos_ status	Gets OFFERED_INCOMPATIBLE_QOS status.	
	get_publication_ match_ status	Gets PUBLICATION_MATCHED_QOS status.	
Status cont'd	get_reliable_ writer_ cache_changed_ status	Gets RELIABLE_WRITER_CACHE_CHANGED status	Statuses for DataWriters (Section 6.3.6 on page 271)
	get_reliable_ reader_ activity_changed_ status	Gets RELIABLE_READER_ACTIVITY_CHANGED status	
	get_datawriter_ cache_ status	Gets DATA_WRITER_CACHE_status	
	get_datawriter_ protocol_status	Gets DATA_WRITER_PROTOCOL status	
	get_matched_ subscription_ datawriter_ protocol_status	Gets DATA_WRITER_PROTOCOL status for this <i>DataWriter</i> , per matched subscription identified by the subscription_handle.	Statuses for DataWriters (Section 6.3.6 on page 271)
	get_matched_ subscription_ datawriter_ protocol_status_ by_locator	Gets DATA_WRITER_PROTOCOL status for this <i>DataWriter</i> , per matched subscription as identified by a locator.	

Table 6.3 DataWriter Operations

Working with ...	Operation	Description	Reference
Other	get_publisher	Gets the <i>Publisher</i> to which the <i>DataWriter</i> belongs.	Finding Related DDS Entities (Section 6.3.16.3 on page 311)
	get_topic	Get the Topic associated with the <i>DataWriter</i> .	
	wait_for_acknowledgements	Blocks the calling thread until either all data written by the <i>DataWriter</i> is acknowledged by all matched Reliable <i>DataReaders</i> , or until the a specified timeout duration, <code>max_wait</code> , elapses.	Waiting for Acknowledgments in a DataWriter (Section 6.3.11 on page 288)

6.3.1 Creating DataWriters

Before you can create a *DataWriter*, you need a *DomainParticipant*, a *Topic*, and optionally, a *Publisher*.

DataWriters are created by calling `create_datawriter()` or `create_datawriter_with_profile()`—these operations exist for *DomainParticipants* and *Publishers*. If you use the *DomainParticipant* to create a *DataWriter*, it will belong to the implicit *Publisher* described in [Creating Publishers Explicitly vs. Implicitly \(Section 6.2.1 on page 248\)](#). If you use a *Publisher*'s operations to create a *DataWriter*, it will belong to that *Publisher*.

A *QoS profile* is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Note: In the Modern C++ API *DataWriters* provide constructors whose first argument is a *Publisher*. The only required arguments are the publisher and the topic.

```

DDSDataWriter* create_datawriter (
    DDS::Topic *topic,
    const DDS::DataWriterQos &qos,
    DDSDataWriterListener *listener,
    DDS_StatusMask mask)
DDSDataWriter * create_datawriter_with_profile (
    DDS::Topic * topic,
    const char * library_name,
    const char * profile_name,
    DDSDataWriterListener * listener,
    DDS_StatusMask mask)

```

Where:

topic The *Topic* that the *DataWriter* will publish. This must have been previously created by the same *DomainParticipant*.

- qos** If you want the default QoS settings (described in the API Reference HTML documentation), use the constant `DDS_DATAWRITER_QOS_DEFAULT` for this parameter (see [Figure 6.9 Creating a DataWriter with Default QoS Policies and a Listener on the facing page](#)). If you want to customize any of the QoS policies, supply a QoS structure (see [Setting DataWriter QoS Policies \(Section 6.3.15 on page 300\)](#)).
- Note:** If you use `DDS_DATAWRITER_QOS_DEFAULT` for the **qos** parameter, it is not safe to create the *DataWriter* while another thread may be simultaneously calling the *Publisher's* `set_default_datawriter_qos()` operation.
- listener** *Listeners* are callback routines. Connex DDS uses them to notify your application of specific events (status changes) that may occur with respect to the *DataWriter*. The *listener* parameter may be set to `NULL`; in this case, the *PublisherListener* (or if that is `NULL`, the *DomainParticipantListener*) will be used instead. For more information, see [Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#).
- mask** This bit-mask indicates which status changes will cause the *Listener* to be invoked. The bits set in the mask must have corresponding callbacks implemented in the *Listener*. If you use `NULL` for the *Listener*, use `DDS_STATUS_MASK_NONE` for this parameter. If the *Listener* implements all callbacks, use `DDS_STATUS_MASK_ALL`. For information on statuses, see [Listeners \(Section 4.4 on page 177\)](#).
- library_name** A QoS Library is a named set of QoS profiles. See [URL Groups \(Section 17.8 on page 814\)](#).
- profile_name** A QoS profile groups a set of related QoS, usually one per entity. See [URL Groups \(Section 17.8 on page 814\)](#)

For more examples on how to create a *DataWriter*, see [Configuring QoS Settings when the DataWriter is Created \(Section 6.3.15.1 on page 303\)](#)

After you create a *DataWriter*, you can use it to write data. See [Writing Data \(Section 6.3.8 on page 283\)](#).

Note: When a *DataWriter* is created, only those transports already registered are available to the *DataWriter*. The built-in transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter* is created, or (c) you look up a built-in data reader, whichever happens first.

Figure 6.9 Creating a DataWriter with Default QosPolicies and a Listener

```

// MyWriterListener is user defined, extends DDSDataWriterListener
DDSDataWriterListener* writer_listener = new MyWriterListener();
DDSDataWriter* writer = publisher->create_datawriter(
    topic,
    DDS_DATAWRITER_QOS_DEFAULT,
    writer_listener,
    DDS_STATUS_MASK_ALL);
if (writer == NULL) {
    // ... error
};
// narrow it for your specific data type
FooDataWriter* foo_writer = FooDataWriter::narrow(writer);

```

6.3.2 Getting All DataWriters

To retrieve all the *DataWriters* created by the *Publisher*, use the *Publisher's* `get_all_datawriters()` operation:

```

DDS_ReturnCode_t get_all_datawriters(DDS_Publisher* self,
                                     struct DDS_DataWriterSeq* writers);

```

In the Modern C++ API, use the freestanding function `rti::pub::find_datawriters()`.

6.3.3 Deleting DataWriters

(Note: in the Modern C++ API, *Entities* are automatically destroyed, see [Creating and Deleting DDS Entities \(Section 4.1.1 on page 153\)](#))

To delete a single *DataWriter*, use the *Publisher's* `delete_datawriter()` operation:

```

DDS_ReturnCode_t delete_datawriter (
    DDSDataWriter *a_datawriter)

```

Note: A *DataWriter* cannot be deleted within its own writer listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#)

To delete all of a *Publisher's* *DataWriters*, use the *Publisher's* `delete_contained_entities()` operation (see [Deleting Contained DataWriters \(Section 6.2.3.1 on page 251\)](#)).

6.3.3.1 Special Instructions for deleting DataWriters if you are using the 'Timestamp' APIs and BY_SOURCE_TIMESTAMP Destination Order:

This section only applies when the *DataWriter's* *DestinationOrderQosPolicy's* **kind** is `BY_SOURCE_TIMESTAMP`.

Calls to `delete_datawriter()` may fail if your application has previously used the “with timestamp” APIs (`write_w_timestamp()`, `register_instance_w_timestamp()`, `unregister_instance_w_timestamp()`, or `dispose_w_timestamp()`) with a timestamp that is larger than the time at which `delete_datawriter()` is called.

To prevent `delete_datawriter()` from failing in this situation, either:

- Change the `WriterDataLifeCycle` QoS Policy so that Connex DDS will not auto-dispose unregistered instances:

```
writer_qos.writer_data_lifecycle.  
    autodispose_unregistered_instances =  
        DDS_BOOLEAN_FALSE;
```

or

- Explicitly call `unregister_instance_w_timestamp()` for all instances modified with the `*_w_timestamp()` APIs before calling `delete_datawriter()`.

6.3.4 Setting Up DataWriterListeners

DataWriters may optionally have *Listeners*. *Listeners* are essentially callback routines and provide the means for Connex DDS to notify your application of the occurrence of events (status changes) relevant to the *DataWriter*. For more general information on *Listeners*, see [Listeners \(Section 4.4 on page 177\)](#).

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

If you do not implement a *DataWriterListener*, the associated *PublisherListener* is used instead. If that *Publisher* also does not have a *Listener*, then the *DomainParticipant's Listener* is used if one exists (see [Setting Up PublisherListeners \(Section 6.2.5 on page 257\)](#) and [Setting Up DomainParticipantListeners \(Section 8.3.5 on page 560\)](#)).

Listeners are typically set up when the *DataWriter* is created (see [Publishers \(Section 6.2 on page 243\)](#)). You can also set one up after creation by using the `set_listener()` operation. Connex DDS will invoke a *DataWriter's Listener* to report the status changes listed in [Table 6.4 DataWriterListener Callbacks](#) (if the *Listener* is set up to handle the particular status, see [Setting Up DataWriterListeners \(Section 6.3.4 above\)](#)).

Table 6.4 DataWriterListener Callbacks

This DataWriterListener callback...	... is triggered by ...
<code>on_instance_replaced()</code>	A replacement of an existing instance by a new instance; see Configuring DataWriter Instance Replacement (Section 6.5.20.2 on page 407)

Table 6.4 DataWriterListener Callbacks

This DataWriterListener callback...	... is triggered by ...
on_liveliness_lost	A change to LIVELINESS_LOST Status (Section 6.3.6.4 on page 276)
on_offered_deadline_missed	A change to OFFERED_DEADLINE_MISSED Status (Section 6.3.6.5 on page 277)
on_offered_incompatible_qos	A change to OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.6.6 on page 277)
on_publication_matched	A change to PUBLICATION_MATCHED Status (Section 6.3.6.7 on page 278)
on_reliable_writer_cache_changed	A change to RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.6.8 on page 279)
on_reliable_reader_activity_changed	A change to RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.6.9 on page 281)

6.3.5 Checking DataWriter Status

You can access an individual communication status for a *DataWriter* with the operations shown in [Table 6.5 DataWriter Status Operations](#).

Table 6.5 DataWriter Status Operations

Use this operation...	...to retrieve this status:
get_datawriter_cache_status	DATA_WRITER_CACHE_STATUS (Section 6.3.6.2 on page 272)
get_datawriter_protocol_status	DATA_WRITER_PROTOCOL_STATUS (Section 6.3.6.3 on page 273)
get_matched_subscription_datawriter_protocol_status	
get_matched_subscription_datawriter_protocol_status_by_locator	
get_liveliness_lost_status	LIVELINESS_LOST Status (Section 6.3.6.4 on page 276)
get_offered_deadline_missed_status	OFFERED_DEADLINE_MISSED Status (Section 6.3.6.5 on page 277)
get_offered_incompatible_qos_status	OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.6.6 on page 277)
get_publication_match_status	PUBLICATION_MATCHED Status (Section 6.3.6.7 on page 278)
get_reliable_writer_cache_changed_status	RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.6.8 on page 279)

Table 6.5 DataWriter Status Operations

Use this operation...	...to retrieve this status:
<code>get_reliable_reader_activity_changed_status</code>	RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.6.9 on page 281)
<code>get_status_changes</code>	A list of what changed in all of the above.

These methods are useful in the event that no *Listener* callback is set to receive notifications of status changes. If a *Listener* is used, the callback will contain the new status information, in which case calling these methods is unlikely to be necessary.

The `get_status_changes()` operation provides a list of statuses that have changed since the last time the status changes were ‘reset.’ A status change is reset each time the application calls the corresponding `get_*_status()`, as well as each time Connex DDS returns from calling the *Listener* callback associated with that status.

For more on status, see [Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#), [Statuses for DataWriters \(Section 6.3.6 below\)](#), and [Listeners \(Section 4.4 on page 177\)](#).

6.3.6 Statuses for DataWriters

There are several types of statuses available for a *DataWriter*. You can use the `get_*_status()` operations ([Setting DataWriter QoS Policies \(Section 6.3.15 on page 300\)](#)) to access them, or use a *DataWriter-Listener* ([Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#)) to listen for changes in their values. Each status has an associated data structure and is described in more detail in the following sections.

- [APPLICATION_ACKNOWLEDGMENT_STATUS \(Section 6.3.6.1 on the facing page\)](#)
- [DATA_WRITER_CACHE_STATUS \(Section 6.3.6.2 on the facing page\)](#)
- [DATA_WRITER_PROTOCOL_STATUS \(Section 6.3.6.3 on page 273\)](#)
- [LIVELINESS_LOST Status \(Section 6.3.6.4 on page 276\)](#)
- [OFFERED_DEADLINE_MISSED Status \(Section 6.3.6.5 on page 277\)](#)
- [OFFERED_INCOMPATIBLE_QOS Status \(Section 6.3.6.6 on page 277\)](#)
- [PUBLICATION_MATCHED Status \(Section 6.3.6.7 on page 278\)](#)
- [RELIABLE_WRITER_CACHE_CHANGED Status \(DDS Extension\) \(Section 6.3.6.8 on page 279\)](#)
- [RELIABLE_READER_ACTIVITY_CHANGED Status \(DDS Extension\) \(Section 6.3.6.9 on page 281\)](#)

6.3.6.1 APPLICATION_ACKNOWLEDGMENT_STATUS

This status indicates that a *DataWriter* has received an application-level acknowledgment for a DDS sample, and triggers a *DataWriter* callback:

```
void DDSDataWriterListener::on_application_acknowledgment(
    DDSDataWriter * writer,
    const DDS_AcknowledgmentInfo & info)
```

on_application_acknowledgment() is called when a DDS sample is application-level acknowledged. It provides identities of the DDS sample and the acknowledging *DataReader*, as well as user-specified response data sent from the *DataReader* by the acknowledgment message—see [Table 6.6 DDS_AcknowledgmentInfo](#).

Table 6.6 DDS_AcknowledgmentInfo

Type	Field Name	Description
DDS_InstanceHandle_t	subscription_handle	Subscription handle of the acknowledging <i>DataReader</i> .
struct DDS_SampleIdentity_t	sample_identity	Identity of the DDS sample being acknowledged.
DDS_Boolean	valid_response_data	Flag indicating validity of the user response data in the acknowledgment.
struct DDS_AckResponseData_t	response_data	User data payload of application-level acknowledgment message.

This status is only applicable when the *DataWriter*'s Reliability QosPolicy's **acknowledgment_kind** is `DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE` or `DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE`.

6.3.6.2 DATA_WRITER_CACHE_STATUS

This status keeps track of the number of DDS samples in the *DataWriter*'s queue.

This status does not have an associated Listener. You can access this status by calling the *DataWriter*'s **get_datawriter_cache_status()** operation, which will return the status structure described in [Table 6.7 DDS_DataWriterCacheStatus](#).

Table 6.7 DDS_DataWriterCacheStatus

Type	Field Name	Description
DDS_Long	sample_count_peak	Highest number of DDS samples in the <i>DataWriter</i> 's queue over the lifetime of the <i>DataWriter</i> .
DDS_Long	sample_count	Current number of DDS samples in the <i>DataWriter</i> 's queue (including DDS unregister and dispose samples)

6.3.6.3 DATA_WRITER_PROTOCOL_STATUS

This status includes internal protocol related metrics (such as the number of DDS samples pushed, pulled, filtered) and the status of wire-protocol traffic.

- **Pulled DDS samples** are DDS samples sent for repairs (that is, DDS samples that had to be resent), for late joiners, and all DDS samples sent by the local *DataWriter* when **push_on_write** (in [DATA_WRITER_PROTOCOL QoSPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347)) is `DDS_BOOLEAN_FALSE`.
- **Pushed DDS samples** are DDS samples sent on `write()` when **push_on_write** is `DDS_BOOLEAN_TRUE`.
- **Filtered DDS samples** are DDS samples that are not sent due to *DataWriter* filtering (time-based filtering and `ContentFilteredTopics`).

This status does not have an associated Listener. You can access this status by calling the following operations on the *DataWriter* (all of which return the status structure described in [Table 6.8 DDS_DataWriter-ProtocolStatus](#)):

- **get_datawriter_protocol_status()** returns the sum of the protocol status for all the matched subscriptions for the *DataWriter*.
- **get_matched_subscription_datawriter_protocol_status()** returns the protocol status of a particular matched subscription, identified by a `subscription_handle`.
- **get_matched_subscription_datawriter_protocol_status_by_locator()** returns the protocol status of a particular matched subscription, identified by a locator. (See [Locator Format](#) (Section 14.2.1.1 on page 714).)

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted. If you try to get the matched subscription status for a remote entity that is no longer alive, the ‘get status’ call will return an error.

Table 6.8 DDS_DataWriterProtocolStatus

Type	Field Name	Description
DDS_LongLong	pushed_sample_count	The number of user DDS samples pushed on write from a local <i>DataWriter</i> to a matching remote <i>DataReader</i> .
	pushed_sample_count_change	The incremental change in the number of user DDS samples pushed on write from a local <i>DataWriter</i> to a matching remote <i>DataReader</i> since the last time the status was read.
	pushed_sample_bytes	The number of bytes of user DDS samples pushed on write from a local <i>DataWriter</i> to a matching remote <i>DataReader</i> .
	pushed_sample_bytes_change	The incremental change in the number of bytes of user DDS samples pushed on write from a local <i>DataWriter</i> to a matching remote <i>DataReader</i> since the last time the status was read.
DDS_LongLong	sent_heartbeat_count	The number of Heartbeats sent between a local <i>DataWriter</i> and matching remote <i>DataReaders</i> .
	sent_heartbeat_count_change	The incremental change in the number of Heartbeats sent between a local <i>DataWriter</i> and matching remote <i>DataReaders</i> since the last time the status was read.
	sent_heartbeat_bytes	The number of bytes of Heartbeats sent between a local <i>DataWriter</i> and matching remote <i>DataReader</i> .
	sent_heartbeat_bytes_change	The incremental change in the number of bytes of Heartbeats sent between a local <i>DataWriter</i> and matching remote <i>DataReaders</i> since the last time the status was read.
DDS_LongLong	pulled_sample_count	The number of user DDS samples pulled from local <i>DataWriter</i> by matching <i>DataReaders</i> .
	pulled_sample_count_change	The incremental change in the number of user DDS samples pulled from local <i>DataWriter</i> by matching <i>DataReaders</i> since the last time the status was read.
	pulled_sample_bytes	The number of bytes of user DDS samples pulled from local <i>DataWriter</i> by matching <i>DataReaders</i> .
	pulled_sample_bytes_change	The incremental change in the number of bytes of user DDS samples pulled from local <i>DataWriter</i> by matching <i>DataReaders</i> since the last time the status was read.
DDS_LongLong	received_ack_count	The number of ACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> .
	received_ack_count_change	The incremental change in the number of ACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> since the last time the status was read.
	received_ack_bytes	The number of bytes of ACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> .
	received_ack_bytes_change	The incremental change in the number of bytes of ACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> since the last time the status was read.

Table 6.8 DDS_DataWriterProtocolStatus

Type	Field Name	Description
DDS_LongLong	received_nack_count	The number of NACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> .
	received_nack_count_change	The incremental change in the number of NACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> since the last time the status was read.
	received_nack_bytes	The number of bytes of NACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> .
	received_nack_bytes_change	The incremental change in the number of bytes of NACKs from a remote <i>DataReader</i> received by a local <i>DataWriter</i> since the last time the status was read.
DDS_LongLong	sent_gap_count	The number of GAPS sent from local <i>DataWriter</i> to matching remote <i>DataReaders</i> .
	sent_gap_count_change	The incremental change in the number of GAPS sent from local <i>DataWriter</i> to matching remote <i>DataReaders</i> since the last time the status was read.
	sent_gap_bytes	The number of bytes of GAPS sent from local <i>DataWriter</i> to matching remote <i>DataReaders</i> .
	sent_gap_bytes_change	The incremental change in the number of bytes of GAPS sent from local <i>DataWriter</i> to matching remote <i>DataReaders</i> since the last time the status was read.
DDS_LongLong	rejected_sample_count	The number of times a DDS sample is rejected for unanticipated reasons in the send path.
	rejected_sample_count_change	The incremental change in the number of times a DDS sample is rejected due to exceptions in the send path since the last time the status was read.
DDS_Long	send_window_size	Current maximum number of outstanding DDS samples allowed in the <i>DataWriter's</i> queue.

Table 6.8 DDS_DataWriterProtocolStatus

Type	Field Name	Description
DDS_SequenceNumber_t	first_available_sample_sequence_number	Sequence number of the first available DDS sample in the <i>DataWriter's</i> reliability queue.
	last_available_sample_sequence_number	Sequence number of the last available DDS sample in the <i>DataWriter's</i> reliability queue.
	first_unacknowledged_sample_sequence_number	Sequence number of the first unacknowledged DDS sample in the <i>DataWriter's</i> reliability queue.
	first_available_sample_virtual_sequence_number	Virtual sequence number of the first available DDS sample in the <i>DataWriter's</i> reliability queue.
	last_available_sample_virtual_sequence_number	Virtual sequence number of the last available DDS sample in the <i>DataWriter's</i> reliability queue.
	first_unacknowledged_sample_virtual_sequence_number	Virtual sequence number of the first unacknowledged DDS sample in the <i>DataWriter's</i> reliability queue.
DDS_SequenceNumber_t	first_unacknowledged_sample_subscription_handle	Instance Handle of the matching remote <i>DataReader</i> for which the <i>DataWriter</i> has kept the first available DDS sample in the reliability queue.
	first_unelapsed_keep_duration_sample_sequence_number	Sequence number of the first DDS sample kept in the <i>DataWriter's</i> queue whose <i>keep_duration</i> (applied when disable_positive_acks is set) has not yet elapsed.

6.3.6.4 LIVELINESS_LOST Status

A change to this status indicates that the *DataWriter* failed to signal its liveliness within the time specified by the [LIVELINESS QosPolicy](#) (Section 6.5.13 on page 382).

It is different than the [RELIABLE_READER_ACTIVITY_CHANGED Status \(DDS Extension\)](#) (Section 6.3.6.9 on page 281) status that provides information about the liveliness of a *DataWriter's* matched *DataReaders*; this status reflects the *DataWriter's own* liveliness.

The structure for this status appears in [Table 6.9 DDS_LivelinessLostStatus](#).

Table 6.9 DDS_LivelinessLostStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the <i>DataWriter</i> failed to explicitly signal its liveliness within the liveliness period.
DDS_Long	total_count_change	The change in total_count since the last time the Listener was called or the status was read.

The *DataWriterListener*'s **on_liveliness_lost()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataWriter*'s **get_liveliness_lost_status()** operation.

6.3.6.5 OFFERED_DEADLINE_MISSED Status

A change to this status indicates that the *DataWriter* failed to write data within the time period set in its [DEADLINE QoS Policy \(Section 6.5.5 on page 363\)](#).

The structure for this status appears in [Table 6.10 DDS_OfferedDeadlineMissedStatus](#).

Table 6.10 DDS_OfferedDeadlineMissedStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the <i>DataWriter</i> failed to write within its offered deadline.
DDS_Long	total_count_change	The change in total_count since the last time the <i>Listener</i> was called or the status was read.
DDS_InstanceHandle_t	last_instance_handle	Handle to the last data-instance in the <i>DataWriter</i> for which an offered deadline was missed.

The *DataWriterListener*'s **on_offered_deadline_missed()** operation is invoked when this status changes. You can also retrieve the value by calling the *DataWriter*'s **get_deadline_missed_status()** operation.

6.3.6.6 OFFERED_INCOMPATIBLE_QOS Status

A change to this status indicates that the *DataWriter* discovered a *DataReader* for the same *Topic*, but that *DataReader* had requested QoS settings incompatible with this *DataWriter*'s offered QoS.

The structure for this status appears in [Table 6.11 DDS_OfferedIncompatibleQoSStatus](#).

Table 6.11 DDS_OfferedIncompatibleQoSStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the <i>DataWriter</i> discovered a <i>DataReader</i> for the same <i>Topic</i> with a requested QoS that is incompatible with that offered by the <i>DataWriter</i> .
DDS_Long	total_count_change	The change in total_count since the last time the <i>Listener</i> was called or the status was read.
DDS_QosPolicyId_t	last_policy_id	The ID of the QoSPolicy that was found to be incompatible the last time an incompatibility was detected. (Note: if there are multiple incompatible policies, only one of them is reported here.)
DDS_QosPolicyCountSeq	policies	A list containing—for each policy—the total number of times that the <i>DataWriter</i> discovered a <i>DataReader</i> for the same <i>Topic</i> with a requested QoS that is incompatible with that offered by the <i>DataWriter</i> .

The *DataWriterListener*'s **on_offered_incompatible_qos()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataWriter*'s **get_offered_incompatible_qos_status()** operation.

6.3.6.7 PUBLICATION_MATCHED Status

A change to this status indicates that the *DataWriter* discovered a matching *DataReader*.

A 'match' occurs only if the *DataReader* and *DataWriter* have the same *Topic*, same data type (implied by having the same *Topic*), and compatible QoS Policies. In addition, if user code has directed Connex DDS to ignore certain *DataReaders*, then those *DataReaders* will never be matched. See [Ignoring Publications and Subscriptions \(Section 16.4.2 on page 786\)](#) for more on setting up a *DomainParticipant* to ignore specific *DataReaders*.

The structure for this status appears in [Table 6.12 DDS_PublicationMatchedStatus](#).

Table 6.12 DDS_PublicationMatchedStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the <i>DataWriter</i> discovered a "match" with a <i>DataReader</i> .
	total_count_change	The change in total_count since the last time the Listener was called or the status was read.
	current_count	The number of <i>DataReaders</i> currently matched to the <i>DataWriter</i> .
	current_count_peak	The highest value that current_count has reached until now.
	current_count_change	The change in current_count since the last time the listener was called or the status was read.
DDS_InstanceHandle_t	last_subscription_handle	Handle to the last <i>DataReader</i> that matched the <i>DataWriter</i> causing the status to change.

The *DataWriterListener*'s **on_publication_matched()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataWriter*'s **get_publication_match_status()** operation.

6.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension)

A change to this status indicates that the number of unacknowledged DDS samples¹ in a reliable *DataWriter*'s cache has reached one of these trigger points:

- The cache is empty (contains no unacknowledged DDS samples)
- The cache is full (the number of unacknowledged DDS samples has reached the value specified in **DDS_ResourceLimitsQosPolicy::max_samples**)
- The number of unacknowledged DDS samples has reached a high or low watermark. See the **high_watermark** and **low_watermark** fields in [Table 6.37 DDS_RtpsReliableWriterProtocol_t](#) of the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347).

For more about the reliable protocol used by Connex DDS and specifically, what it means for a DDS sample to be 'unacknowledged,' see [Reliable Communications \(Section Chapter 10 on page 629\)](#).

The structure for this status appears in [Table 6.13 DDS_ReliableWriterCacheChangedStatus](#). The supporting structure, **DDS_ReliableWriterCacheEventCount**, is described in [Table 6.14 DDS_ReliableWriterCacheEventCount](#).

¹If batching is enabled, this still refers to a number of *DDS samples*, not *batches*.

Table 6.13 DDS_ReliableWriterCacheChangedStatus

Type	Field Name	Description
DDS_ReliableWriterCacheEventCount	empty_reliable_writer_cache	How many times the reliable <i>DataWriter's</i> cache of unacknowledged DDS samples has become empty.
	full_reliable_writer_cache	How many times the reliable <i>DataWriter's</i> cache of unacknowledged DDS samples has become full.
	low_watermark_reliable_writer_cache	How many times the reliable <i>DataWriter's</i> cache of unacknowledged DDS samples has fallen to the low watermark.
	high_watermark_reliable_writer_cache	How many times the reliable <i>DataWriter's</i> cache of unacknowledged DDS samples has risen to the high watermark.
DDS_Long	unacknowledged_sample_count	The current number of unacknowledged DDS samples in the <i>DataWriter's</i> cache.
	unacknowledged_sample_count_peak	The highest value that unacknowledged_sample_count has reached until now.

Table 6.14 DDS_ReliableWriterCacheEventCount

Type	Field Name	Description
DDS_Long	total_count	The total number of times the event has occurred.
DDS_Long	total_count_change	The number of times the event has occurred since the <i>Listener</i> was last invoked or the status read.

The *DataWriterListener's* **on_reliable_writer_cache_changed()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataWriter's* **get_reliable_writer_cache_changed_status()** operation.

If a reliable *DataWriter's* send window is finite, with both **RtpsReliableWriterProtocol_t.min_send_window_size** and **RtpsReliableWriterProtocol_t.max_send_window_size** set to positive values, then **full_reliable_writer_cache_status** counts the number of times the unacknowledged DDS sample count reaches the send window size.

6.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension)

This status indicates that one or more reliable *DataReaders* has become active or inactive.

This status is the reciprocal status to the [LIVELINESS_CHANGED Status \(Section 7.3.7.4 on page 475\)](#) on the *DataReader*. It is different than [LIVELINESS_LOST Status \(Section 6.3.6.4 on page 276\)](#) status on the *DataWriter*, in that the latter informs the *DataWriter* about its *own* liveliness; this status informs the *DataWriter* about the liveliness of its matched *DataReaders*.

A reliable *DataReader* is considered active by a reliable *DataWriter* with which it is matched if that *DataReader* acknowledges the DDS samples that it has been sent in a timely fashion. For the definition of "timely" in this context, see [DATA_WRITER_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#).

This status is only used for *DataWriters* whose [RELIABILITY QoSPolicy \(Section 6.5.19 on page 400\)](#) is set to RELIABLE. For best-effort *DataWriters*, all counts in this status will remain at zero.

The structure for this status appears in [Table 6.15 DDS_ReliableReaderActivityChangedStatus](#).

Table 6.15 DDS_ReliableReaderActivityChangedStatus

Type	Field Name	Description
DDS_Long	active_count	The current number of reliable readers currently matched with this reliable <i>DataWriter</i> .
	inactive_count	The number of reliable readers that have been dropped by this reliable <i>DataWriter</i> because they failed to send acknowledgments in a timely fashion.
	active_count_change	The change in the number of active reliable <i>DataReaders</i> since the <i>Listener</i> was last invoked or the status read.
	inactive_count_change	The change in the number of inactive reliable <i>DataReaders</i> since the <i>Listener</i> was last invoked or the status read.
DDS_InstanceHandle_t	last_instance_handle	The instance handle of the last reliable <i>DataReader</i> to be determined to be inactive.

The *DataWriterListener*'s `on_reliable_reader_activity_changed()` callback is invoked when this status changes. You can also retrieve the value by calling the *DataWriter*'s `get_reliable_reader_activity_changed_status()` operation.

6.3.7 Using a Type-Specific DataWriter (FooDataWriter)

(Note: this section does not apply to the Modern C++ API where a *DataWriter*'s data type is part of its template definition: `DataWriter<Foo>`)

Recall that a *Topic* is bound to a data type that specifies the format of the data associated with the *Topic*. Data types are either defined dynamically or in code generated from definitions in IDL or XML; see [Data Types and DDS Data Samples \(Section Chapter 3 on page 23\)](#). For each of your application's generated data types, such as 'Foo', there will be a `FooDataWriter` class (or a set of functions in C). This class allows the application to use a type-safe interface to interact with DDS samples of type 'Foo'. You will use the `FooDataWriter`'s `write()` operation used to send data. For dynamically defined data-types, you will use the `DynamicDataWriter` class.

In fact, you will use the *FooDataWriter* any time you need to perform type-specific operations, such as registering or writing instances. [Table 6.3 DataWriter Operations](#) indicates which operations must be called using *FooDataWriter*. For operations that are not type-specific, you can call the operation using either a *FooDataWriter* or a *DDSDataWriter* object¹.

You may notice that the *Publisher*'s `create_datawriter()` operation returns a pointer to an object of type `DDSDataWriter`; this is because the `create_datawriter()` method is used to create *DataWriters* of any data type. However, when executed, the function actually returns a specialization (an object of a derived class) of the *DataWriter* that is specific for the data type of the associated *Topic*. For a *Topic* of type 'Foo', the object actually returned by `create_datawriter()` is a `FooDataWriter`.

To safely cast a generic `DDSDataWriter` pointer to a `FooDataWriter` pointer, you should use the static `narrow()` method of the `FooDataWriter` class. The `narrow()` method will return `NULL` if the generic `DDSDataWriter` pointer is not pointing at an object that is really a `FooDataWriter`.

For instance, if you create a *Topic* bound to the type 'Alarm', all *DataWriters* created for that *Topic* will be of type 'AlarmDataWriter.' To access the type-specific methods of `AlarmDataWriter`, you must cast the generic `DDSDataWriter` pointer returned by `create_datawriter()`. For example:

```
DDSDataWriter* writer = publisher->create_datawriter(topic,writer_qos, NULL, NULL);
AlarmDataWriter *alarm_writer = AlarmDataWriter::narrow(writer);

if (alarm_writer == NULL) {

    // ... error

};
```

In the C API, there is also a way to do the opposite of `narrow()`. `FooDataWriter_as_datawriter()` casts a `FooDataWriter` as a `DDSDataWriter`, and `FooDataReader_as_datareader()` casts a `FooDataReader` as a `DDSDataReader`.

¹In the C API, the non type-specific operations must be called using a `DDS_DataWriter` pointer.

6.3.8 Writing Data

The **write()** operation informs Connex DDS that there is a new value for a data-instance to be published for the corresponding *Topic*. By default, calling **write()** will send the data immediately over the network (assuming that there are matched *DataReaders*). However, you can configure and execute operations on the *DataWriter*'s *Publisher* to buffer the data so that it is sent in a batch with data from other *DataWriters* or even to prevent the data from being sent. Those sending “modes” are configured using the [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#) as well as the *Publisher*'s **suspend/resume_publications()** operations. The actual transport-level communications may be done by a separate, lower-priority thread when the *Publisher* is configured to send the data for its *DataWriters*. For more information on threads, see [Connex DDS Threading Model \(Section Chapter 19 on page 837\)](#).

When you call **write()**, Connex DDS automatically attaches a stamp of the current time that is sent with the DDS data sample to the *DataReader*(s). The timestamp appears in the **source_timestamp** field of the **DDS_SampleInfo** structure that is provided along with your data using *DataReaders* (see [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#)).

```
DDS_ReturnCode_t write (const Foo &instance_data,
                      const DDS_InstanceHandle_t &handle)
```

You can use an alternate *DataWriter* operation called **write_w_timestamp()**. This performs the same action as **write()**, but allows the application to explicitly set the **source_timestamp**. This is useful when you want the user application to set the value of the timestamp instead of the default clock used by Connex DDS.

```
DDS_ReturnCode_t write_w_timestamp (
    const Foo &instance_data,
    const DDS_InstanceHandle_t &handle,
    const DDS_Time_t &source_timestamp)
```

Note that, in general, the application should not mix these two ways of specifying timestamps. That is, for each *DataWriter*, the application should either always use the automatic timestamping mechanism (by calling the normal operations) or always specify a timestamp (by calling the “**w_timestamp**” variants of the operations). Mixing the two methods may result in not receiving sent data.

You can also use an alternate *DataWriter* operation, **write_w_params()**, which performs the same action as **write()**, but allows the application to explicitly set the fields contained in the **DDS_WriteParams** structure, see [Table 6.16 DDS_WriteParams_t](#).

Table 6.16 DDS_WriteParams_t

Type	Field Name	Description
DDS_Boolean	replace_auto	<p>Allows retrieving the actual value of those fields that were automatic.</p> <p>When this field is set to true, the fields that were configured with an automatic value (for example, DDS_AUTO_SAMPLE_IDENTITY in identity) receive their actual value after write_w_params is called.</p>
DDS_SampleIdentity_t	identity	<p>Identity of the DDS sample being written. The identity consists of a pair (Virtual Writer GUID, Virtual Sequence Number).</p> <p>When the value DDS_AUTO_SAMPLE_IDENTITY is used, the <code>write_w_params()</code> operation will determine the DDS sample identity as follows:</p> <ul style="list-style-type: none"> The Virtual Writer GUID (<code>writer_guid</code>) is the virtual GUID associated with the <i>DataWriter</i> writing the DDS sample. This virtual GUID is configured using the member <code>virtual_guid</code> in DATA_WRITER_PROTOCOL_STATUS (Section 6.3.6.3 on page 273). The Virtual Sequence Number (<code>sequence_number</code>) is increased by one with respect to the previous value. <p>The virtual sequence numbers for a given virtual GUID must be strictly monotonically increasing. If you try to write a DDS sample with a sequence number smaller or equal to the last sequence number, the write operation will fail.</p> <p>A <i>DataReader</i> can inspect the identity of a received DDS sample by accessing the fields <code>original_publication_virtual_guid</code> and <code>original_publication_virtual_sequence_number</code> in The SampleInfo Structure (Section 7.4.6 on page 504).</p>
DDS_SampleIdentity_t	related_sample_identity	<p>The identity of another DDS sample related to this one.</p> <p>The value of this field identifies another DDS sample that is logically related to the one that is written.</p> <p>For example, the <i>DataWriter</i> created by a Replier (sets Introduction to the Request-Reply Communication Pattern (Section Chapter 22 on page 874)) uses this field to associate the identity of the DDS request sample to reponse sample.</p> <p>To specify that there is no related DDS sample identity use the value DDS_UNKNOWN_SAMPLE_IDENTITY,</p> <p>A <i>DataReader</i> can inspect the related DDS sample identity of a received DDS sample by accessing the fields <code>related_original_publication_virtual_guid</code> and <code>related_original_publication_virtual_sequence_number</code> in The SampleInfo Structure (Section 7.4.6 on page 504).</p>
DDS_Time	source_timestamp	<p>Source timestamp that will be associated to the DDS sample that is written.</p> <p>If <code>source_timestamp</code> is set to DDS_TIMER_INVALID, the middleware will assign the value.</p> <p>A <i>DataReader</i> can inspect the <code>source_timestamp</code> value of a received DDS sample by accessing the field <code>source_timestamp</code> The SampleInfo Structure (Section 7.4.6 on page 504).</p>

Table 6.16 DDS_WriteParams_t

Type	Field Name	Description
DDS_InstanceHandle_t	handle	The instance handle. This value can be either the handle returned by a previous call to register_instance() or the special value DDS_HANDLE_NIL.
DDS_Long	priority	Positive integer designating the relative priority of the DDS sample, used to determine the transmission order of pending transmissions. To use publication priorities, the <i>DataWriter's</i> PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.18 on page 397) must be set for asynchronous publishing and the <i>DataWriter</i> must use a <i>FlowController</i> with a highest-priority first scheduling_policy. For Multi-channel <i>DataWriters</i> , the publication priority of a DDS sample may be used as a filter criteria for determining channel membership. For more information, see Prioritized DDS Samples (Section 6.6.4 on page 428) .
DDS_Long	flag	Flags for the DDS sample, represented as a 32-bit integer, of which only the 16 least-significant bits are used. RTI reserves least-significant bits [0-7] for middleware-specific usage. The application can use least-significant bits [8-15]. The first bit, REDELIVERED_SAMPLE, is reserved to mark a DDS sample as redelivered when using RTI Queuing Service. The second bit, INTERMEDIATE_REPLY_SEQUENCE_SAMPLE, is used to indicate that a response DDS sample is not the last response DDS sample for a given request. This bit is usually set by <i>Connex DDS Repliers</i> sending multiple responses for a request. The third bit, REPLICATE_SAMPLE, indicates if a sample must be broad- cast by one Queuing Service replica to other replicas. The fourth bit, LAST_SHARED_READER_QUEUE_SAMPLE, indicates that a sample is the last sample in a SharedReaderQueue for a QueueConsumer <i>DataReader</i> . An application can inspect the flags associated with a received DDS sample by checking the field flag field in The SampleInfo Structure (Section 7.4.6 on page 504) . Default 0 (no flags are set)
struct DDS_GUID_t	source_guid	Identifies the application logical data source associated with the sample being written.
struct DDS_GUID_t	related_source_guid	Identifies the application logical data source that is related to the sample being written.
struct DDS_GUID_t	related_reader_guid	Identifies a <i>DataReader</i> that is logically related to the sample that is being written.

Note: Prioritized DDS samples are *not* supported when using the Java, Ada, or .NET APIs. Therefore the **priority** field in **DDS_WriteParams_t** does not exist when using these APIs.

When using the C API, a newly created variable of type **DDS_WriteParams_t** should be initialized by setting it to **DDS_WRITEPARAMS_DEFAULT**.

The **write()** operation also asserts liveness on the *DataWriter*, the associated *Publisher*, and the associated *DomainParticipant*. It has the same effect with regards to liveness as an explicit call to **assert_liveliness()**, see [Asserting Liveness \(Section 6.3.17 on page 311\)](#) and the **LIVELINESS QoS Policy (Section 6.5.13 on page 382)**. Maintaining liveness is important for *DataReaders* to know that the *DataWriter* still exists and for the proper behavior of the **OWNERSHIP QoS Policy (Section 6.5.15 on page 389)**.

See also: [Clock Selection \(Section 8.6 on page 619\)](#).

6.3.8.1 Blocking During a write()

The **write()** operation may block if the **RELIABILITY QoS Policy (Section 6.5.19 on page 400)** *kind* is set to **Reliable** and the modification would cause data to be lost or cause one of the limits specified in the **RESOURCE_LIMITS QoS Policy (Section 6.5.20 on page 405)** to be exceeded. Specifically, **write()** may block in the following situations (note that the list may not be exhaustive), even if its **HISTORY QoS Policy (Section 6.5.10 on page 376)** is **KEEP_LAST**:

- If $\text{max_samples}^1 < \text{max_instances}$, the *DataWriter* may block regardless of the **depth** field in the **HISTORY QoS Policy (Section 6.5.10 on page 376)**.
- If $\text{max_samples} < (\text{max_instances} * \text{depth})$, in the situation where the **max_samples** resource limit is exhausted, Connex DDS may discard DDS samples of some other instance, as long as at least one DDS sample remains for such an instance. If it is still not possible to make space available to store the modification, the writer is allowed to block.
- If $\text{min_send_window_size} < \text{max_samples}$, it is possible for the **send_window_size** limit to be reached before Connex DDS is allowed to discard DDS samples, in which case the *DataWriter* will block.

This operation may also block when using **BEST_EFFORT Reliability (RELIABILITY QoS Policy (Section 6.5.19 on page 400))** and **ASYNCHRONOUS Publish Mode (PUBLISH_MODE QoS Policy (DDS Extension) (Section 6.5.18 on page 397))** QoS settings. In this case, the *DataWriter* will queue DDS samples until they are sent by the asynchronous publishing thread. The number of DDS samples that can be stored is determined by the **HISTORY QoS Policy (Section 6.5.10 on page 376)**. If the asynchronous thread does not send DDS samples fast enough (such as when using a slow **FlowController (FlowControllers (DDS Extension) (Section 6.6 on page 422))**), the queue may fill up. In that case, subsequent write calls will block.

¹max_samples in is DDS_ResourceLimitsQoS Policy

If this operation does block for any of the above reasons, the RELIABILITY **max_blocking_time** configures the maximum time the write operation may block (waiting for space to become available). If **max_blocking_time** elapses before the *DataWriter* can store the modification without exceeding the limits, the operation will fail and return **RETCODE_TIMEOUT**.

6.3.9 Flushing Batches of DDS Data Samples

The **flush()** operation makes a batch of DDS data samples available to be sent on the network.

```
DDS_ReturnCode_t flush ()
```

If the *DataWriter's* **PUBLISH_MODE QosPolicy (DDS Extension)** (Section 6.5.18 on page 397) **kind** is **not ASYNCHRONOUS**, the batch will be sent on the network immediately in the context of the calling thread.

If the *DataWriter's* **PublishModeQosPolicy kind** is **ASYNCHRONOUS**, the batch will be sent in the context of the asynchronous publishing thread.

The **flush()** operation may block based on the conditions described in **Blocking During a write()** (Section 6.3.8.1 on the previous page).

If this operation does block, the **max_blocking_time** in the **RELIABILITY QosPolicy** (Section 6.5.19 on page 400) configures the maximum time the write operation may block (waiting for space to become available). If **max_blocking_time** elapses before the *DataWriter* is able to store the modification without exceeding the limits, the operation will fail and return **TIMEOUT**.

For more information on batching, see the **BATCH QosPolicy (DDS Extension)** (Section 6.5.2 on page 341).

6.3.10 Writing Coherent Sets of DDS Data Samples

A publishing application can request that a set of DDS data-sample changes be propagated in such a way that they are interpreted at the receivers' side as a cohesive set of modifications. In this case, the receiver will only be able to access the data after all the modifications in the set are available at the subscribing end.

This is useful in cases where the values are inter-related. For example, suppose you have two data-instances representing the 'altitude' and 'velocity vector' of the same aircraft. If both are changed, it may be important to ensure that reader see both together (otherwise, it may erroneously interpret that the aircraft is on a collision course).

To use this mechanism in C, Traditional C++, Java and .NET:

1. Call the *Publisher's* **begin_coherent_changes()** operation to indicate the start a coherent set.
2. For each DDS sample in the coherent set: call the *FooDataWriter's* **write()** operation.
3. Call the *Publisher's* **end_coherent_changes()** operation to terminate the set.

In the Modern C++ API:

1. Instantiate a `dds::pub::CoherentSet` passing a publisher to the constructor
2. For each DDS sample in the coherent set call `dds::pub::DataWriter<Foo>::write()`.
3. Let the `dds::pub::CoherentSet` destructor terminate the set or explicitly call `dds::pub::CoherentSet::end()`

Calls to `begin_coherent_changes()` and `end_coherent_changes()` can be nested.

See also: the `coherent_access` field in the [PRESENTATION QosPolicy](#) (Section 6.4.6 on page 330).

6.3.11 Waiting for Acknowledgments in a DataWriter

The *DataWriter's* `wait_for_acknowledgments()` operation blocks the calling thread until either all data written by the reliable *DataWriter* is acknowledged by (a) all reliable *DataReaders* that are matched and alive *and* (b) by all required subscriptions (see [Required Subscriptions](#) (Section 6.3.13 on page 294)), or until the duration specified by the `max_wait` parameter elapses, whichever happens first.

Note that if a thread is blocked in the call to `wait_for_acknowledgments()` on a *DataWriter* and a different thread writes new DDS samples on the same *DataWriter*, the new DDS samples must be acknowledged before unblocking the thread waiting on `wait_for_acknowledgments()`.

```
DDS_ReturnCode_t wait_for_acknowledgments (
    const DDS_Duration_t & max_wait)
```

This operation returns `DDS_RETCODE_OK` if all the DDS samples were acknowledged, or `DDS_RETCODE_TIMEOUT` if the `max_wait` duration expired first.

If the *DataWriter* does not have its [RELIABILITY QosPolicy](#) (Section 6.5.19 on page 400) `kind` set to `RELIABLE`, the operation will immediately return `DDS_RETCODE_OK`.

There is a similar operation available at the *Publisher* level, see [Waiting for Acknowledgments in a Publisher](#) (Section 6.2.7 on page 260).

The reliability protocol used by Connex DDS is discussed in [Reliable Communications](#) (Section Chapter 10 on page 629). The application acknowledgment mechanism is discussed in [Application Acknowledgment](#) (Section 6.3.12 below) and [Guaranteed Delivery of Data](#) (Section Chapter 13 on page 695).

6.3.12 Application Acknowledgment

The [RELIABILITY QosPolicy](#) (Section 6.5.19 on page 400) determines whether or not data published by a *DataWriter* will be reliably delivered by *Connex DDS* to matching *DataReaders*. The reliability protocol used by *Connex DDS* is discussed in [Reliable Communications](#) (Section Chapter 10 on page 629).

With protocol-level reliability alone, the producing application knows that the information is received by the protocol layer on the consuming side. However, the producing application cannot be certain that the

consuming application read that information or was able to successfully understand and process it. The information could arrive in the consumer's protocol stack and be placed in the *DataReader* cache but the consuming application could either crash before it reads it from the cache, not read its cache, or read the cache using queries or conditions that prevent that particular DDS data sample from being accessed. Furthermore, the consuming application could access the DDS sample, but not be able to interpret its meaning or process it in the intended way.

The mechanism to let a *DataWriter* know to keep the DDS sample around, not just until it has been acknowledged by the reliability protocol, but until the application has been able to process the DDS sample is aptly called *Application Acknowledgment*. A reliable *DataWriter* will keep the DDS samples until the application acknowledges the DDS samples. When the subscriber application is restarted, the middleware will know that the application did not acknowledge successfully processing the DDS samples and will resend them.

6.3.12.1 Application Acknowledgment Kinds

Connex DDS supports *three* kinds of application acknowledgment, which is configured in the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#):

1. `DDS_PROTOCOL_ACKNOWLEDGMENT_MODE` (Default): In essence, this mode is identical to using no application-level acknowledgment. DDS samples are acknowledged according to the Real-Time Publish-Subscribe (RTPS) reliability protocol. RTPS AckNack messages will acknowledge that the middleware received the DDS sample.
2. `DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE`: DDS samples are automatically acknowledged by the middleware after the subscribing application accesses them, either through calling `take()` or `read()` on the DDS sample. The DDS samples are acknowledged after `return_loan()` is called.
3. `DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE`: DDS samples are acknowledged after the subscribing application explicitly calls `acknowledge` on the DDS sample. This can be done by either calling the *DataReader's* `acknowledge_sample()` or `acknowledge_all()` operations. When using `acknowledge_sample()`, the application will provide the `DDS_SampleInfo` to identify the DDS sample being acknowledged. When using `acknowledge_all`, all the DDS samples that have been read or taken by the reader will be acknowledged.

Note: Even in `DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE`, some DDS samples may be automatically acknowledged. This is the case when DDS samples are filtered out by the reader using time-based filter, or using content filters. Additionally, when the reader is explicitly configured to use `KEEP_LAST` history kind, DDS samples may be replaced in the reader queue due to resource constraints. In that case, the DDS sample will be automatically acknowledged by the middleware if it has not been read by the application before it was replaced. To truly guarantee successful processing of DDS samples, it is recommended to use `KEEP_ALL` history kind.

6.3.12.2 Explicitly Acknowledging a Single DDS Sample (C++)

```

void MyReaderListener::on_data_available(DDSDataReader *reader)
{
    Foo sample;
    DDS_SampleInfo info;
    FooDataReader* fooReader = FooDataReader::narrow(reader);
    DDS_ReturnCode_t retcode = fooReader->take_next_sample(
        sample, info);
    if (retcode == DDS_RETCODE_OK) {
        if (info.valid_data) {
            // Process sample
            ...
            retcode = reader->acknowledge_sample(info);
            if (retcode != DDS_RETCODE_OK) {
                // Error
            }
        }
    } else {
        // Not OK or NO DATA
    }
}

```

6.3.12.3 Explicitly Acknowledging All DDS samples (C++)

```

void MyReaderListener::on_data_available(DDSDataReader *reader)
{
    ...
    // Loop while samples available
    for(;;) {
        retcode = string_reader->take_next_sample(
            sample, info);
        if (retcode == DDS_RETCODE_NO_DATA) {
            // No more samples
            break;
        }
        // Process sample
        ...
    }
    retcode = reader->acknowledge_all();
    if (retcode != DDS_RETCODE_OK) {
        // Error
    }
}

```

6.3.12.4 Notification of Delivery with Application Acknowledgment

A *DataWriter* can get notification of delivery with Application Acknowledgment using two different mechanisms:

- *DataWriter's* **wait_for_acknowledgments()** operation

A *DataWriter* can use the **wait_for_acknowledgments()** operation to be notified when all the DDS samples in the *DataWriter's* queue have been acknowledged. See [Waiting for Acknowledgments in a DataWriter \(Section 6.3.11 on page 288\)](#).

```
retCode = fooWriter->write(sample, DDS_HANDLE_NIL);
if (retCode != DDS_RETCODE_OK) {
    // Error
}
retcode = writer->wait_for_acknowledgments(timeout);
if (retCode != DDS_RETCODE_OK) {
    if (retCode == DDS_RETCODE_TIMEOUT) {
        // Timeout: Sample not acknowledged yet
    } else {
        // Error
    }
}
}
```

Using **wait_for_acknowledgments()** does not provide a way to get delivery notifications on a per *DataReader* and DDS sample basis. If your application requires acknowledgment of message receipt, use the the second mechanism described below.

- *DataWriter's* listener callback **on_application_acknowledgment()**

An application can install a *DataWriter* listener callback **on_application_acknowledgment()** to receive a notification when a DDS sample is acknowledged by a *DataReader*. As part of this notification, you can access:

- The subscription handle of the acknowledging *DataReader*.
- The Identity of the DDS sample being acknowledged.
- The response data associated with the DDS sample being acknowledged.

For more information, see [APPLICATION_ACKNOWLEDGMENT_STATUS \(Section 6.3.6.1 on page 272\)](#).

6.3.12.5 Application-Level Acknowledgment Protocol

When the subscribing application confirms it has successfully processed a DDS sample, an AppAck RTPS message is sent to the publishing application. This message will be resent until the publishing application confirms receipt of the AppAck message by sending an AppAckConf RTPS message. See [Figures Figure 6.10 AppAck RTPS Messages Sent when Application Acknowledges a DDS Sample on the facing page through Figure 6.12 AppAck RTPS Messages Sent as a Sequence of Intervals, Combined to Optimize for Bandwidth on page 293](#).

Figure 6.10 AppAck RTPS Messages Sent when Application Acknowledges a DDS Sample

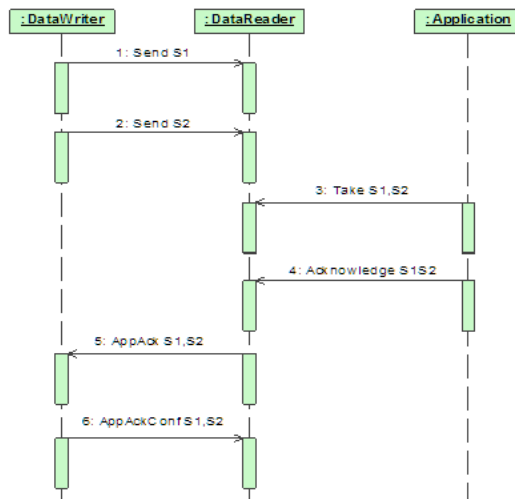


Figure 6.11 AppAck RTPS Messages Resent Until Acknowledged Through AppAckConf RTPS Message

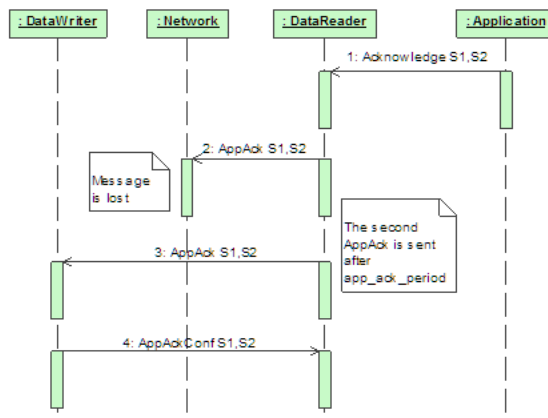
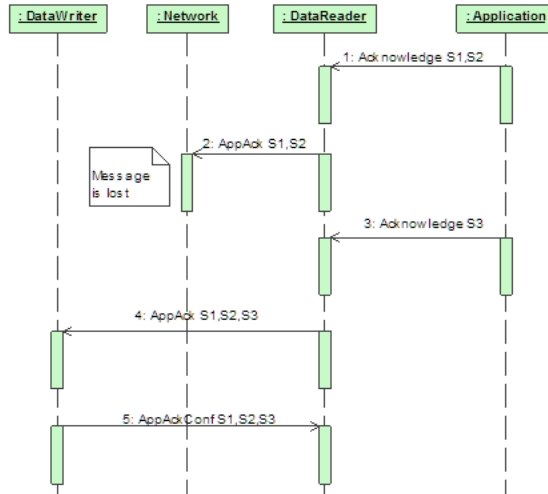


Figure 6.12 AppAck RTPS Messages Sent as a Sequence of Intervals, Combined to Optimize for Bandwidth



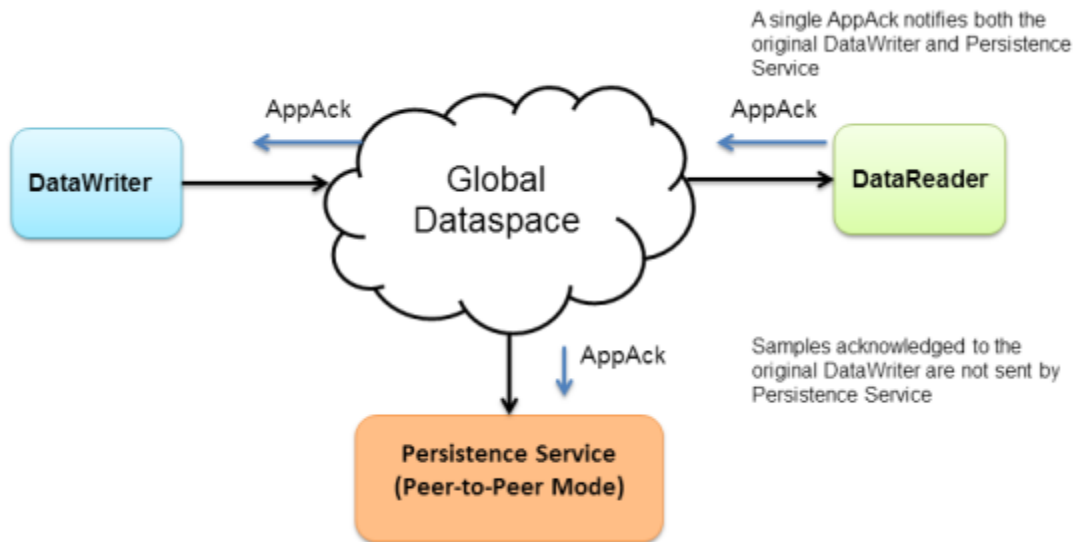
6.3.12.6 Periodic and Non-Periodic AppAck Messages

You can configure whether AppAck RTPS messages are sent immediately or periodically through the [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 7.6.1 on page 511). The [samples_per_app_ack](#) (Section on page 515) (in Table 7.20 `DDS_RtpsReliableReaderProtocol_t`) determines the minimum number of DDS samples acknowledged by one application-level Acknowledgment message. The middleware will not send an AppAck message until it has at least this many DDS samples pending acknowledgment. By default, `samples_per_app_ack` is 1 and the AppAck RTPS message is sent immediately. Independently, the [app_ack_period](#) (Section on page 514) (in Table 7.20 `DDS_RtpsReliableReaderProtocol_t`) determines the rate at which a *DataReader* will send AppAck messages.

6.3.12.7 Application Acknowledgment and Persistence Service

Application Acknowledgment is fully supported by *RTI Persistence Service*. The combination of Application Acknowledgment and *Persistence Service* is actually a common configuration. In addition to keeping DDS samples available until fully acknowledged, *Persistence Service*, when used in peer-to-peer mode, can take advantage of AppAck messages to avoid sending duplicate messages to the subscribing application. Because AppAck messages are sent to all matching writers, when the subscriber acknowledges the original publisher, *Persistence Service* will also be notified of this event and will not send out duplicate messages. This is illustrated in [Figure 6.13 Application Acknowledgment and Persistence Service on the facing page](#).

Figure 6.13 Application Acknowledgment and Persistence Service



6.3.12.8 Application Acknowledgment and Routing Service

Application Acknowledgment is supported by *RTI Routing Service*: That is, *Routing Service* will acknowledge the DDS sample it has processed. *Routing Service* is an active participant in the *Connex DDS* system and not transparent to the publisher or subscriber. As such, *Routing Service* will acknowledge to the publisher, and the subscriber will acknowledge to *Routing Service*. However, the publisher will not get a notification from the subscriber directly.

6.3.13 Required Subscriptions

The [DURABILITY QoS Policy \(Section 6.5.7 on page 368\)](#) specifies whether acknowledged DDS samples need to be kept in the *DataWriter's* queue and made available to late-joining applications. When a late joining application is discovered, available DDS samples will be sent to the late joiner. With the Durability QoS alone, there is no way to specify or characterize the intended consumers of the information and you do not have control over which DDS samples will be kept for late-joining applications. If while waiting for late-joining applications, the middleware needs to free up DDS samples, it will reclaim DDS samples if they have been previously acknowledged by active/matching readers.

There are scenarios where you know a priori that a particular set of applications will join the system: e.g., a logging service or a known processing application. The *Required Subscription* feature is designed to keep data until these known late joining applications acknowledge the data.

Another use case is when *DataReaders* become temporarily inactive due to not responding to heartbeats, or when the subscriber temporarily became disconnected and purged from the discovery database. In both cases, the *DataWriter* will no longer keep the DDS sample for this *DataReader*. The *Required Subscription* feature will keep the data until these known *DataReaders* have acknowledged the data.

To use Required Subscriptions, the *DataReaders* and *DataWriters* must have their [RELIABILITY QoSPolicy \(Section 6.5.19 on page 400\)](#) **kind** set to RELIABLE.

6.3.13.1 Named, Required and Durable Subscriptions

Before describing the Required Subscriptions, it is important to understand a few concepts:

- **Named Subscription:** Through the [ENTITY_NAME QoSPolicy \(DDS Extension\) \(Section 6.5.9 on page 374\)](#), each *DataReader* can be given a specific name. This name can be used by tools to identify a specific *DataReader*. Additionally, the *DataReader* can be given a `role_name`. For example: LOG_APP_1 *DataReader* belongs to the logger applications (**role_name** = “LOGGER”).
- **Required Subscription** is a named subscription to which a *DataWriter* is configured to deliver data to. This is true even if the *DataReaders* serving those subscriptions are not available yet. The *DataWriter* must store the DDS sample until it has been acknowledged by all active reliable *DataReaders* and acknowledged by all required subscriptions. The *DataWriter* is not waiting for a specific *DataReader*, rather it is waiting for *DataReaders* belonging to the required subscription by setting their **role_name** to the subscription name.
- **Durable Subscription** is a required subscription where DDS samples are stored and forwarded by an external service. In this case, the required subscription is served by *RTI Persistence Service*. See [Configuring Durable Subscriptions in Persistence Service \(Section 27.9 on page 955\)](#).

6.3.13.2 Durability QoS and Required Subscriptions

The [DURABILITY QoSPolicy \(Section 6.5.7 on page 368\)](#) and the Required Subscriptions feature complement each other.

The *DurabilityQoSPolicy* determines whether or not *Connex DDS* will store and deliver previously acknowledged DDS samples to new *DataReaders* that join the network later. You can specify to either *not* make the DDS samples available (DDS_VOLATILE_DURABILITY_QOS kind), or to make them available and declare you are storing the DDS samples in memory (DDS_TRANSIENT_LOCAL_DURABILITY_QOS or DDS_TRANSIENT_DURABILITY_QOS kind) or in permanent storage (DDS_PERSISTENT_DURABILITY_QOS).

Required subscriptions help answer the question of *when* a DDS sample is considered acknowledged before the *DurabilityQoSPolicy* determines whether to keep it. When required subscriptions are used, a DDS sample is considered acknowledged by a *DataWriter* when both the active *DataReaders* and a quorum of required subscriptions have acknowledged the DDS sample. (Acknowledging a DDS sample

can be done either at the protocol or application level—see [Application Acknowledgment \(Section 6.3.12 on page 288\)](#)).

6.3.13.3 Required Subscriptions Configuration

Each *DataReader* can be configured to be part of a named subscription, by giving it a **role_name** using the [ENTITY_NAME QosPolicy \(DDS Extension\) \(Section 6.5.9 on page 374\)](#). A *DataWriter* can then be configured using the [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#) (**required_matched_endpoint_groups**) with a list of required named subscriptions identified by the **role_name**. Additionally, the *DataWriter* can be configured with a *quorum* or minimum number of *DataReaders* from a given named subscription that must receive a DDS sample.

When configured with a list of required subscriptions, a *DataWriter* will store a DDS sample until the DDS sample is acknowledged by all active reliable *DataReaders*, as well as all required subscriptions. When a quorum is specified, a minimum number of *DataReaders* of the required subscription must acknowledge a DDS sample in order for the DDS sample to be considered acknowledged. Specifying a quorum provides a level of redundancy in the system as multiple applications or services acknowledge they have received the DDS sample. Each individual *DataReader* is identified using its own virtual GUID (see [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#)).

6.3.14 Managing Data Instances (Working with Keyed Data Types)

This section applies only to data types that use keys, see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#). Using the following operations for non-keyed types has no effect.

Topics come in two flavors: those whose associated data type has specified some fields as defining the ‘key,’ and those whose associated data type has not. An example of a data-type that specifies key fields is shown in [Data Type with a Key \(Section Figure 6.14 below\)](#).

Figure 6.14 Data Type with a Key

```
typedef struct Flight {
    long    flightId; //@key
    string  departureAirport;
    string  arrivalAirport;
    Time_t  departureTime;
    Time_t  estimatedArrivalTime;
    Location_t  currentPosition;
};
```

If the data type has some fields that act as a ‘key,’ the *Topic* essentially defines a collection of data-instances whose values can be independently maintained. In [Figure 6.14 Data Type with a Key above](#), the **flightId** is the ‘key’. Different flights will have different values for the key. Each flight is an instance of the *Topic*. Each **write()** will update the information about a single flight. *DataReaders* can be informed when new flights appear or old ones disappear.

Since the key fields are contained within the data structure, Connex DDS could examine the key fields each time it needs to determine which data-instance is being modified. However, for performance and semantic reasons, it is better for your application to declare all the data-instances it intends to modify—prior to actually writing any DDS samples. This is known as *registration*, described below in [Registering and Unregistering Instances \(Section 6.3.14.1 below\)](#).

The **register_instance()** operation provides a handle to the instance (of type **DDS_InstanceHandle_t**) that can be used later to refer to the instance.

6.3.14.1 Registering and Unregistering Instances

If your data type has a key, you may improve performance by registering an instance (data associated with a particular value of the key) before you write data for the instance. You can do this for any number of instances up to the maximum number of instances configured in the *DataWriter*'s **RESOURCE_LIMITS QoSPolicy** ([Section 6.5.20 on page 405](#)). Instance registration is completely optional.

Registration tells Connex DDS that you are about to modify (write or dispose of) a specific instance. This allows Connex DDS to pre-configure itself to process that particular instance, which can improve performance.

If you write without registering, you can pass the NIL instance handle as part of the **write()** call.

If you register the instance first, Connex DDS can look up the instance beforehand and return a handle to that instance. Then when you pass this handle to the **write()** operation, Connex DDS no longer needs to analyze the data to check what instance it is for. Instead, it can directly update the instance pointed to by the instance handle.

In summary, by registering an instance, all subsequent **write()** calls to that instance become more efficient. If you only plan to write once to a particular instance, registration does not ‘buy’ you much in performance, but in general, it is good practice.

To register an instance, use the *DataWriter*'s **register_instance()** operation. For best performance, it should be invoked prior to calling any operation that modifies the instance, such as **write()**, **write_w_timestamp()**, **dispose()**, or **dispose_w_timestamp()**.

When you are done using that instance, you can unregister it. To unregister an instance, use the *DataWriter*'s **unregister_instance()** operation. Unregistering tells Connex DDS that the *DataWriter* does not intend to modify that data-instance anymore, allowing Connex DDS to recover any resources it allocated for the instance. It does not delete the instance; that is done with the **dispose_instance()** operation, see [Disposing of Data \(Section 6.3.14.2 on page 299\)](#). **autodispose_unregistered_instances** ([Section on page 419](#)) in the **WRITER_DATA_LIFECYCLE QoS Policy** ([Section 6.5.27 on page 419](#)) controls whether instances are automatically disposed when they are unregistered.

unregister_instance() should only be used on instances that have been previously registered. The use of these operations is illustrated in [Figure 6.15 Registering an Instance on the facing page](#).

Figure 6.15 Registering an Instance

```

Flight myFlight;
// writer is a previously-created FlightDataWriter
myFlight.flightId = 265;
DDS_InstanceHandle_t fl265Handle =
writer->register_instance(myFlight);
...
// Each time we update the flight, we can pass the handle
myFlight.departureAirport    = "SJC";
myFlight.arrivalAirport     = "LAX";
myFlight.departureTime      = {120000, 0};
myFlight.estimatedArrivalTime = {130200, 0};
myFlight.currentPosition    = { {37, 20}, {121, 53} };
if (writer->write(myFlight, fl265Handle) != DDS_RETCODE_OK) {
// ... handle error
}
// After updating the flight, it can be unregistered
if (writer->unregister_instance(myFlight, fl265Handle) !=
DDS_RETCODE_OK) {
// ... handle error
}

```

Once an instance has been unregistered, and assuming that no other *DataWriters* are writing values for the instance, the matched *DataReaders* will eventually get an indication that the instance no longer has any *DataWriters*. This is communicated to the *DataReaders* by means of the **DDS_SampleInfo** that accompanies each DDS data-sample (see [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#)). Once there are no *DataWriters* for the instance, the *DataReader* will see the value of **DDS_InstanceStateKind** for that instance to be **NOT_ALIVE_NO_WRITERS**.

The **unregister_instance()** operation may affect the ownership of the data instance (see the [OWNERSHIP QosPolicy \(Section 6.5.15 on page 389\)](#)). If the **DataWriter** was the exclusive owner of the instance, then calling **unregister_instance()** relinquishes that ownership, and another *DataWriter* can become the exclusive owner of the instance.

The **unregister_instance()** operation indicates only that a particular *DataWriter* no longer has anything to say about the instance.

Note that this is different than the **dispose()** operation discussed in the next section, which informs *DataReaders* that the data-instance is no longer “alive.” The state of an instance is stored in the **DDS_SampleInfo** structure that accompanies each DDS sample of data that is received by a *DataReader*. User code can access the instance state to see if an instance is “alive”—meaning there is at least one *DataWriter* that is publishing DDS samples for the instance, see [Instance States \(Section 7.4.6.4 on page 507\)](#).

See also:

- [Unregistering vs. Disposing: \(Section on page 420\)](#).
- [Use Cases for Unregistering without Disposing: \(Section on page 420\)](#).

6.3.14.2 Disposing of Data

The **dispose()** operation informs *DataReaders* that, as far as the *DataWriter* knows, the data-instance no longer exists and can be considered “not alive.” When the **dispose()** operation is called, the instance state stored in the **DDS_SampleInfo** structure, accessed through *DataReaders*, will change to **NOT_ALIVE_DISPOSED** for that particular instance.

See [Unregistering vs. Disposing: \(Section on page 420\)](#).

By default, instances are automatically disposed when they are unregistered. This behavior is controlled by the **autodispose_unregistered_instances** ([Section on page 419](#)) field in the **WRITER_DATA_LIFECYCLE** QoS Policy ([Section 6.5.27 on page 419](#)).

For example, in a flight tracking system, when a flight lands, a *DataWriter* may dispose the data-instance corresponding to the flight. In that case, all *DataReaders* who are monitoring the flight will see the instance state change to **NOT_ALIVE_DISPOSED**, indicating that the flight has landed.

If a particular instance is never disposed, its instance state will eventually change from **ALIVE** to **NOT_ALIVE_NO_WRITERS** once all the *DataWriters* that were writing that instance unregister the instance or lose their liveliness. For more information on *DataWriter* liveliness, see the **LIVELINESS** QoS Policy ([Section 6.5.13 on page 382](#)).

See also:

- [Propagating Serialized Keys with Disposed-Instance Notifications \(Section 6.5.3.5 on page 356\)](#).
- [Use Cases for Unregistering without Disposing: \(Section on page 420\)](#).

6.3.14.3 Looking Up an Instance Handle

Some operations, such as **write()**, require an **instance_handle** parameter. If you need to get such as handle, you can call the *FooDataWriter*'s **lookup_instance()** operation, which takes an instance as a parameter and returns a handle to that instance. This is useful for keyed data types.

```
DDS_InstanceHandle_t lookup_instance (const Foo & key_holder)
```

The instance must have already been registered (see [Registering and Unregistering Instances \(Section 6.3.14.1 on page 297\)](#)). If the instance is not registered, this operation returns **DDS_HANDLE_NIL**.

6.3.14.4 Getting the Key Value for an Instance

Once you have an instance handle (using **register_instance()** or **lookup_instance()**), you can use the *DataWriter*'s **get_key_value()** operation to retrieve the value of the key of the corresponding instance.

The key fields of the data structure passed into `get_key_value()` will be filled out with the original values used to generate the instance handle. The key fields are defined when the data type is defined, see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#) for more information.

Following our example in [Figure 6.15 Registering an Instance on page 298](#), `register_instance()` returns a `DDS_InstanceHandle_t` (`fl265Handle`) that can be used in the call to the `FlightDataWriter`'s `get_key_value()` operation. The value of the key is returned in a structure of type `Flight` with the `flightId` field filled in with the integer 265.

See also: [Propagating Serialized Keys with Disposed-Instance Notifications \(Section 6.5.3.5 on page 356\)](#).

6.3.15 Setting DataWriter QoS Policies

The *DataWriter*'s QoS Policies control its resources and behavior.

The `DDS_DataWriterQos` structure has the following format:

```
DDS_DataWriterQos struct {
    DDS_DurabilityQosPolicy           durability;
    DDS_DurabilityServiceQosPolicy   durability_service;
    DDS_DeadlineQosPolicy             deadline;
    DDS_LatencyBudgetQosPolicy        latency_budget;
    DDS_LivelinessQosPolicy           liveliness;
    DDS_ReliabilityQosPolicy          reliability;
    DDS_DestinationOrderQosPolicy     destination_order;
    DDS_HistoryQosPolicy              history;
    DDS_ResourceLimitsQosPolicy       resource_limits;
    DDS_TransportPriorityQosPolicy     transport_priority;
    DDS_LifespanQosPolicy              lifespan;
    DDS_UserDataQosPolicy              user_data;
    DDS_OwnershipQosPolicy             ownership;
    DDS_OwnershipStrengthQosPolicy    ownership_strength;
    DDS_WriterDataLifecycleQosPolicy  writer_data_lifecycle;
    // extensions to the DDS standard:
    DDS_DataWriterResourceLimitsQosPolicy  writer_resource_limits;
    DDS_DataWriterProtocolQosPolicy        protocol;
    DDS_TransportSelectionQosPolicy        transport_selection;
    DDS_TransportUnicastQosPolicy          unicast;
    DDS_PublishModeQosPolicy               publish_mode;
    DDS_PropertyQosPolicy                  property;
    DDS_ServiceQosPolicy                   service;
    DDS_BatchQosPolicy                     batch;
    DDS_MultiChannelQosPolicy              multi_channel;
    DDS_AvailabilityQosPolicy              availability;
    DDS_EntityNameQosPolicy                publication_name;
    DDS_TypeSupportQosPolicy               type_support;
} DDS_DataWriterQos;
```

Note: `set_qos()` cannot always be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

[Table 6.17 DataWriter QoS Policies](#) summarizes the meaning of each policy. (They appear alphabetically in the table.) For information on *why* you would want to change a particular QoS Policy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.17 DataWriter QoS Policies

QoS Policy	Description
Availability	<p>This QoS policy is used in the context of two features:</p> <p>Availability QoS Policy and Collaborative DataWriters (Section 6.5.1.1 on page 338)</p> <p>AVAILABILITY QoS Policy (DDS Extension) (Section 6.5.1 on page 337)</p> <p>For Collaborative DataWriters, Availability specifies the group of <i>DataWriters</i> expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip DDS samples.</p> <p>For Required Subscriptions, Availability configures a set of Required Subscriptions on a <i>DataWriter</i>.</p> <p>See AVAILABILITY QoS Policy (DDS Extension) (Section 6.5.1 on page 337)</p>
Batch	<p>Specifies and configures the mechanism that allows Connex DDS to collect multiple DDS user data samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput. See BATCH QoS Policy (DDS Extension) (Section 6.5.2 on page 341).</p>
DataWriterProtocol	<p>This QoS Policy configures the Connex DDS on-the-network protocol, RTPS. See DATA_WRITER_PROTOCOL QoS Policy (DDS Extension) (Section 6.5.3 on page 347).</p>
DataWriterResourceLimits	<p>Controls how many threads can concurrently block on a write() call of this <i>DataWriter</i>. See DATA_WRITER_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 6.5.4 on page 359).</p>
Deadline	<p>For a <i>DataReader</i>, it specifies the maximum expected elapsed time between arriving DDS data samples.</p> <p>For a <i>DataWriter</i>, it specifies a commitment to publish DDS samples with no greater elapsed time between them.</p> <p>See DEADLINE QoS Policy (Section 6.5.5 on page 363).</p>
DestinationOrder	<p>Controls how Connex DDS will deal with data sent by multiple <i>DataWriters</i> for the same topic. Can be set to "by reception timestamp" or to "by source timestamp". See DESTINATION_ORDER QoS Policy (Section 6.5.6 on page 365).</p>
Durability	<p>Specifies whether or not Connex DDS will store and deliver data that were previously published to new <i>DataReaders</i>. See DURABILITY QoS Policy (Section 6.5.7 on page 368).</p>
DurabilityService	<p>Various settings to configure the external Persistence Service¹ used by Connex DDS for <i>DataWriters</i> with a Durability QoS setting of Persistent Durability. See DURABILITY SERVICE QoS Policy (Section 6.5.8 on page 372).</p>

¹Persistence Service is provided with the Connex DDS Professional, Evaluation, and Basic package types.

Table 6.17 DataWriter QoS Policies

QoS Policy	Description
EntityName	Assigns a name to a <i>DataWriter</i> . See ENTITY_NAME QoS Policy (DDS Extension) (Section 6.5.9 on page 374) .
History	Specifies how much data must be stored by Connex DDS for the <i>DataWriter</i> or <i>DataReader</i> . This QoS Policy affects the RELIABILITY QoS Policy (Section 6.5.19 on page 400) as well as the DURABILITY QoS Policy (Section 6.5.7 on page 368) . See HISTORY QoS Policy (Section 6.5.10 on page 376) .
LatencyBudget	Suggestion to Connex DDS on how much time is allowed to deliver data. See LATENCYBUDGET QoS Policy (Section 6.5.11 on page 380) .
Lifespan	Specifies how long Connex DDS should consider data sent by an user application to be valid. See LIFESPAN QoS Policy (Section 6.5.12 on page 381) .
Liveliness	Specifies and configures the mechanism that allows <i>DataReaders</i> to detect when <i>DataWriters</i> become disconnected or "dead." See LIVELINESS QoS Policy (Section 6.5.13 on page 382) .
MultiChannel	Configures a <i>DataWriter's</i> ability to send data on different multicast groups (addresses) based on the value of the data. See MULTI_CHANNEL QoS Policy (DDS Extension) (Section 6.5.14 on page 386) .
Ownership	Along with OwnershipStrength, specifies if <i>DataReaders</i> for a topic can receive data from multiple <i>DataWriters</i> at the same time. See OWNERSHIP QoS Policy (Section 6.5.15 on page 389) .
OwnershipStrength	Used to arbitrate among multiple <i>DataWriters</i> of the same instance of a Topic when Ownership QoS Policy is EXCLUSIVE. See OWNERSHIP_STRENGTH QoS Policy (Section 6.5.16 on page 393) .
Partition	Adds string identifiers that are used for matching <i>DataReaders</i> and <i>DataWriters</i> for the same <i>Topic</i> . See PARTITION QoS Policy (Section 6.4.5 on page 323) .
Property	Stores name/value (string) pairs that can be used to configure certain parameters of Connex DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See PROPERTY QoS Policy (DDS Extension) (Section 6.5.17 on page 394) .
PublishMode	Specifies how Connex DDS sends application data on the network. By default, data is sent in the user thread that calls the <i>DataWriter's</i> <code>write()</code> operation. However, this QoS Policy can be used to tell Connex DDS to use its own thread to send the data. See PUBLISH_MODE QoS Policy (DDS Extension) (Section 6.5.18 on page 397) .
Reliability	Specifies whether or not Connex DDS will deliver data reliably. See RELIABILITY QoS Policy (Section 6.5.19 on page 400) .
ResourceLimits	Controls the amount of physical memory allocated for <i>Entities</i> , if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See RESOURCE_LIMITS QoS Policy (Section 6.5.20 on page 405) .
Service	Intended for use by RTI infrastructure services. User applications should not modify its value. See SERVICE QoS Policy (DDS Extension) (Section 6.5.21 on page 408) .

Table 6.17 DataWriter QoS Policies

QoS Policy	Description
TransportPriority	Set by a <i>DataWriter</i> to tell Connex DDS that the data being sent is a different "priority" than other data. See TRANSPORT_PRIORITY QoS Policy (Section 6.5.22 on page 409) .
TransportSelection	Allows you to select which physical transports a <i>DataWriter</i> or <i>DataReader</i> may use to send or receive its data. See TRANSPORT_SELECTION QoS Policy (DDS Extension) (Section 6.5.23 on page 411) .
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See TRANSPORT_UNICAST QoS Policy (DDS Extension) (Section 6.5.24 on page 412) .
TypeSupport	Used to attach application-specific value(s) to a <i>DataWriter</i> or <i>DataReader</i> . These values are passed to the serialization or deserialization routine of the associated data type. Also controls whether padding bytes are set to 0 during serialization. See TYPESUPPORT QoS Policy (DDS Extension) (Section 6.5.25 on page 416) .
UserData	Along with Topic Data QoS Policy and Group Data QoS Policy, used to attach a buffer of bytes to Connex DDS's discovery meta-data. See USER_DATA QoS Policy (Section 6.5.26 on page 417) .
WriterDataLifeCycle	Controls how a <i>DataWriter</i> handles the lifecycle of the instances (keys) that the <i>DataWriter</i> is registered to manage. See WRITER_DATA_LIFECYCLE QoS Policy (Section 6.5.27 on page 419) .

Many of the *DataWriter* QoS Policies also apply to *DataReaders* (see [DataReaders \(Section 7.3 on page 459\)](#)). For a *DataWriter* to communicate with a *DataReader*, their QoS Policies must be compatible. Generally, for the QoS Policies that apply both to the *DataWriter* and the *DataReader*, the setting in the *DataWriter* is considered an “offer” and the setting in the *DataReader* is a “request.” Compatibility means that what is offered by the *DataWriter* equals or surpasses what is requested by the *DataReader*. Each policy’s description includes compatibility restrictions. For more information on compatibility, see [QoS Requested vs. Offered Compatibility—the RxO Property \(Section 4.2.1 on page 167\)](#).

Some of the policies may be changed after the *DataWriter* has been created. This allows the application to modify the behavior of the *DataWriter* while it is in use. To modify the QoS of an already-created *DataWriter*, use the `get_qos()` and `set_qos()` operations on the *DataWriter*. This is a general pattern for all *Entities*, described in [Changing the QoS for an Existing Entity \(Section 4.1.7.3 on page 161\)](#).

6.3.15.1 Configuring QoS Settings when the DataWriter is Created

As described in [Creating DataWriters \(Section 6.3.1 on page 266\)](#), there are different ways to create a *DataWriter*, depending on how you want to specify its QoS (with or without a QoS Profile).

- In [Creating a DataWriter with Default QoS Policies and a Listener \(Section Figure 6.9 on page 268\)](#), there is an example of how to create a *DataWriter* with default QoS Policies by using the special constant, `DDS_DATAWRITER_QOS_DEFAULT`, which indicates that the default QoS values for a *DataWriter* should be used. The default *DataWriter* QoS values are configured in the *Publisher* or *DomainParticipant*; you can change them with `set_default_datawriter_qos()` or `set_default_`

datawriter_qos_with_profile(). Then any *DataWriters* created with the *Publisher* will use the new default values. As described in [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#), this is a general pattern that applies to the construction of all *Entities*.

- To create a *DataWriter* with non-default QoS without using a QoS Profile, see the example code in [Figure 6.16 Creating a DataWriter with Modified QoS Policies \(not from a profile\) below](#). It uses the *Publisher's* **get_default_writer_qos()** method to initialize a **DDS_DataWriterQos** structure. Then the policies are modified from their default values before the structure is used in the **create_datawriter()** method.
- You can also create a *DataWriter* and specify its QoS settings via a QoS Profile. To do so, you will call **create_datawriter_with_profile()**, as seen in [Figure 6.17 Creating a DataWriter with a QoS Profile on the next page](#).
- If you want to use a QoS profile, but then make some changes to the QoS before creating the *DataWriter*, call **get_datawriter_qos_from_profile()** and **create_datawriter()** as seen in [Figure 6.18 Getting QoS Values from a Profile, Changing QoS Values, Creating a DataWriter with Modified QoS Values on the next page](#).

For more information, see [Creating DataWriters \(Section 6.3.1 on page 266\)](#) and [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 6.16 Creating a DataWriter with Modified QoS Policies (not from a profile)

```
DDS_DataWriterQos writer_qos;1
// initialize writer_qos with default values
publisher->get_default_datawriter_qos(writer_qos);
// make QoS changes
writer_qos.history.depth = 5;
// Create the writer with modified qos
DDSDataWriter * writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);
if (writer == NULL) {
    // ... error
}
// narrow it for your specific data type
FooDataWriter* foo_writer = FooDataWriter::narrow(writer);
```

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

Figure 6.17 Creating a DataWriter with a QoS Profile

```

// Create the datawriter
DDSDataWriter * writer =
    publisher->create_datawriter_with_profile(
        topic, "MyWriterLibrary", "MyWriterProfile",
        NULL, DDS_STATUS_MASK_NONE);
if (writer == NULL) {
    // ... error
};
// narrow it for your specific data type
FooDataWriter* foo_writer = FooDataWriter::narrow(writer);

```

Figure 6.18 Getting QoS Values from a Profile, Changing QoS Values, Creating a DataWriter with Modified QoS Values

```

DDS_DataWriterQos writer_qos;1
// Get writer QoS from profile
retcode = factory->get_datawriter_qos_from_profile(
    writer_qos, "WriterProfileLibrary", "WriterProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes
writer_qos.history.depth = 5;
DDSDataWriter * writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // handle error
}

```

6.3.15.2 Comparing QoS Values

The **equals()** operation compares two *DataWriter*'s `DDS_DataWriterQoS` structures for equality. It takes two parameters for the two *DataWriter*'s QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

6.3.15.3 Changing QoS Settings After the DataWriter Has Been Created

There are two ways to change an existing *DataWriter*'s QoS after it has been created—again depending on whether or not you are using a QoS Profile.

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

- To change QoS programmatically (that is, without using a QoS Profile), use `get_qos()` and `set_qos()`. See the example code in [Figure 6.19 Changing the QoS of an Existing DataWriter \(without a QoS Profile\) below](#). It retrieves the current values by calling the *DataWriter's* `get_qos()` operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QoS Policies cannot be changed after the *DataWriter* has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.
- You can also change a *DataWriter's* (and all other *Entities'*) QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see [Figure 6.20 Changing the QoS of an Existing DataWriter with a QoS Profile below](#). For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 6.19 Changing the QoS of an Existing DataWriter (without a QoS Profile)

```
DDS_DataWriterQos writer_qos;1
// Get current QoS.
if (datawriter->get_qos(writer_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
writer_qos.history.depth = 5;
// Set the new QoS
if (datawriter->set_qos(writer_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 6.20 Changing the QoS of an Existing DataWriter with a QoS Profile

```
retcode = writer->set_qos_with_profile(
    "WriterProfileLibrary", "WriterProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

6.3.15.4 Using a Topic's QoS to Initialize a DataWriter's QoS

Several *DataWriter* QoS Policies can also be found in the QoS Policies for *Topics* (see [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#)). The QoS Policies set in the Topic do not directly affect the *DataWriters* (or *DataReaders*) that use that *Topic*. In many ways, some QoS Policies are a *Topic*-level concept, even though the DDS standard allows you to set different values for those policies for different

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

DataWriters and *DataReaders* of the same *Topic*. Thus, the policies in the **DDS_TopicQos** structure exist as a way to help centralize and annotate the intended or suggested values of those QoS Policies. Connex DDS does not check to see if the actual policies set for a *DataWriter* is aligned with those set in the *Topic* to which it is bound.

There are many ways to use the QoS Policies' values set in the *Topic* when setting the QoS Policies' values in a *DataWriter*. The most straightforward way is to get the values of policies directly from the *Topic* and use them in the policies for the *DataWriter*, as shown in [Figure 6.21 Copying Selected QoS from a Topic when Creating a DataWriter below](#).

Figure 6.21 Copying Selected QoS from a Topic when Creating a DataWriter

```
DDS_DataWriterQos writer_qos;1
DDS_TopicQos topic_qos;
// topic and publisher already created
// get current QoS for the topic, default QoS for the writer
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
if (publisher->get_default_datawriter_qos(writer_qos)
    != DDS_RETCODE_OK) {
    // handle error
}
// Copy specific policies from topic QoS to writer QoS
writer_qos.deadline = topic_qos.deadline;
writer_qos.reliability = topic_qos.reliability;
// Create the DataWriter with the modified QoS
DDSDataWriter* writer = publisher->create_datawriter(topic,
    writer_qos, NULL, DDS_STATUS_MASK_NONE);
```

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

You can use the *Publisher's* `copy_from_topic_qos()` operation to copy all of the common policies from the *Topic* QoS to a *DataWriter* QoS. This is illustrated in [Figure 6.22 Copying all QoS from a Topic when Creating a DataWriter below](#).

Figure 6.22 Copying all QoS from a Topic when Creating a DataWriter

```

DDS_DataWriterQos writer_qos;1
DDS_TopicQos topic_qos;
// topic, publisher, writer_listener already created
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
// handle error
}
if (publisher->get_default_datawriter_qos(writer_qos)
    != DDS_RETCODE_OK)
{
    // handle error
}
// copy relevant QoS from topic into writer's qos
publisher->copy_from_topic_qos(writer_qos, topic_qos);
// Optionally, modify policies as desired
writer_qos.deadline.duration.sec = 1;
writer_qos.deadline.duration.nanosec = 0;
// Create the DataWriter with the modified QoS
DDSDataWriter* writer = publisher->create_datawriter(topic,
    writer_qos, writer_listener, DDS_STATUS_MASK_ALL);

```

In another design pattern, you may want to start with the default QoS values for a *DataWriter* and override them with the QoS values of the *Topic*. [Figure 6.23 Combining Default Topic and DataWriter QoS \(Option 1\) on the next page](#) gives an example of how to do this.

Because this is a common pattern, Connex DDS provides a special macro, **DDS_DATAWRITER_QOS_USE_TOPIC_QOS**, that can be used to indicate that the *DataWriter* should be created with the set of QoS values that results from modifying the default *DataWriter* QoS Policies with the QoS values specified by the *Topic*. [Figure 6.24 Combining Default Topic and DataWriter QoS \(Option 2\) on the next page](#) shows how the macro is used.

The code fragments shown in [Figure 6.23 Combining Default Topic and DataWriter QoS \(Option 1\) on the next page](#) and [Figure 6.24 Combining Default Topic and DataWriter QoS \(Option 2\) on the next page](#) result in identical QoS settings for the created *DataWriter*.

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

Figure 6.23 Combining Default Topic and DataWriter QoS (Option 1)

```

DDS_DataWriterQos writer_qos;1
DDS_TopicQos topic_qos;
// topic, publisher, writer_listener already created
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
// handle error
}
if (publisher->get_default_datawriter_qos(writer_qos)
    != DDS_RETCODE_OK) {
// handle error
}
if (publisher->copy_from_topic_qos(writer_qos, topic_qos)
    != DDS_RETCODE_OK) {
// handle error
}
// Create the DataWriter with the combined QoS
DDSDataWriter* writer =
    publisher->create_datawriter(topic, writer_qos,
    writer_listener, DDS_STATUS_MASK_ALL);

```

Figure 6.24 Combining Default Topic and DataWriter QoS (Option 2)

```

// topic, publisher, writer_listener already created
DDSDataWriter* writer = publisher->create_datawriter (topic,
DDS_DATAWRITER_QOS_USE_TOPIC_QOS,
writer_listener, DDS_STATUS_MASK_ALL);

```

For more information on the general use and manipulation of QoS Policies, see [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#).

6.3.16 Navigating Relationships Among DDS Entities

6.3.16.1 Finding Matching Subscriptions

The following *DataWriter* operations can be used to get information on the *DataReaders* that are currently associated with the *DataWriter* (that is, the *DataReaders* to which Connex DDS will send the data written by the *DataWriter*).

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

- `get_matched_subscriptions()`
- `get_matched_subscription_data()`
- `get_matched_subscription_locators()`

`get_matched_subscriptions()` will return a sequence of handles to matched *DataReaders*. You can use these handles in the `get_matched_subscription_data()` method to get information about the *DataReader* such as the values of its QoS Policies.

`get_matched_subscription_locators()` retrieves a list of locators for subscriptions currently "associated" with the *DataWriter*. Matched subscription locators include locators for all those subscriptions in the same DDS domain that have a matching Topic, compatible QoS, and a common partition that the *DomainParticipant* has not indicated should be "ignored." These are the locators that Connex DDS uses to communicate with matching *DataReaders*. (See [Locator Format \(Section 14.2.1.1 on page 714\)](#).)

Note: In the Modern C++ API these operations are freestanding functions in the `dds::pub` or `rti::pub` namespaces.

You can also get the `DATA_WRITER_PROTOCOL_STATUS` for matching subscriptions with these operations (see [DATA_WRITER_PROTOCOL_STATUS \(Section 6.3.6.3 on page 273\)](#)):

- `get_matched_subscription_datawriter_protocol_status()`
- `get_matched_subscription_datawriter_protocol_status_by_locator()`

Notes:

- Status/data for a matched subscription is only kept while the matched subscription is alive. Once a matched subscription is no longer alive, its status is deleted. If you try to get the status/data for a matched subscription that is no longer alive, the 'get status' or 'get data' call will return an error.
- *DataReaders* that have been ignored using the *DomainParticipant's* `ignore_subscription()` operation are not considered to be matched even if the *DataReader* has the same *Topic* and compatible QoS Policies. Thus, they will not be included in the list of *DataReaders* returned by `get_matched_subscriptions()` or `get_matched_subscription_locators()`. See [Ignoring Publications and Subscriptions \(Section 16.4.2 on page 786\)](#) for more on `ignore_subscription()`.
- The `get_matched_subscription_data()` operation does not retrieve the following information from built-in-topic data structures: `type_code`, `property`, and `content_filter_property`. This information is available through the `on_data_available()` callback (if a *DataReaderListener* is installed on the *SubscriptionBuiltinTopicDataReader*). (bug 11914)

See also: [Finding the Matching Subscription's ParticipantBuiltinTopicData \(Section 6.3.16.2 on the next page\)](#)

6.3.16.2 Finding the Matching Subscription's ParticipantBuiltinTopicData

get_matched_subscription_participant_data() allows you to get the `DDS_ParticipantBuiltinTopicData` (see [Table 16.1 Participant Built-in Topic's Data Type \(DDS_ParticipantBuiltinTopicData\)](#)) of a matched subscription using a subscription handle.

This operation retrieves the information on a discovered *DomainParticipant* associated with the subscription that is currently matching with the *DataWriter*. The subscription handle passed into this operation must correspond to a subscription currently associated with the *DataWriter*. Otherwise, the operation will fail with `RETCODE_BAD_PARAMETER`. The operation may also fail with `RETCODE_PRECONDITION_NOT_MET` if the subscription corresponds to the same *DomainParticipant* to which the *DataWriter* belongs.

Use **get_matched_subscriptions()** (see [Finding Matching Subscriptions \(Section 6.3.16.1 on page 309\)](#)) to find the subscriptions that are currently matched with the *DataWriter*.

6.3.16.3 Finding Related DDS Entities

These operations are useful for obtaining a handle to various related *Entities*:

- **get_publisher()**
- **get_topic()**

get_publisher() returns the *Publisher* that created the *DataWriter*. **get_topic()** returns the *Topic* with which the *DataWriter* is associated.

6.3.17 Asserting Liveliness

The **assert_liveliness()** operation can be used to manually assert the liveliness of the *DataWriter* without writing data. This operation is only useful if the kind of `LIVELINESS QosPolicy` ([Section 6.5.13 on page 382](#)) is `MANUAL_BY_PARTICIPANT` or `MANUAL_BY_TOPIC`.

How *DataReaders* determine if *DataWriters* are alive is configured using the `LIVELINESS QosPolicy` ([Section 6.5.13 on page 382](#)). The **lease_duration** parameter of the `LIVELINESS QosPolicy` is a contract by the *DataWriter* to all of its matched *DataReaders* that it will send a packet within the time value of the **lease_duration** to state that it is still alive.

There are three ways to assert liveliness. One is to have Connex DDS itself send liveliness packets periodically when the kind of `LIVELINESS QosPolicy` is set to `AUTOMATIC`. The other two ways to assert liveliness, used when liveliness is set to `MANUAL`, are to call **write()** to send data or to call the **assert_liveliness()** operation without sending data.

6.3.18 Turbo Mode and Automatic Throttling for DataWriter Performance—Experimental Features

This section describes two experimental features. The *DataWriter* has many QoS settings that can affect the latency and throughput of outgoing data. There are QoS settings to control send window size (see [Understanding the Send Queue and Setting its Size \(Section 10.3.2.1 on page 639\)](#)) and settings that allow to aggregate multiple DDS samples together to reduce CPU and bandwidth utilization (see [BATCH QoS Policy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#) and [FlowControllers \(DDS Extension\) \(Section 6.6 on page 422\)](#)). The choice of settings that provide the best performance depends on several factors, such as the frequency of writing data, the size of the data, or the condition of the network. If these factors do not change over time, you can choose values for those QoS settings that best suit your system. If these factors do change over time in your system, you can use the following properties to let *Connex DDS* automatically adjust the QoS settings as system conditions change:

- **dds.domain_participant.auto_throttle.enable**: Configures the *DomainParticipant* to gather internal measurements (during *DomainParticipant* creation) that are required for the Auto Throttle feature. This allows *DataWriters* belonging to this *DomainParticipant* to use the Auto Throttle feature. Default: false.
- **dds.data_writer.auto_throttle.enable**: Enables automatic throttling in the *DataWriter* so it can automatically adjust the writing rate and the send window size; this minimizes the need for repair DDS samples and improves latency. Default: false. For additional information on automatic throttling, see [Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature \(Section 6.5.2.4 on page 344\)](#).

Note: This property takes effect only in *DataWriters* that belong to a *DomainParticipant* that has set the property **dds.domain_participant.auto_throttle.enable** (described above) to true.

- **dds.data_writer.enable_turbo_mode**: Enables Turbo Mode and adjusts the batch `max_data_bytes` ([Section on page 342](#)) (see [BATCH QoS Policy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#)) based on how frequently the *DataWriter* writes data. Default: false. For additional information, see [Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature \(Section 6.5.2.4 on page 344\)](#).

The Built-in QoS profile **BuiltinQoSLibExp::Generic.AutoTuning** enables both Turbo Mode and Auto Throttling.

6.4 Publisher/Subscriber QoS Policies

This section provides detailed information on the QoS Policies associated with a *Publisher*. Note that *Subscribers* have the exact same set of policies. [Table 6.2 Publisher QoS Policies](#) provides a quick reference. They are presented here in alphabetical order.

- [ASYNCHRONOUS_PUBLISHER QosPolicy \(DDS Extension\) \(Section 6.4.1 below\)](#)
- [ENTITYFACTORY QosPolicy \(Section 6.4.2 on page 315\)](#)
- [EXCLUSIVE_AREA QosPolicy \(DDS Extension\) \(Section 6.4.3 on page 318\)](#)
- [GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)
- [PARTITION QosPolicy \(Section 6.4.5 on page 323\)](#)
- [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#)

6.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)

This QosPolicy is used to enable or disable asynchronous publishing and asynchronous batch flushing for the *Publisher*.

This QosPolicy can be used to reduce amount of time spent in the user thread to send data. You can use it to send *large* data reliably. *Large* in this context means that the data cannot be sent as a single packet by a transport. For example, to send data larger than 63K reliably using UDP/IP, you must configure Connex DDS to send the data using asynchronous *Publishers*.

If so configured, the *Publisher* will spawn two threads, one for asynchronous publishing and one for asynchronous batch flushing. The asynchronous publisher thread will be shared by all *DataWriters* (belonging to this *Publisher*) that have their [PUBLISH_MODE QosPolicy \(DDS Extension\) \(Section 6.5.18 on page 397\)](#) **kind** set to ASYNCHRONOUS. The asynchronous publishing thread will then handle the data transmission chores for those *DataWriters*. This thread will only be spawned when the first of these *DataWriters* is enabled.

The asynchronous batch flushing thread will be shared by all *DataWriters* (belonging to this *Publisher*) that have batching enabled and **max_flush_delay** different than DURATION_INFINITE in [BATCH QosPolicy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#). This thread will only be spawned when the first of these *DataWriters* is enabled.

This QosPolicy allows you to adjust the asynchronous publishing and asynchronous batch flushing threads independently.

Batching and asynchronous publication are independent of one another. Flushing a batch on an asynchronous *DataWriter* makes it available for sending to the *DataWriter's* [FlowControllers \(DDS Extension\) \(Section 6.6 on page 422\)](#). From the point of view of the FlowController, a batch is treated like one large DDS sample.

Connex DDS will sometimes coalesce multiple DDS samples into a single network datagram. For example, DDS samples buffered by a FlowController or sent in response to a negative acknowledgement (NACK) may be coalesced. This behavior is distinct from DDS sample batching. DDS data samples sent by different asynchronous *DataWriters* belonging to the same *Publisher* to the same destination will not be coalesced into a single network packet. Instead, two separate network packets will be sent. Only DDS samples written by the *same DataWriter* and intended for the *same destination* will be coalesced.

This QosPolicy includes the members in [Table 6.18 DDS_AsynchronousPublisherQosPolicy](#).

Table 6.18 DDS_AsynchronousPublisherQosPolicy

Type	Field Name	Description
DDS_Boolean	disable_asynchronous_write	Disables asynchronous publishing. To write asynchronously, this field must be FALSE (the default).
DDS_ThreadSettings_t	thread	Settings for the publishing thread. These settings are OS-dependent (see the RTI Connex DDS Core Libraries Platform Notes).
DDS_Boolean	disable_asynchronous_batch	Disables asynchronous batch flushing. To flush asynchronously, this field must be FALSE (the default).
DDS_ThreadSettings_t	asynchronous_batch_thread	Settings for the asynchronous batch flushing thread. These settings are OS-dependent (see the RTI Connex DDS Core Libraries Platform Notes).

6.4.1.1 Properties

This QosPolicy cannot be modified after the *Publisher* is created.

Since it is only for *Publishers*, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

6.4.1.2 Related QosPolicies

- If *disable_asynchronous_write* is TRUE (not the default), then any *DataWriters* created from this *Publisher* must have their [PUBLISH_MODE QosPolicy \(DDS Extension\)](#) (Section 6.5.18 on page 397) *kind* set to SYNCHRONOUS. (Otherwise *create_datawriter()* will return INCONSISTENT_QOS.)
- If *disable_asynchronous_batch* is TRUE (not the default), then any *DataWriters* created from this *Publisher* must have *max_flush_delay* in [BATCH QosPolicy \(DDS Extension\)](#) (Section 6.5.2 on page 341) set to DURATION_INFINITE. (Otherwise *create_datawriter()* will return INCONSISTENT_QOS.)
- *DataWriters* configured to use the [MULTI_CHANNEL QosPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386) do not support asynchronous publishing; an error is returned if a multi-channel *DataWriter* is configured for asynchronous publishing.

6.4.1.3 Applicable DDS Entities

- [Publishers](#) (Section 6.2 on page 243)

6.4.1.4 System Resource Considerations

Two threads can potentially be created:

- For asynchronous publishing, system resource usage depends on the activity of the asynchronous thread controlled by the FlowController (see [FlowControllers \(DDS Extension\)](#) (Section 6.6 on page 422)).
- For asynchronous batch flushing, system resource usage depends on the activity of the asynchronous thread controlled by *max_flush_delay* in [BATCH QosPolicy \(DDS Extension\)](#) (Section 6.5.2 on page 341).

6.4.2 ENTITYFACTORY QosPolicy

This QosPolicy controls whether or not child *Entities* are created in the enabled state.

This QosPolicy applies to the *DomainParticipantFactory*, *DomainParticipants*, *Publishers*, and *Subscribers*, which act as ‘factories’ for the creation of subordinate *Entities*. A *DomainParticipantFactory* is used to create *DomainParticipants*. A *DomainParticipant* is used to create both *Publishers* and *Subscribers*. A *Publisher* is used to create *DataWriters*, similarly a *Subscriber* is used to create *DataReaders*.

Entities can be created either in an ‘enabled’ or ‘disabled’ state. An enabled entity can actively participate in communication. A disabled entity cannot be discovered or take part in communication until it is explicitly enabled. For example, Connex DDS will not send data if the *write()* operation is called on a disabled *DataWriter*, nor will Connex DDS deliver data to a disabled *DataReader*. You can only enable a disabled entity. Once an entity is enabled, you cannot disable it, see [Enabling DDS Entities](#) (Section 4.1.2 on page 154) about the *enable()* method.

The ENTITYFACTORY contains only one member, as illustrated in [Table 6.19 DDS_EntityFactoryQosPolicy](#).

Table 6.19 DDS_EntityFactoryQosPolicy

Type	Field Name	Description
DDS_Boolean	autoenable_created_entities	DDS_BOOLEAN_TRUE: enable <i>Entities</i> when they are created DDS_BOOLEAN_FALSE: do not enable <i>Entities</i> when they are created

The ENTITYFACTORY QosPolicy controls whether the *Entities* created from the factory are automatically enabled upon creation or are left disabled. For example, if a *Publisher* is configured to auto-enable created *Entities*, then all *DataWriters* created from that *Publisher* will be automatically enabled.

Note: if an entity is disabled, then all of the child *Entities* it creates are also created in a disabled state, regardless of the setting of this QosPolicy. However, enabling a disabled entity will enable all of its children if this QosPolicy is set to autoenable child *Entities*.

Note: an entity can only be enabled; it cannot be disabled after its been enabled.

See [Example \(Section 6.4.2.1 below\)](#) for an example of how to set this policy.

There are various reasons why you may want to create *Entities* in the disabled state:

- To get around a “chicken and egg”-type issue. Where you need to have an entity in order to modify it, but you don’t want the entity to be used by Connex DDS until it has been modified.

For example, if you create a *DomainParticipant* in the enabled state, it will immediately start sending packets to other nodes trying to discover if other Connex DDS applications exist. However, you may want to configure the built-in topic reader listener before discovery occurs. To do this, you need to create a *DomainParticipant* in the disabled state because once enabled, discovery will occur. If you set up the built-in topic reader listener after the *DomainParticipant* is enabled, you may miss some discovery traffic.

- You may want to create *Entities* without having them automatically start to work. This especially pertains to *DataReaders*. If you create a *DataReader* in an enabled state and you are using *DataReaderListeners*, Connex DDS will immediately search for matching *DataWriters* and call-back the listener as soon as data is published. This may not be what you want to happen if your application is still in the middle of initialization when data arrives.

So typically, you would create all *Entities* in a disabled state, and then when all parts of the application have been initialized, one would enable all *Entities* at the same time using the *enable()* operation on the *DomainParticipant*, see [Enabling DDS Entities \(Section 4.1.2 on page 154\)](#).

- An entity’s existence is not advertised to other participants in the network until the entity is enabled. Instead of sending an individual declaration packet to other applications announcing the existence of the entity, Connex DDS can be more efficient in bundling multiple declarations into a single packet when you enable all *Entities* at the same time.

See [Enabling DDS Entities \(Section 4.1.2 on page 154\)](#) for more information about enabled/disabled *Entities*.

6.4.2.1 Example

The code in [Figure 6.25 Configuring a Publisher so that New DataWriters are Disabled on the next page](#) illustrates how to use the ENTITYFACTORY QoS.

Figure 6.25 Configuring a Publisher so that New DataWriters are Disabled

```

DDS_PublisherQos publisher_qos;1
// topic, publisher, writer_listener already created
if (publisher->get_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
publisher_qos.entity_factory.autoenable_created_entities
    = DDS_BOOLEAN_FALSE;
if (publisher->set_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Subsequently created DataWriters are created disabled and
// must be explicitly enabled by the user-code
DDSDataWriter* writer = publisher->create_datawriter(topic,
    DDS_DATAWRITER_QOS_DEFAULT, writer_listener, DDS_STATUS_MASK_ALL);
// now do other initialization
// Now explicitly enable the DataWriter, this will allow other
// applications to discover the DataWriter and for this application
// to send data when the DataWriter's write() method is called
writer->enable();

```

6.4.2.2 Properties

This QosPolicy can be modified at any time.

It can be set differently on the publishing and subscribing sides.

6.4.2.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

6.4.2.4 Applicable DDS Entities

- [DomainParticipantFactory](#) (Section 8.2 on page 539)
- [DomainParticipants](#) (Section 8.3 on page 547)
- [Publishers](#) (Section 6.2 on page 243)
- [Subscribers](#) (Section 7.2 on page 440)

6.4.2.5 System Resource Considerations

This QosPolicy does not significantly impact the use of system resources.

¹Note in C, you must initialize the QoS structures before they are used, see [Special QosPolicy Handling Considerations for C](#) (Section 4.2.2 on page 168).

6.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension)

This QosPolicy controls the creation and use of Exclusive Areas. An exclusive area (EA) is a mutex with built-in deadlock protection when multiple EAs are in use. It is used to provide mutual exclusion among different threads of execution. Multiple EAs allow greater concurrency among the internal and user threads when executing Connex DDS code.

EAs allow Connex DDS to be multi-threaded while preventing threads from a classical deadlock scenario for multi-threaded applications. EAs prevent a *DomainParticipant's* internal threads from deadlocking with each other when executing internal code as well as when executing the code of user-registered listener callbacks.

Within an EA, all calls to the code protected by the EA are single threaded. Each *DomainParticipant*, *Publisher* and *Subscriber* represents a separate EA. All *DataWriters* of the same *Publisher* and all *DataReaders* of the same *Subscriber* share the EA of its parent. This means that the *DataWriters* of the same *Publisher* and the *DataReaders* of the same *Subscriber* are inherently single threaded.

Within an EA, there are limitations on how code protected by a different EA can be accessed. For example, when data is being processed by user code received in the *DataReaderListener* of a *Subscriber* EA, the user code may call the *write()* function of a *DataWriter* that is protected by the EA of its *Publisher*. So you can send data in the function called to process received data. However, you cannot create *Entities* or call functions that are protected by the EA of the *DomainParticipant*. See [Exclusive Areas \(EAs\) \(Section 4.5 on page 182\)](#) for the complete documentation on Exclusive Areas.

With this QoS, you can force a *Publisher* or *Subscriber* to share the same EA as its *DomainParticipant*. Using this capability, the restriction of not being to create *Entities* in a *DataReaderListener's on_data_available()* callback is lifted. However, the trade-off is that the application has reduced concurrency through the *Entities* that share an EA.

Note that the restrictions on calling methods in a different EA only exists for user code that is called in registered Listeners by internal *DomainParticipant* threads. User code may call all Connex DDS functions for any *Entities* from their own threads at any time.

The EXCLUSIVE_AREA includes a single member, as listed in [Table 6.20 DDS_ExclusiveAreaQosPolicy](#). For the default value, please see the API Reference HTML documentation.

Table 6.20 DDS_ExclusiveAreaQosPolicy

Type	Field Name	Description
DDS_Boolean	use_shared_exclusive_area	DDS_BOOLEAN_FALSE: subordinates will not use the same EA DDS_BOOLEAN_TRUE: subordinates will use the same EA

The implications and restrictions of using a private or shared EA are discussed in [Exclusive Areas \(EAs\) \(Section 4.5 on page 182\)](#). The basic trade-off is concurrency versus restrictions on which methods can be called in user, listener, callback functions. To summarize:

Behavior when the *Publisher* or *Subscriber*'s `use_shared_exclusive_area` is set to *FALSE*:

- The creation of the *Publisher/Subscriber* will create an EA that will be used only by the *Publisher/Subscriber* and the *DataWriters/DataReaders* that belong to them.
- Consequences: This setting maximizes concurrency at the expense of creating a mutex for the *Publisher* or *Subscriber*. In addition, using a separate EA may restrict certain Connex DDS operations (see [Operations Allowed within Listener Callbacks \(Section 4.4.5 on page 182\)](#)) from being called from the callbacks of Listeners attached to those *Entities* and the *Entities* that they create. This limitation results from a built-in deadlock protection mechanism.

Behavior when the *Publisher* or *Subscriber*'s `use_shared_exclusive_area` is set to *TRUE*:

- The creation of the *Publisher/Subscriber* does not create a new EA. Instead, the *Publisher/Subscriber*, along with the *DataWriters/DataReaders* that they create, will use a common EA shared with the *DomainParticipant*.
- Consequences: By sharing the same EA among multiple *Entities*, you may decrease the amount of concurrency in the application, which can adversely impact performance. However, this setting does use less resources and allows you to call almost any operation on any Entity within a listener callback (see [Exclusive Areas \(EAs\) \(Section 4.5 on page 182\)](#) for full details).

6.4.3.1 Example

The code in [Figure 6.26 Creating a Publisher with a Shared Exclusive Area on the facing page](#) illustrates how to change the `EXCLUSIVE_AREA` policy.

Figure 6.26 Creating a Publisher with a Shared Exclusive Area

```

DDS_PublisherQos publisher_qos;1
// domain, publisher_listener have been previously created
if (participant->get_default_publisher_qos(publisher_qos) !=
    DDS_RETCODE_OK) {
    // handle error
}
publisher_qos.exclusive_area.use_shared_exclusive_area = DDS_BOOLEAN_TRUE;
DDSPublisher* publisher = participant->create_publisher(publisher_qos,
    publisher_listener, DDS_STATUS_MASK_ALL);

```

6.4.3.2 Properties

This QoSPolicy cannot be modified after the Entity has been created.

It can be set differently on the publishing and subscribing sides.

6.4.3.3 Related QoS Policies

This QoSPolicy does not interact with any other policies.

6.4.3.4 Applicable DDS Entities

- [Publishers \(Section 6.2 on page 243\)](#)
- [Subscribers \(Section 7.2 on page 440\)](#)

6.4.3.5 System Resource Considerations

This QoSPolicy affects the use of operating-system mutexes. When `use_shared_exclusive_area` is `FALSE`, the creation of a *Publisher* or *Subscriber* will create an operating-system mutex.

6.4.4 GROUP_DATA QoS Policy

This QoSPolicy provides an area where your application can store additional information related to the *Publisher* and *Subscriber*. This information is passed between applications during discovery (see [Discovery \(Section Chapter 14 on page 709\)](#)) using built-in-topics (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)). How this information is used will be up to user code. Connex DDS does not do anything with the information stored as `GROUP_DATA` except to pass it to other applications.

¹Note in C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

Use cases are often application-to-application identification, authentication, authorization, and encryption purposes. For example, applications can use this QosPolicy to send security certificates to each other for RSA-type security.

The value of the GROUP_DATA QosPolicy is sent to remote applications when they are first discovered, as well as when the *Publisher* or *Subscriber*'s *set_qos()* method is called after changing the value of the GROUP_DATA. User code can set listeners on the built-in *DataReaders* of the built-in *Topics* used by Connex DDS to propagate discovery information. Methods in the built-in topic listeners will be called whenever new *DomainParticipants*, *DataReaders*, and *DataWriters* are found. Within the user callback, you will have access to the GROUP_DATA that was set for the associated *Publisher* or *Subscriber*.

Currently, GROUP_DATA of the associated *Publisher* or *Subscriber* is only propagated with the information that declares a *DataWriter* or *DataReader*. Thus, you will need to access the value of GROUP_DATA through DDS_PublicationBuiltinTopicData or DDS_SubscriptionBuiltinTopicData (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)).

The structure for the GROUP_DATA QosPolicy includes just one field, as seen in [Table 6.21 DDS_GroupDataQosPolicy](#). The field is a sequence of octets that translates to a contiguous buffer of bytes whose contents and length is set by the user. The maximum size for the data are set in the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#).

Table 6.21 DDS_GroupDataQosPolicy

Type	Field Name	Description
DDS_OctetSeq	value	Empty by default

This policy is similar to the [USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#) and [TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#) that apply to other types of *Entities*.

6.4.4.1 Example

One possible use of GROUP_DATA is to pass some credential or certificate that your subscriber application can use to accept or reject communication with the *DataWriters* that belong to the *Publisher* (or vice versa, where the publisher application can validate the permission of *DataReaders* of a *Subscriber* to receive its data). The value of the GROUP_DATA of the *Publisher* is propagated in the 'group_data' field of the DDS_PublicationBuiltinTopicData that is sent with the declaration of each *DataWriter*. Similarly, the value of the GROUP_DATA of the *Subscriber* is propagated in the 'group_data' field of the DDS_SubscriptionBuiltinTopicData that is sent with the declaration of each *DataReader*.

When Connex DDS discovers a *DataWriter/DataReader*, the application can be notified of the discovery of the new entity and retrieve information about the *DataWriter/DataReader* QoS by reading the DCPSPublication or DCPSSubscription built-in topics (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)). Your application can then examine the GROUP_DATA field in the built-in Topic and decide

whether or not the *DataWriter/DataReader* should be allowed to communicate with local *DataReaders/DataWriters*. If communication is not allowed, the application can use the *DomainParticipant's ignore_publication()* or *ignore_subscription()* operation to reject the newly discovered remote entity as one with which the application allows Connex DDS to communicate. See [Figure 16.2, “Ignoring Publications,”](#) on page 16-12 for an example of how to do this.

The code in [Figure 6.27 Creating a Publisher with GROUP_DATA](#) below illustrates how to change the `GROUP_DATA` policy.

Figure 6.27 Creating a Publisher with GROUP_DATA

```

DDS_PublisherQos publisher_qos;1
int i = 0;
// Bytes that will be used for the group data. In this case, 8 bytes
// of some information that is meaningful to the user application
char myGroupData[GROUP_DATA_SIZE] =
    { 0x34, 0xaa, 0xfe, 0x31, 0x7a, 0xf2, 0x34, 0xaa};
// assume domainparticipant and publisher_listener already created
if (participant->get_default_publisher_qos(publisher_qos) !=
    DDS_RETCODE_OK) {
    // handle error
}
// Must set the size of the sequence first
publisher_qos.group_data.value.maximum(GROUP_DATA_SIZE);
publisher_qos.group_data.value.length(GROUP_DATA_SIZE);
for (i = 0; i < GROUP_DATA_SIZE; i++) {
    publisher_qos.group_data.value[i] = myGroupData[i]
}
DDSPublisher* publisher = participant->create_publisher( publisher_qos,
    publisher_listener, DDS_STATUS_MASK_ALL);

```

6.4.4.2 Properties

This QoSPolicy can be modified at any time.

It can be set differently on the publishing and subscribing sides.

6.4.4.3 Related QoS Policies

- [TOPIC_DATA QoSPolicy](#) (Section 5.2.1 on page 209)
- [USER_DATA QoSPolicy](#) (Section 6.5.26 on page 417)

¹Note in C, you must initialize the QoS structures before they are used, see [Special QoSPolicy Handling Considerations for C](#) (Section 4.2.2 on page 168).

- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#)

6.4.4.4 Applicable DDS Entities

- [Publishers \(Section 6.2 on page 243\)](#)
- [Subscribers \(Section 7.2 on page 440\)](#)

6.4.4.5 System Resource Considerations

The maximum size of the `GROUP_DATA` is set in the `publisher_group_data_max_length` and `subscriber_group_data_max_length` fields of the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#). Because Connex DDS will allocate memory based on this value, you should only increase this value if you need to. If your system does not use `GROUP_DATA`, then you can set this value to zero to save memory. Setting the value of the `GROUP_DATA` QosPolicy to hold data longer than the value set in the `[publisher/subscriber]_group_data_max_length` fields will result in failure and an `INCONSISTENT_QOS_POLICY` return code.

However, should you decide to change the maximum size of `GROUP_DATA`, you *must* make certain that all applications in the DDS domain have changed the value of `[publisher/subscriber]_group_data_max_length` to be the same. If two applications have different limits on the size of `GROUP_DATA`, and one application sets the `GROUP_DATA` QosPolicy to hold data that is greater than the maximum size set by another application, then the matching *DataWriters* and *DataReaders* of the *Publisher* and *Subscriber* between the two applications will *not* connect. This is also true for the `TOPIC_DATA` ([TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#)) and `USER_DATA` ([USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)) QosPolicies.

6.4.5 PARTITION QoS

The `PARTITION` QoS provides another way to control which *DataWriters* will match—and thus communicate with—which *DataReaders*. It can be used to prevent *DataWriters* and *DataReaders* that would have otherwise matched with the same *Topic* and compatible QoS policies from talking to each other. Much in the same way that only applications within the same DDS domain will communicate with each other, only *DataWriters* and *DataReaders* that belong to the same partition can talk to each other.

The `PARTITION` QoS applies to *Publishers* and *Subscribers*, therefore the *DataWriters* and *DataReaders* belong to the partitions as set on the *Publishers* and *Subscribers* that created them. The mechanism implementing the `PARTITION` QoS is relatively lightweight, and membership in a partition can be dynamically changed. Unlike the creation and destruction of *DomainParticipants*, there is no spawning and killing of threads or allocation and deallocation of memory when *Publishers* and *Subscribers* add or remove themselves from partitions.

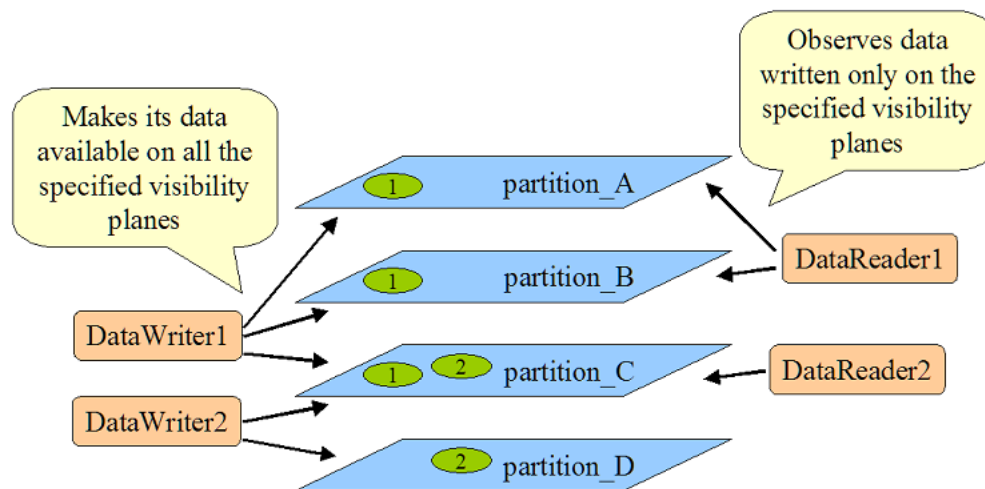
The `PARTITION` QoS consists of a set of partition names that identify the partitions of which the Entity is a member. These names are simply strings, and *DataWriters* and *DataReaders* are considered to be in the

same partition if they have at least one partition name in common in the PARTITION QoS set on their *Publishers* or *Subscribers*. By default, *Publishers* and *Subscribers* belong to a single partition whose name is the empty string, “”.

Conceptually each partition name can be thought of as defining a “visibility plane” within the DDS domain. *DataWriters* will make their data available on all the visibility planes that correspond to its *Publisher’s* partition names, and the *DataReaders* will see the data that is placed on any of the visibility planes that correspond to its *Subscriber’s* partition names.

Figure 6.28 Controlling Visibility of Data with the PARTITION QoS below illustrates the concept of PARTITION QoS. In this figure, all *DataWriters* and *DataReaders* belong to the same DDS domain and refer to the same *Topic*. *DataWriter1* is configured to belong to three partitions: *partition_A*, *partition_B*, and *partition_C*. *DataWriter2* belongs to *partition_C* and *partition_D*.

Figure 6.28 Controlling Visibility of Data with the PARTITION QoS



Similarly, *DataReader1* is configured to belong to *partition_A* and *partition_B*, and *DataReader2* belongs only to *partition_C*. Given this topology, the data written by *DataWriter1* is visible in partitions A, B, and C. The oval tagged with the number “1” represents one DDS data sample written by *DataWriter1*.

Similarly, the data written by *DataWriter2* is visible in partitions C and D. The oval tagged with the number “2” represents one DDS data sample written by *DataWriter2*.

The result is that the data written by *DataWriter1* will be received by both *DataReader1* and *DataReader2*, but the data written by *DataWriter2* will only be visible by *DataReader2*.

Publishers and *Subscribers* always belong to a partition. By default, *Publishers* and *Subscribers* belong to a single partition whose name is the empty string, “”. If you set the PARTITION QoS to be an empty set, Connex DDS will assign the *Publisher* or *Subscriber* to the default partition, “”. Thus, for the example above, without using the PARTITION QoS, *DataReaders* 1 and 2 would have received all DDS data samples written by *DataWriters* 1 and 2.

6.4.5.1 Rules for PARTITION Matching

On the *Publisher* side, the PARTITION QoSPolicy associates a set of strings (partition names) with the *Publisher*. On the *Subscriber* side, the application also uses the PARTITION QoS to associate partition names with the *Subscriber*.

Taking into account the PARTITION QoS, a *DataWriter* will communicate with a *DataReader* if and only if the following conditions apply:

1. The *DataWriter* and *DataReader* belong to the same DDS domain. That is, their respective *DomainParticipants* are bound to the same DDS domain ID (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#)).
2. The *DataWriter* and *DataReader* have matching *Topics*. That is, each is associated with a *Topic* with the same `topic_name` and data type.
3. The QoS offered by the *DataWriter* is compatible with the QoS requested by the *DataReader*.
4. The application has not used the `ignore_participant()`, `ignore_datareader()`, or `ignore_datawriter()` APIs to prevent the association (see [Restricting Communication—Ignoring Entities \(Section 16.4 on page 784\)](#)).
5. The *Publisher* to which the *DataWriter* belongs and the *Subscriber* to which the *DataReader* belongs must have at least one matching partition name.

The last condition reflects the visibility of the data introduced by the PARTITION QoS. Matching partition names is done by string comparison, thus partition names are case sensitive.

Note: Failure to match partitions is not considered an incompatible QoS and does not trigger any listeners or change any status conditions.

6.4.5.2 Pattern Matching for PARTITION Names

You may also add strings that are regular expressions¹ to the PARTITION QoSPolicy. A regular expression does not define a set of partitions to which the *Publisher* or *Subscriber* belongs, as much as it is used in the partition matching process to see if a remote entity has a partition name that would be matched with the regular expression. That is, the regular expressions in the PARTITION QoS of a *Publisher* are never matched against those found in the PARTITION QoS of a *Subscriber*. Regular expressions are always matched against “concrete” partition names. Thus, a concrete partition name may not contain any reserved characters that are used to define regular expressions, for example ‘*’, ‘.’, ‘+’, etc.

For more on regular expressions, see [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 on page 228\)](#).

¹As defined by the POSIX `fnmatch` API (1003.2-1992 section B.6).

If a PARTITION QoS only contains regular expressions, then the *Publisher* or *Subscriber* will be assigned automatically to the default partition with the empty string name (“”). Thus, do not be fooled into thinking that a PARTITION QoS that only contains the string “*” matches another PARTITION QoS that only contains the string “*”. Yes, the *Publisher* will match the *Subscriber*, but it is because they both belong to the default “” partition.

DataWriters and *DataReaders* are considered to have a partition in common if the sets of partitions that their associated *Publishers* and *Subscribers* have defined have:

At least one concrete partition name in common

A regular expression in one Entity that matches a concrete partition name in another Entity

The programmatic representation of the PARTITION QoS is shown in [Table 6.22 DDS_PartitionQoSPolicy](#). The QoSPolicy contains the single string sequence, name. Each element in the sequence can be a concrete name or a regular expression. The *Entity* will be assigned to the default “” partition if the sequence is empty.

Table 6.22 DDS_PartitionQoSPolicy

Type	Field Name	Description
DDS_StringSeq	name	Empty by default. There can be up to 64 names, with a maximum of 256 characters summed across all names.

You can have one long partition string of 256 chars, or multiple shorter strings that add up to 256 or less characters. For example, you can have one string of 4 chars and one string of 252 chars.

6.4.5.3 Example

Since the set of partitions for a *Publisher* or *Subscriber* can be dynamically changed, the Partition QoSPolicy is useful to control which *DataWriters* can send data to which *DataReaders* and vice versa—even if all of the *DataWriters* and *DataReaders* are for the same topic. This facility is useful for creating temporary separation groups among *Entities* that would otherwise be connected to and exchange data each other.

Note when using Partitions and Durability: If a Publisher changes partitions after startup, it is possible for a reliable, late-joining DataReader to receive data that was written for both the original and the new partition. For example, suppose a DataWriter with TRANSIENT_LOCAL Durability initially writes DDS samples with Partition A, but later changes to Partition B. In this case, a reliable, late-joining DataReader configured for Partition B will receive whatever DDS samples have been saved for the DataWriter. These may include DDS samples which were written when the DataWriter was using Partition A.

The code in [Figure 6.29 Setting Partition Names on a Publisher on the next page](#) illustrates how to change the PARTITION policy.

Figure 6.29 Setting Partition Names on a Publisher

```

DDS_PublisherQos publisher_qos;1
// domain, publisher_listener have been previously created
if (participant->get_default_publisher_qos(publisher_qos) !=
    DDS_RETCODE_OK) {
    // handle error
}
// Set the partition QoS
publisher_qos.partition.name.maximum(3);
publisher_qos.partition.name.length(3);
publisher_qos.partition.name[0] = DDS_String_dup("partition_A");
publisher_qos.partition.name[1] = DDS_String_dup("partition_B");
publisher_qos.partition.name[2] = DDS_String_dup("partition_C");
DDSPublisher* publisher = participant->create_publisher(
    publisher_qos, publisher_listener, DDS_STATUS_MASK_ALL);

```

The ability to dynamically control which *DataWriters* are matched to which *DataReaders* (of the same *Topic*) offered by the PARTITION QoS can be used in many different ways. Using partitions, connectivity can be controlled based on location-based partitioning, access-control groups, purpose, or a combination of these and other application-defined criteria. We will examine some of these options via concrete examples.

Example of location-based partitions. Assume you have a set of *Topics* in a traffic management system such as “TrafficAlert,” “AccidentReport,” and “CongestionStatus.” You may want to control the visibility of these *Topics* based on the actual location to which the information applies. You can do this by placing the *Publisher* in a partition that represents the area to which the information applies. This can be done using a string that includes the city, state, and country, such as “USA/California/Santa Clara.” A *Subscriber* can then choose whether it wants to see the alerts in a single city, the accidents in a set of states, or the congestion status across the US. Some concrete examples are shown in [Table 6.23 Example of Using Location-Based Partitions](#).

Table 6.23 Example of Using Location-Based Partitions

Publisher Partitions	Subscriber Partitions	Result
Specify a single partition name using the pattern: “<country>/<state>/<city>”	Specify multiple partition names, one per region of interest	Limits the visibility of the data to Subscribers that express interest in the geographical region.

¹Note in C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

Table 6.23 Example of Using Location-Based Partitions

Publisher Partitions	Subscriber Partitions	Result
“USA/California/Santa Clara”	(Subscriber participant is irrelevant here.)	Send only information for Santa Clara, California.
(Publisher partition is irrelevant here.)	“USA/California/Santa Clara”	Receive only information for Santa Clara, California.
	“USA/California/Santa Clara” “USA/California/Sunnyvale”	Receive information for Santa Clara or Sunnyvale, California.
	“USA/California/*” “USA/Nevada/*”	Receive information for California or Nevada.
	“USA/California/*” “USA/Nevada/Reno” “USA/Nevada/Las Vegas”	Receive information for California and two cities in Nevada.

Example of access-control group partitions. Suppose you have an application where access to the information must be restricted based on reader membership to access-control groups. You can map this group-controlled visibility to partitions by naming all the groups (e.g. executives, payroll, financial, general-staff, consultants, external-people) and assigning the *Publisher* to the set of partitions that represents which groups should have access to the information. The *Subscribers* specify the groups to which they belong, and the partition-matching behavior will ensure that the information is only distributed to *Subscribers* belonging to the appropriate groups. Some concrete examples are shown in [Table 6.24 Example of Access-Control Group Partitions](#).

Table 6.24 Example of Access-Control Group Partitions

Publisher Partitions	Subscriber Partitions	Result
Specify several partition names, one per group that is allowed access:	Specify multiple partition names, one per group to which the Subscriber belongs.	Limits the visibility of the data to Subscribers that belong to the access-groups specified by the Publisher.
“payroll” “financial”	(Subscriber participant is irrelevant here.)	Makes information available only to Subscribers that have access to either financial or payroll information.
(Publisher participant is irrelevant here.)	“executives” “financial”	Gain access to information that is intended for executives or people with access to the finances.

A slight variation of this pattern could be used to confine the information based on security levels.

Example of purpose-based partitions: Assume an application containing subsystems that can be used for multiple purposes, such as training, simulation, and real use. In some occasions it is convenient to be able

to dynamically switch the subsystem from operating in the “simulation world” to the “training world” or to the “real world.” For supervision purposes, it may be convenient to observe multiple worlds, so that you can compare the each one’s results. This can be accomplished by setting a partition name in the *Publisher* that represents the “world” to which it belongs and a set of partition names in the *Subscriber* that model the worlds that it can observe.

6.4.5.4 Properties

This QosPolicy can be modified at any time.

Strictly speaking, this QosPolicy does not have request-offered semantics, although it is matched between *DataWriters* and *DataReaders*, and communication is established only if there is a match between partition names.

6.4.5.5 Related QosPolicies

- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#).

6.4.5.6 Applicable DDS Entities

- [Publishers \(Section 6.2 on page 243\)](#)
- [Subscribers \(Section 7.2 on page 440\)](#)

6.4.5.7 System Resource Considerations

Partition names are propagated along with the declarations of the *DataReaders* and the *DataWriters* and can be examined by user code through built-in topics (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)). Thus the sum-total length of the partition names will impact the bandwidth needed to transmit those declarations, as well as the memory used to store them.

The maximum number of partitions and the maximum number of characters that can be used for the sum-total length of all partition names are configured using the `max_partitions` and `max_partition_cumulative_characters` fields of the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#). Setting more partitions or using longer names than allowed by those limits will result in failure and an `INCONSISTENT_QOS_POLICY` return code.

However, should you decide to change the maximum number of partitions or maximum cumulative length of partition names, then you *must* make certain that all applications in the DDS domain have changed the values of `max_partitions` and `max_partition_cumulative_characters` to be the same. If two applications have different values for those settings, and one application sets the `PARTITION` QosPolicy to hold more partitions or longer names than set by another application, then the matching *DataWriters* and *DataReaders* of the *Publisher* and *Subscriber* between the two applications will *not* connect. This similar to the restrictions for the `GROUP_DATA` ([GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)), `USER_`

DATA ([USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)), and TOPIC_DATA ([TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#)) QosPolicies.

6.4.6 PRESENTATION QosPolicy

Usually *DataReaders* will receive data in the order that it was sent by a *DataWriter*. In addition, data is presented to the *DataReader* as soon as the application receives the next value expected.

Sometimes, you may want a set of data for the same *DataWriter* to be presented to the receiving *DataReader* only after ALL the elements of the set have been received, but not before. You may also want the data to be presented in a different order than it was received. Specifically, for keyed data, you may want Connex DDS to present the data in keyed or instance order.

The Presentation QosPolicy allows you to specify different scopes of presentation: within a *DataWriter*, across instances of a *DataWriter*, and even across different *DataWriters* of a publisher. It also controls whether or not a set of changes within the scope must be delivered at the same time or delivered as soon as each element is received.

There are three components to this QoS, the boolean flag `coherent_access`, the boolean flag `ordered_access`, and an enumerated setting for the `access_scope`. The structure used is shown in [Table 6.25 DDS_PresentationQosPolicy](#).

Table 6.25 DDS_PresentationQosPolicy

Type	Field Name	Description
DDS_Presentation_QosPolicyAccessScopeKind	access_scope	<p>Controls the granularity used when <code>coherent_access</code> and/or <code>ordered_access</code> are TRUE.</p> <p>If both <code>coherent_access</code> and <code>ordered_access</code> are FALSE, <code>access_scope</code>'s setting has no effect.</p> <ul style="list-style-type: none"> • DDS_INSTANCE_PRESENTATION_QOS: Queue is ordered/sorted per instance • DDS_TOPIC_PRESENTATION_QOS: Queue is ordered/sorted per topic (across all instances) • DDS_GROUP_PRESENTATION_QOS: Queue is ordered/sorted per topic across all instances belonging to <i>DataWriter</i> (or <i>DataReaders</i>) within the same Publisher (or <i>Subscriber</i>). Not supported for <code>coherent_access</code> = TRUE. • DDS_HIGHEST_OFFERED_PRESENTATION_QOS: Only applies to <i>Subscribers</i>. With this setting, the <i>Subscriber</i> will use the access scope specified by each remote <i>Publisher</i>.

Table 6.25 DDS_PresentationQosPolicy

Type	Field Name	Description
DDS_Boolean	coherent_access	<p>Controls whether Connex DDS will preserve the groupings of changes made by the publishing application by means of begin_coherent_changes() and end_coherent_changes().</p> <ul style="list-style-type: none"> • DDS_BOOLEAN_FALSE:Coherency is not preserved. The value of access_scope is ignored. • DDS_BOOLEAN_TRUE:Changes made to instances within each <i>DataWriter</i> will be available to the <i>DataReader</i> as a coherent set, based on the value of access_scope. Not supported for access_scope = GROUP.
DDS_Boolean	ordered_access	<p>Controls whether Connex DDS will preserve the order of changes.</p> <ul style="list-style-type: none"> • DDS_BOOLEAN_FALSE:The order of DDS samples is only preserved for each instance, not across instances. The value of access_scope is ignored. • DDS_BOOLEAN_TRUE:The order of DDS samples from a <i>DataWriter</i> is preserved, based on the value set in access_scope.

6.4.6.1 Coherent Access

A 'coherent set' is a set of DDS data-sample modifications that must be propagated in such a way that they are interpreted at the receiver's side as a consistent set; that is, the receiver will only be able to access the data after all the modifications in the set are available at the subscribing end.

Coherency enables a publishing application to change the value of several data-instances and have those changes be seen atomically (as a cohesive set) by the readers.

Setting *coherent_access* to TRUE only behaves as described in the DDS specification when the *DataWriter* and *DataReader* are configured for *reliable* delivery. Non-reliable *DataReaders* will never receive DDS samples that belong to a coherent set.

To send a coherent set of DDS data samples, the publishing application uses the *Publisher's* **begin_coherent_changes()** and **end_coherent_changes()** operations (see [Writing Coherent Sets of DDS Data Samples \(Section 6.3.10 on page 287\)](#)).

- If *coherent_access* is TRUE, then the *access_scope* controls the maximum extent of the coherent changes, as follows:
- If *access_scope* is INSTANCE, the use of **begin_coherent_changes()** and **end_coherent_changes()** has no effect on how the subscriber can access the data. This is because, with the scope limited to

each instance, changes to separate instances are considered independent and thus cannot be grouped by a coherent change.

- If *access_scope* is TOPIC, then coherent changes (indicated by their enclosure within calls to **begin_coherent_changes()** and **end_coherent_changes()**) will be made available as such to each remote *DataReader* independently. That is, changes made to instances within the each individual *DataWriter* will be available as a coherent set with respect to other changes to instances in that same *DataWriter*, but will not be grouped with changes made to instances belonging to a different *DataWriter*.

If *access_scope* is GROUP, coherent changes made to instances through a *DataWriter* attached to a common *Publisher* are made available as a unit to remote subscribers. Coherent access with GROUP access scope is currently not supported.

6.4.6.2 Ordered Access

If *ordered_access* is TRUE, then *access_scope* controls the scope of the order in which DDS samples are presented to the subscribing application, as follows:

- If *access_scope* is INSTANCE, the relative order of DDS samples sent by a *DataWriter* is only preserved on an per-instance basis. If two DDS samples refer to the same instance (identified by *Topic* and a particular value for the key) then the order in which they are stored in the *DataReader*'s queue is consistent with the order in which the changes occurred. However, if the two DDS samples belong to different instances, the order in which they are presented may or may not match the order in which the changes occurred.
- If *access_scope* is TOPIC, the relative order of DDS samples sent by a *DataWriter* is preserved for all DDS samples of all instances. The coherent grouping and/or order in which DDS samples appear in the *DataReader*'s queue is consistent with the grouping/order in which the changes occurred—even if the DDS samples affect different instances.
- If *access_scope* is GROUP, the scope spans all instances belonging to *DataWriters* within the same *Publisher*—even if they are instances of different topics. Changes made to instances via *DataWriters* attached to the same *Publisher* are made available to *Subscribers* on the same order they occurred.
- If *access_scope* is HIGHEST_OFFERED, the *Subscriber* will use the access scope specified by each remote *Publisher*.

The data stored in the *DataReader* is accessed by the *DataReader*'s **read()/take()** APIs. The application does not have to access the DDS data samples in the same order as they are stored in the queue. How the application actually gets the data from the *DataReader* is ultimately under the control of the user code, see [Using DataReaders to Access Data \(Read & Take\) \(Section 7.4 on page 491\)](#).

6.4.6.3 Example

Coherency is useful in cases where the values are inter-related (for example, if there are two data-instances representing the altitude and velocity vector of the same aircraft and both are changed, it may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

Ordered access is useful when you need to ensure that DDS samples appear on the *DataReader's* queue in the order sent by one or multiple *DataWriters* within the same *Publisher*.

To illustrate the effect of the PRESENTATION QoS Policy with TOPIC and INSTANCE access scope, assume the following sequence of DDS samples was written by the *DataWriter*: {A1, B1, C1, A2, B2, C2}. In this example, A, B, and C represent different instances (i.e., different keys). Assume all of these DDS samples have been propagated to the *DataReader's* history queue before your application invokes the `read()` operation. The DDS data-sample sequence returned depends on how the PRESENTATION QoS is set, as shown in [Table 6.26 Effect of ordered_access for access_scope INSTANCE and TOPIC](#).

Table 6.26 Effect of ordered_access for access_scope INSTANCE and TOPIC

PRESENTATION QoS	Sequence retrieved via “read()”. Order sent was {A1, B1, C1, A2, B2, C2} Order received was {A1, A2, B1, B2, C1, C2}
ordered_access = FALSE access_scope = <any>	{A1, A2, B1, B2, C1, C2}
ordered_access = TRUE access_scope = INSTANCE	{A1, A2, B1, B2, C1, C2}
ordered_access = TRUE access_scope = TOPIC	{A1, B1, C1, A2, B2, C2}

To illustrate the effect of a PRESENTATION QoS Policy with GROUP *access_scope*, assume the following sequence of DDS samples was written by two *DataWriters*, W1 and W2, within the same *Publisher*: {(W1,A1), (W2,B1), (W1,C1), (W2,A2), (W1,B2), (W2,C2)}. As in the previous example, A, B, and C represent different instances (i.e., different keys). With *access_scope* set to INSTANCE or TOPIC, the middleware cannot guarantee that the application will receive the DDS samples in the same order they were published by W1 and W2. With *access_scope* set to GROUP, the middleware is able to provide the DDS samples in order to the application as long as the `read()/take()` operations are invoked within a `begin_access()/end_access()` block (see [Beginning and Ending Group-Ordered Access \(Section 7.2.5 on page 453\)](#)).

Table 6.27 Effect of `ordered_access` for `access_scope` GROUP

PRESENTATION QoS	Sequence retrieved via “read()”. Order sent was {(W1,A1), (W2,B1), (W1,C1), (W2,A2), (W1,B2), (W2,C2)}
<code>ordered_access = FALSE</code> or <code>access_scope = TOPIC</code> or <code>INSTANCE</code>	The order across <i>DataWriters</i> will not be preserved. DDS samples may be delivered in multiple orders. For example: {(W1,A1), (W1,C1), (W1,B2), (W2,B1), (W2,A2), (W2,C2)} {(W1,A1), (W2,B1), (W1,B2), (W1,C1), (W2,A2), (W2,C2)}
<code>ordered_access = TRUE</code> <code>access_scope = GROUP</code>	DDS samples are delivered in the same order they were published: {(W1,A1), (W2,B1), (W1,C1), (W2,A2), (W1,B2), (W2,C2)}

6.4.6.4 Properties

This QoSPolicy cannot be modified after the *Publisher* or *Subscriber* is enabled.

This QoS must be set compatibly between the *DataWriter's Publisher* and the *DataReader's Subscriber*. The compatible combinations are shown in [Table 6.28 Valid Combinations of `ordered_access` and `access_scope`, with `Subscriber's ordered_access = False`](#) and [Table 6.29 Valid Combinations of `ordered_access` and `access_scope`, with `Subscriber's ordered_access = True`](#) for *ordered_access* and [Table 6.30 Valid Combinations of Presentation Coherent Access and Access Scope](#) for *coherent_access*.

Table 6.28 Valid Combinations of `ordered_access` and `access_scope`, with `Subscriber's ordered_access = False`

{ <code>ordered_access/access_scope</code> }		Subscriber Requests:			
		False/Instance	False/Topic	False/Group	False/Highest
Publisher offers:	False/Instance	4	incompatible	incompatible	4
	False/Topic	4	4	incompatible	4
	False/Group	4	4	4	4
	True/Instance	4	incompatible	incompatible	4
	True/Topic	4	4	incompatible	4
	True/Group	4	4	4	4

Table 6.29 Valid Combinations of `ordered_access` and `access_scope`, with Subscriber's `ordered_access = True`

{ <code>ordered_access/access_scope</code> }		Subscriber Requests:			
		True/Instance	True/Topic	True/Group	True/Highest
Publisher offers:	False/Instance	incompatible	incompatible	incompatible	incompatible
	False/Topic	incompatible	incompatible	incompatible	incompatible
	False/Group	incompatible	incompatible	incompatible	incompatible
	True/Instance	4	incompatible	incompatible	4
	True/Topic	4	4	incompatible	4
	True/Group	4	4	4	4

Table 6.30 Valid Combinations of Presentation Coherent Access and Access Scope

{ <code>coherent_access/access_scope</code> }		Subscriber requests:			
		False/Instance	False/Topic	True/Instance	True/Topic
Publisher offers:	False/Instance	4	incompatible	incompatible	incompatible
	False/Topic	4	4	incompatible	incompatible
	True/Instance	4	incompatible	4	incompatible
	True/Topic	4	4	4	4

6.4.6.5 Related QoS Policies

- The [DESTINATION_ORDER QoS Policy \(Section 6.5.6 on page 365\)](#) is closely related and also affects the ordering of DDS data samples on a per-instance basis when there are multiple *DataWriters*.
- The [DATA_READER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#) may be used to configure the DDS sample ordering process in the Subscribers configured with *GROUP* or *HIGHEST_OFFERED access_scope*.

6.4.6.6 Applicable DDS Entities

- [Publishers \(Section 6.2 on page 243\)](#)
- [Subscribers \(Section 7.2 on page 440\)](#)

6.4.6.7 System Resource Considerations

The use of this policy does not significantly impact the usage of resources.

6.5 DataWriter QoS Policies

This section provides detailed information about the QoS Policies associated with a *DataWriter*. [Table 6.17 DataWriter QoS Policies](#) provides a quick reference. They are presented here in alphabetical order.

- [AVAILABILITY QoS Policy \(DDS Extension\) \(Section 6.5.1 on the next page\)](#)
- [BATCH QoS Policy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#)
- [DATA_WRITER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#)
- [DATA_WRITER_RESOURCE_LIMITS QoS Policy \(DDS Extension\) \(Section 6.5.4 on page 359\)](#)
- [DEADLINE QoS Policy \(Section 6.5.5 on page 363\)](#)
- [DESTINATION_ORDER QoS Policy \(Section 6.5.6 on page 365\)](#)
- [DURABILITY QoS Policy \(Section 6.5.7 on page 368\)](#)
- [DURABILITY SERVICE QoS Policy \(Section 6.5.8 on page 372\)](#)
- [ENTITY_NAME QoS Policy \(DDS Extension\) \(Section 6.5.9 on page 374\)](#)
- [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#)
- [LATENCYBUDGET QoS Policy \(Section 6.5.11 on page 380\)](#)
- [LIFESPAN QoS Policy \(Section 6.5.12 on page 381\)](#)
- [LIVELINESS QoS Policy \(Section 6.5.13 on page 382\)](#)
- [MULTI_CHANNEL QoS Policy \(DDS Extension\) \(Section 6.5.14 on page 386\)](#)
- [OWNERSHIP QoS Policy \(Section 6.5.15 on page 389\)](#)
- [OWNERSHIP_STRENGTH QoS Policy \(Section 6.5.16 on page 393\)](#)
- [PROPERTY QoS Policy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#)
- [PUBLISH_MODE QoS Policy \(DDS Extension\) \(Section 6.5.18 on page 397\)](#)
- [RELIABILITY QoS Policy \(Section 6.5.19 on page 400\)](#)
- [RESOURCE_LIMITS QoS Policy \(Section 6.5.20 on page 405\)](#)

- [SERVICE QoS Policy \(DDS Extension\) \(Section 6.5.21 on page 408\)](#)
- [TRANSPORT_PRIORITY QoS Policy \(Section 6.5.22 on page 409\)](#)
- [TRANSPORT_SELECTION QoS Policy \(DDS Extension\) \(Section 6.5.23 on page 411\)](#)
- [TRANSPORT_UNICAST QoS Policy \(DDS Extension\) \(Section 6.5.24 on page 412\)](#)
- [TYPESUPPORT QoS Policy \(DDS Extension\) \(Section 6.5.25 on page 416\)](#)
- [USER_DATA QoS Policy \(Section 6.5.26 on page 417\)](#)
- [WRITER_DATA_LIFECYCLE QoS Policy \(Section 6.5.27 on page 419\)](#)

6.5.1 AVAILABILITY QoS Policy (DDS Extension)

This QoS policy configures the availability of data and it is used in the context of two features:

- Collaborative DataWriters ([Availability QoS Policy and Collaborative DataWriters \(Section 6.5.1.1 on the facing page\)](#))
- Required Subscriptions ([Availability QoS Policy and Required Subscriptions \(Section 6.5.1.2 on page 339\)](#))

It contains the members listed in [Table 6.31 DDS_AvailabilityQoSPolicy](#).

Table 6.31 DDS_AvailabilityQoSPolicy

Type	Field Name	Description
DDS_Boolean	enable_required_subscriptions	Enables support for required subscriptions in a <i>DataWriter</i> . For Collaborative DataWriters: Not applicable. For Required Subscriptions: See Table 6.34 Configuring Required Subscriptions with DDS_AvailabilityQoSPolicy .
struct DDS_Duration_t	max_data_availability_waiting_time	Defines how much time to wait before delivering a DDS sample to the application without having received some of the previous DDS samples. For Collaborative DataWriters: See Table 6.33 Configuring Collaborative DataWriters with DDS_AvailabilityQoSPolicy . For Required Subscriptions: Not applicable.
struct DDS_Duration_t	max_endpoint_availability_waiting_time	Defines how much time to wait to discover <i>DataWriters</i> providing DDS samples for the same data source. For Collaborative DataWriters: See Table 6.33 Configuring Collaborative DataWriters with DDS_AvailabilityQoSPolicy . For Required Subscriptions: Not applicable.

Table 6.31 DDS_AvailabilityQoSPolicy

Type	Field Name	Description
struct DDS_Endpoint- GroupSeq	required_matched_ endpoint_groups	A sequence of endpoint groups, described in Table 6.32 struct DDS_EndpointGroup_t . For Collaborative DataWriters: See Table 6.33 Configuring Collaborative DataWriters with DDS_AvailabilityQoSPolicy . For Required Subscriptions: See Table 6.34 Configuring Required Subscriptions with DDS_AvailabilityQoSPolicy

Table 6.32 struct DDS_EndpointGroup_t

Type	Field Name	Description
char *	role_ name	Defines the role name of the endpoint group. If used in the AvailabilityQoSPolicy on a <i>DataWriter</i> , it specifies the name that identifies a Required Subscription.
int	quorum_ count	Defines the minimum number of members that satisfies the endpoint group. If used in the AvailabilityQoSPolicy on a <i>DataWriter</i> , it specifies the number of <i>DataReaders</i> with a specific role name that must acknowledge a DDS sample before the DDS sample is considered to be acknowledged by the Required Subscription.

6.5.1.1 Availability QoS Policy and Collaborative DataWriters

The *Collaborative DataWriters* feature allows you to have multiple *DataWriters* publishing DDS samples from a common logical data source. The *DataReaders* will combine the DDS samples coming from the *DataWriters* in order to reconstruct the correct order at the source. The Availability QoSPolicy allows you to configure the DDS sample combination (synchronization) process in the *DataReader*.

Each DDS sample published in a DDS domain for a given logical data source is uniquely identified by a pair (virtual GUID, virtual sequence number). DDS samples from the same data source (same virtual GUID) can be published by different *DataWriters*.

A *DataReader* will deliver a DDS sample (VGUID_n, VSN_m) to the application if one of the following conditions is satisfied:

- (GUID_n, SN_{m-1}) has already been delivered to the application.
- All the known *DataWriters* publishing VGUID_n have announced that they do not have (VGUID_n, VSN_{m-1}).

- None of the known *DataWriters* publishing VGUIDn have announced potential availability of (VGUIDn, VSNm-1) and both timeouts in this QoS policy have expired.

A *DataWriter* announces potential availability of DDS samples by using virtual heartbeats. The frequency at which virtual heartbeats are sent is controlled by the protocol parameters [virtual_heartbeat_period](#) (Section on page 350) and [samples_per_virtual_heartbeat](#) (Section on page 350) (see Table 6.37 [DDS_RtpsReliableWriterProtocol_t](#)).

Table 6.33 [Configuring Collaborative DataWriters with DDS_AvailabilityQoSPolicy](#) describes the fields of this policy when used for a Collaborative *DataWriter*.

For further information, see [Collaborative DataWriters](#) (Section Chapter 11 on page 670).

Table 6.33 Configuring Collaborative DataWriters with DDS_AvailabilityQoSPolicy

Field Name	Description for Collaborative DataWriters
max_data_availability_waiting_time	<p>Defines how much time to wait before delivering a DDS sample to the application without having received some of the previous DDS samples.</p> <p>A DDS sample identified by (VGUIDn, VSNm) will be delivered to the application if this timeout expires for the DDS sample and the following two conditions are satisfied:</p> <p>None of the known <i>DataWriters</i> publishing VGUIDn have announced potential availability of (VGUIDn, VSNm-1).</p> <p>The <i>DataWriters</i> for all the endpoint groups specified in required_matched_endpoint_groups (Section on the previous page) have been discovered or max_endpoint_availability_waiting_time (Section on the facing page) has expired.</p>
max_endpoint_availability_waiting_time	<p>Defines how much time to wait to discover <i>DataWriters</i> providing DDS samples for the same data source.</p> <p>The set of endpoint groups that are required to provide DDS samples for a data source can be configured using required_matched_endpoint_groups (Section on the previous page).</p> <p>A non-consecutive DDS sample identified by (GUIDn, SNm) cannot be delivered to the application unless the <i>DataWriters</i> for all the endpoint groups in required_matched_endpoint_groups (Section on the previous page) are discovered or this timeout expires.</p>
required_matched_endpoint_groups	<p>Specifies the set of endpoint groups that are expected to provide DDS samples for the same data source.</p> <p>The quorum count in a group represents the number of <i>DataWriters</i> that must be discovered for that group before the <i>DataReader</i> is allowed to provide non consecutive DDS samples to the application.</p> <p>A <i>DataWriter</i> becomes a member of an endpoint group by configuring the role_name in the <i>DataWriter</i>'s ENTITY_NAME QoSPolicy (DDS Extension) (Section 6.5.9 on page 374).</p> <p>The <i>DataWriters</i> created by <i>RTI Persistence Service</i> have a predefined role_name of 'PERSISTENCE_SERVICE'. For other <i>DataWriters</i>, the role_name is not set by default.</p>

6.5.1.2 Availability QoS Policy and Required Subscriptions

In the context of Required Subscriptions, the Availability QoSPolicy can be used to configure a set of required subscriptions on a *DataWriter*.

Required Subscriptions are preconfigured, named subscriptions that may leave and subsequently rejoin the network from time to time, at the same or different physical locations. Any time a required subscription is disconnected, any DDS samples that would have been delivered to it are stored for delivery if and when the subscription rejoins the network.

Table 6.34 [Configuring Required Subscriptions with DDS_AvailabilityQosPolicy](#) describes the fields of this policy when used for a Required Subscription.

For further information, see [Required Subscriptions \(Section 6.3.13 on page 294\)](#).

Table 6.34 Configuring Required Subscriptions with DDS_AvailabilityQosPolicy

Field Name	Description for Required Subscriptions
enable_required_subscriptions	Enables support for Required Subscriptions in a <i>DataWriter</i> .
max_data_availability_waiting_time	Not applicable to Required Subscriptions.
max_endpoint_availability_waiting_time	
required_matched_endpoint_groups	<p>A sequence of endpoint groups that specify the Required Subscriptions on a <i>DataWriter</i>.</p> <p>Each Required Subscription is specified by a name and a quorum count.</p> <p>The quorum count represents the number of <i>DataReaders</i> that have to acknowledge the DDS sample before it can be considered fully acknowledged for that Required Subscription.</p> <p>A <i>DataReader</i> is associated with a Required Subscription by configuring the role_name in the <i>DataReader</i>'s ENTITY_NAME QosPolicy (DDS Extension) (Section 6.5.9 on page 374).</p>

6.5.1.3 Properties

For *DataWriters*, all the members in this QosPolicy can be changed after the *DataWriter* is created except for the member **enable_required_subscriptions**.

For *DataReaders*, this QosPolicy cannot be changed after the *DataReader* is created.

There are no compatibility restrictions for how it is set on the publishing and subscribing sides.

6.5.1.4 Related QosPolicies

- [ENTITY_NAME QosPolicy \(DDS Extension\) \(Section 6.5.9 on page 374\)](#)
- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4](#)

on page 593)

- [DURABILITY QosPolicy](#) (Section 6.5.7 on page 368)

6.5.1.5 Applicable DDS Entities

- [DataWriters](#) (Section 6.3 on page 261)
- [DataReaders](#) (Section 7.3 on page 459)

6.5.1.6 System Resource Considerations

The resource limits for the endpoint groups in **required_matched_endpoint_groups** are determined by two values in the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy](#) (DDS Extension) (Section 8.5.4 on page 593):

- **max_endpoint_groups**
- **max_endpoint_group_cumulative_characters**

The maximum number of virtual writers (identified by a virtual GUID) that can be managed by a *DataReader* is determined by the **max_remote_virtual_writers** in [DATA_READER_RESOURCE_LIMITS QosPolicy](#) (DDS Extension) (Section 7.6.2 on page 517). When the *Subscriber's* **access_scope** is GROUP, **max_remote_virtual_writers** determines the maximum number of *DataWriter* groups supported by the *Subscriber*. Since the *Subscriber* may contain more than one *DataReader*, only the setting of the first applies.

6.5.2 BATCH QosPolicy (DDS Extension)

This QosPolicy can be used to decrease the amount of communication overhead associated with the transmission and (in the case of reliable communication) acknowledgement of small DDS samples, in order to increase throughput.

It specifies and configures the mechanism that allows Connex DDS to collect multiple user data DDS samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput.

This QosPolicy can be used to increase effective throughput dramatically for small data DDS samples. Throughput for small DDS samples (size < 2048 bytes) is typically limited by CPU capacity and not by network bandwidth. Batching many smaller DDS samples to be sent in a single large packet will increase network utilization and thus throughput in terms of DDS samples per second.

It contains the members listed in [Table 6.35 DDS_BatchQosPolicy](#).

Table 6.35 DDS_BatchQosPolicy

Type	Field Name	Description
DDS_Boolean	enable	Enables/disables batching.
DDS_Long	max_data_bytes	Sets the maximum cumulative length of all serialized DDS samples in a batch. Before or when this limit is reached, the batch is automatically flushed. The size does not include the meta-data associated with the batch DDS samples.
DDS_Long	max_samples	Sets the maximum number of DDS samples in a batch. When this limit is reached, the batch is automatically flushed.
struct DDS_Duration_t	max_flush_delay	Sets the maximum flush delay. When this duration is reached, the batch is automatically flushed. The delay is measured from the time the first DDS sample in the batch is written by the application.
struct DDS_Duration_t	source_timestamp_resolution	Sets the batch source timestamp resolution. The value of this field determines how the source timestamp is associated with the DDS samples in a batch. A DDS sample written with timestamp 't' inherits the source timestamp 't2' associated with the previous DDS sample, unless ('t' - 't2') is greater than source_timestamp_resolution . If source_timestamp_resolution is DURATION_INFINITE, every DDS sample in the batch will share the source timestamp associated with the first DDS sample. If source_timestamp_resolution is zero, every DDS sample in the batch will contain its own source timestamp corresponding to the moment when the DDS sample was written. The performance of the batching process is better when source_timestamp_resolution is set to DURATION_INFINITE.
DDS_Boolean	thread_safe_write	Determines whether or not the write operation is thread-safe. If TRUE, multiple threads can call write on the <i>DataWriter</i> concurrently. A setting of FALSE can be used to increase batching throughput for batches with many small DDS samples.

If batching is enabled (not the default), DDS samples are not immediately sent when they are written. Instead, they get collected into a "batch." A batch always contains whole number of DDS samples—a DDS sample will never be fragmented into multiple batches.

A batch is sent on the network ("flushed") when one of the following things happens:

- User-configurable flushing conditions
 - A batch size limit (**max_data_bytes**) is reached.
 - A number of DDS samples are in the batch (**max_samples**).

- A time-limit (**max_flush_delay**) is reached, as measured from the time the first DDS sample in the batch is written by the application.
- The application explicitly calls a *DataWriter's flush()* operation.
- Non-user configurable flushing conditions:
 - A coherent set starts or ends.
 - The number of DDS samples in the batch is equal to **max_samples** in RESOURCE_LIMITS for unkeyed topics or **max_samples_per_instance** in RESOURCE_LIMITS for keyed topics.

Additional batching configuration takes place in the *Publisher's ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)* (Section 6.4.1 on page 313).

The **flush()** operation is described in *Flushing Batches of DDS Data Samples* (Section 6.3.9 on page 287).

6.5.2.1 Synchronous and Asynchronous Flushing

Usually, a batch is flushed synchronously:

- When a batch reaches its application-defined size limit (**max_data_bytes** or **max_samples**) because the application called **write()**, the batch is flushed immediately in the context of the writing thread.
- When an application manually flushes a batch, the batch is flushed immediately in the context of the calling thread.
- When the first DDS sample in a coherent set is written, the batch in progress (without including the DDS sample in the coherent set) is immediately flushed in the context of the writing thread.
- When a coherent set ends, the batch in progress is immediately flushed in the context of the calling thread.
- When the number of DDS samples in a batch is equal to **max_samples** in RESOURCE_LIMITS for unkeyed topics or **max_samples_per_instance** in RESOURCE_LIMITS for keyed topics, the batch is flushed immediately in the context of the writing thread.

However, some behavior is asynchronous:

- To flush batches based on a time limit (**max_flush_delay**), enable asynchronous batch flushing in the *ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)* (Section 6.4.1 on page 313) of the *DataWriter's Publisher*. This will cause the *Publisher* to create an additional thread that will be used to flush batches of that *Publisher's DataWriters*. This behavior is analogous to the way asynchronous publishing works.

- You may also use batching alongside asynchronous publication with [FlowControllers \(DDS Extension\) \(Section 6.6 on page 422\)](#). These features are independent of one another. Flushing a batch on an asynchronous *DataWriter* makes it available for sending to the *DataWriter's* FlowController. From the point of view of the FlowController, a batch is treated like one large DDS sample.

6.5.2.2 Batching vs. Coalescing

Even when batching is disabled, Connex DDS will sometimes coalesce multiple DDS samples into a single network datagram. For example, DDS samples buffered by a FlowController or sent in response to a negative acknowledgement (NACK) may be coalesced. This behavior is distinct from DDS sample batching.

DDS samples that are sent individually (not part of a batch) are always treated as separate DDS samples by Connex DDS. Each DDS sample is accompanied by a complete RTPS header on the network (although DDS samples may share UDP and IP headers) and (in the case of reliable communication) a unique physical sequence number that must be positively or negatively acknowledged.

In contrast, batched DDS samples share an RTPS header and an entire batch is acknowledged—positively or negatively—as a unit, potentially reducing the amount of meta-traffic on the network and the amount of processing per individual DDS sample.

Batching can also improve latency relative to simply coalescing. Consider two use cases:

1. A *DataWriter* is configured to write asynchronously with a FlowController. Even if the FlowController's rules would allow it to publish a new DDS sample immediately, the send will always happen in the context of the asynchronous publishing thread. This context switch can add latency to the send path.
2. A *DataWriter* is configured to write synchronously but with batching turned on. When the batch is full, it will be sent on the wire immediately, eliminating a thread context switch from the send path.

6.5.2.3 Batching and ContentFilteredTopics

When batching is enabled, content filtering is always done on the reader side.

6.5.2.4 Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature

Turbo Mode is an experimental feature that uses an intelligent algorithm that automatically adjusts the number of bytes in a batch at run time according to current system conditions, such as write speed (or write frequency) and DDS sample size. This intelligence is what gives it the ability to increase throughput at high message rates and avoid negatively impacting message latency at low message rates.

To enable Turbo mode, set the *DataWriter's* property `dds.data_writer.enable_turbo_mode` to true. Turbo mode is not enabled by default.

Note: If you explicitly enable batching by setting `enable` to `TRUE` in `BatchQosPolicy`, the value of the `turbo mode` property is ignored and turbo mode is not used.

6.5.2.5 Performance Considerations

The purpose of batching is to increase throughput when writing small DDS samples at a high rate. In such cases, throughput can be increased several-fold, approaching much more closely the physical limitations of the underlying network transport.

However, collecting DDS samples into a batch implies that they are not sent on the network immediately when the application writes them; this can potentially increase latency. However, if the application sends data faster than the network can support, an increased proportion of the network's available bandwidth will be spent on acknowledgements and DDS sample resends. In this case, reducing that overhead by turning on batching could decrease latency while increasing throughput.

As a general rule, to improve batching throughput:

- Set `thread_safe_write` to `FALSE` when the batch contains a big number of small DDS samples. If you do not use a thread-safe write configuration, asynchronous batch flushing must be disabled.
- Set `source_timestamp_resolution` to `DURATION_INFINITE`. Note that you set this value, every DDS sample in the batch will share the same source timestamp.

Batching affects how often piggyback heartbeats are sent; see `heartbeats_per_max_samples` in [Table 6.37 DDS_RtpsReliableWriterProtocol_t](#).

6.5.2.6 Maximum Transport Datagram Size

Batches cannot be fragmented. As a result, the maximum batch size (`max_data_bytes`) must be set no larger than the maximum transport datagram size. For example, a UDP datagram is limited to 64 KB, so any batches sent over UDP must be less than or equal to that size.

6.5.2.7 Properties

This `QosPolicy` cannot be modified after the `DataWriter` is enabled.

Since it is only for `DataWriters`, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

All batching configuration occurs on the publishing side. A subscribing application does not configure anything specific to receive batched DDS samples, and in many cases, it will be oblivious to whether the DDS samples it processes were received individually or as part of a batch.

Consistency rules:

- **max_samples** must be consistent with **max_data_bytes**: they cannot both be set to LENGTH_UNLIMITED.
- If **max_flush_delay** is not DURATION_INFINITE, **disable_asynchronous_batch** in the [ASYNCHRONOUS_PUBLISHER QoS Policy \(DDS Extension\)](#) (Section 6.4.1 on page 313) must be FALSE.
- If **thread_safe_write** is FALSE, **source_timestamp_resolution** must be DURATION_INFINITE.

6.5.2.8 Related QoS Policies

To flush batches based on a time limit, enable batching in the [ASYNCHRONOUS_PUBLISHER QoS Policy \(DDS Extension\)](#) (Section 6.4.1 on page 313) of the *DataWriter's Publisher*.

Be careful when configuring a *DataWriter's LIFESPAN QoS Policy* (Section 6.5.12 on page 381) with a **duration** shorter than the batch flush period (**max_flush_delay**). If the batch does not fill up before the flush period elapses, the short **duration** will cause the DDS samples to be lost without being sent.

Do not configure the *DataReader's or DataWriter's HISTORY QoS Policy* (Section 6.5.10 on page 376) to be shallower than the *DataWriter's* maximum batch size (**max_samples**). When the HISTORY QoS Policy is shallower on the DataWriter, some DDS samples may not be sent. When the HISTORY QoS Policy is shallower on the DataReader, DDS samples may be dropped before being provided to the application.

The initial and maximum numbers of batches that a *DataWriter* will manage is set in the [DATA_WRITER_RESOURCE_LIMITS QoS Policy \(DDS Extension\)](#) (Section 6.5.4 on page 359).

The maximum number of DDS samples that a *DataWriter* can store is determined by the value **max_samples** in the [RESOURCE_LIMITS QoS Policy](#) (Section 6.5.20 on page 405) and **max_batches** in the [DATA_WRITER_RESOURCE_LIMITS QoS Policy \(DDS Extension\)](#) (Section 6.5.4 on page 359). The limit that is reached first is applied.

The amount of resources required for batching depends on the configuration of the [RESOURCE_LIMITS QoS Policy](#) (Section 6.5.20 on page 405) and the [DATA_WRITER_RESOURCE_LIMITS QoS Policy \(DDS Extension\)](#) (Section 6.5.4 on page 359). See [System Resource Considerations](#) (Section 6.5.2.10 below).

6.5.2.9 Applicable DDS Entities

- [DataWriters](#) (Section 6.3 on page 261)

6.5.2.10 System Resource Considerations

- Batching requires additional resources to store the meta-data associated with the DDS samples in the batch.

- For unkeyed topics, the meta-data will be at least 8 bytes, with a maximum of 20 bytes.
- For keyed topics, the meta-data will be at least 8 bytes, with a maximum of 52 bytes.
- Other resource considerations are described in [Related QosPolicies \(Section 6.5.2.8 on the previous page\)](#).

6.5.3 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)

Connex DDS uses a standard protocol for packet (user and meta data) exchange between applications. The `DataWriterProtocol` QosPolicy gives you control over configurable portions of the protocol, including the configuration of the reliable data delivery mechanism of the protocol on a per *DataWriter* basis.

These configuration parameters control timing and timeouts, and give you the ability to trade off between speed of data loss detection and repair, versus network and CPU bandwidth used to maintain reliability.

It is important to tune the reliability protocol on a per *DataWriter* basis to meet the requirements of the end-user application so that data can be sent between *DataWriters* and *DataReaders* in an efficient and optimal manner in the presence of data loss. You can also use this QosPolicy to control how Connex DDS responds to "slow" reliable *DataReaders* or ones that disconnect or are otherwise lost.

This policy includes the members presented in [Table 6.36 DDS_DataWriterProtocolQosPolicy](#) and [Table 6.37 DDS_RtpsReliableWriterProtocol_t](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

For details on the reliability protocol used by Connex DDS, see [Reliable Communications \(Section Chapter 10 on page 629\)](#). See the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) for more information on per-*DataReader/DataWriter* reliability configuration. The [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#) also play important roles in the DDS reliability protocol.

Table 6.36 DDS_DataWriterProtocolQosPolicy

Type	Field Name	Description
DDS_GUID_t	virtual_guid	<p>The virtual GUID (Global Unique Identifier) is used to uniquely identify the same <i>DataWriter</i> across multiple incarnations. In other words, this value allows Connex DDS to remember information about a <i>DataWriter</i> that may be deleted and then recreated.</p> <p>Connex DDS uses the virtual GUID to associate a durable writer history to a <i>DataWriter</i>.</p> <p>Persistence Service¹ uses the virtual GUID to send DDS samples on behalf of the original <i>DataWriter</i>.</p> <p>A <i>DataReader</i> persists its state based on the virtual GUIDs of matching remote <i>DataWriters</i>.</p> <p>For more information, see Durability and Persistence Based on Virtual GUIDs (Section 12.2 on page 680).</p> <p>By default, Connex DDS will assign a virtual GUID automatically. If you want to restore the state of the durable writer history after a restart, you can retrieve the value of the writer's virtual GUID using the <i>DataWriter</i>'s <code>get_qos()</code> operation, and set the virtual GUID of the restarted <i>DataWriter</i> to the same value.</p>
DDS_Unsigned-Long	rtps_object_id	<p>Determines the <i>DataWriter</i>'s RTPS object ID, according to the DDS-RTPS Interoperability Wire Protocol.</p> <p>Only the last 3 bytes are used; the most significant byte is ignored.</p> <p>The <code>rtps_host_id</code>, <code>rtps_app_id</code>, and <code>rtps_instance_id</code> in the WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9 on page 610), together with the 3 least significant bytes in <code>rtps_object_id</code>, and another byte assigned by Connex DDS to identify the entity type, forms the BuiltinTopicKey in <code>PublicationBuiltinTopicData</code>.</p>
DDS_Boolean	push_on_write	<p>Controls when a DDS sample is sent after <code>write()</code> is called on a <i>DataWriter</i>. If TRUE, the DDS sample is sent immediately; if FALSE, the DDS sample is put in a queue until an ACK/NACK is received from a reliable <i>DataReader</i>.</p>
DDS_Boolean	disable_positive_acks	<p>Determines whether matching <i>DataReaders</i> send positive acknowledgements (ACKs) to the <i>DataWriter</i>.</p> <p>When TRUE, the <i>DataWriter</i> will keep DDS samples in its queue for ACK-disabled readers for a minimum keep duration (see Disabling Positive Acknowledgements (Section 6.5.3.3 on page 354)).</p> <p>When strict reliability is not required, setting this to TRUE reduces overhead network traffic.</p>

^aPersistence Service is included with the Connex DDS Professional, Evaluation, and Basic package types. It saves DDS data samples so they can be delivered to subscribing applications that join the system at a later time (see [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)).

Table 6.36 DDS_DataWriterProtocolQosPolicy

Type	Field Name	Description
DDS_Boolean	disable_inline_keyhash	<p>Controls whether or not the key-hash is propagated on the wire with DDS samples.</p> <p>This field only applies to keyed writers.</p> <p>Connex DDS associates a key-hash (an internal 16-byte representation) with each key.</p> <p>When FALSE, the key-hash is sent on the wire with every data instance.</p> <p>When TRUE, the key-hash is not sent on the wire (so the readers must compute the value using the received data).</p> <p>If the <i>reader</i> is CPU bound, sending the key-hash on the wire may increase performance, because the reader does not have to get the key-hash from the data.</p> <p>If the writer is CPU bound, sending the key-hash on the wire may decrease performance, because it requires more bandwidth (16 more bytes per DDS sample).</p> <p>Setting disable_inline_keyhash to TRUE is not compatible with using RTI Database Integration Service or RTI Recording Service.</p>
DDS_Boolean	serialize_key_with_dispose	<p>Controls whether or not the serialized key is propagated on the wire with dispose notifications.</p> <p>This field only applies to keyed writers.</p> <p>RTI recommends setting this field to TRUE if there are <i>DataReaders</i> with propagate_dispose_of_unregister_instances (in the DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1 on page 511)) also set to TRUE.</p> <p>Important: When this field TRUE, batching will not be compatible with <i>RTI Data Distribution Service</i> 4.3e, 4.4b, or 4.4c—the <i>DataReaders</i> will receive incorrect data and/or encounter deserialization errors.</p>
DDS_Boolean	propagate_app_ack_with_no_response	<p>Controls whether or not a <i>DataWriter</i> receives on_application_acknowledgment() notifications with an empty or invalid response.</p> <p>When FALSE, on_application_acknowledgment() will not be invoked if the DDS sample being acknowledged has an empty or invalid response.</p>
DDS_RtpsReliableWriterProtocol_t	rtps_reliable_writer	<p>This structure includes the fields in Table 6.37 DDS_RtpsReliableWriterProtocol_t.</p>

Table 6.37 DDS_RtpsReliableWriterProtocol_t

Type	Field Name	Description
DDS_ Long	low_watermark	Queue levels that control when to switch between the regular and fast heartbeat rates (heartbeat_period (Section below) and fast_heartbeat_period (Section below)). See High and Low Watermarks (Section 6.5.3.1 on page 352).
	high_watermark	
DDS_ Duration_ t	heartbeat_period	Rates at which to send heartbeats to <i>DataReaders</i> with unacknowledged DDS samples. See Normal, Fast, and Late-Joiner Heartbeat Periods (Section 6.5.3.2 on page 353) and How Often Heartbeats are Resent (heartbeat_period) (Section 10.3.4.1 on page 645).
	fast_heartbeat_period	
	late_joiner_heartbeat_period	
DDS_ Duration_ t	virtual_heartbeat_period	The rate at which a reliable <i>DataWriter</i> will send virtual heartbeats. Virtual heartbeat informs the reliable <i>DataReader</i> about the range of DDS samples currently present for each virtual GUID in the reliable writer's queue. See Virtual Heartbeats (Section 6.5.3.6 on page 357).
DDS_ Long	samples_per_virtual_heartbeat	The number of DDS samples that a reliable <i>DataWriter</i> must publish before sending a virtual heartbeat. See Virtual Heartbeats (Section 6.5.3.6 on page 357).
DDS_ Long	max_heartbeat_retries	<p>Maximum number of <i>periodic</i> heartbeats sent without receiving an ACK/NACK packet before marking a <i>DataReader</i> 'inactive.'</p> <p>When a <i>DataReader</i> has not acknowledged all the DDS samples the reliable <i>DataWriter</i> has sent to it, and max_heartbeat_retries number of periodic heartbeats have been sent without receiving any ACK/NACK packets in return, the <i>DataReader</i> will be marked as inactive (not alive) and be ignored until it resumes sending ACK/NACKs.</p> <p>Note that <i>piggyback</i> heartbeats do <i>not</i> count towards this value.</p> <p>See Controlling How Many Times Heartbeats are Resent (max_heartbeat_retries) (Section 10.3.4.4 on page 650).</p>
DDS_ Boolean	inactivate_nonprogressing_readers	<p>Allows the <i>DataWriter</i> to treat <i>DataReaders</i> that send successive non-progressing NACK packets as inactive.</p> <p>See Treating Non-Progressing Readers as Inactive Readers (inactivate_nonprogressing_readers) (Section 10.3.4.5 on page 650).</p>
DDS_ Long	heartbeats_per_max_samples	<p>A piggyback heartbeat is sent every [current send-window size/heartbeats_per_max_samples] number of DDS samples written.</p> <p>If set to zero, no piggyback heartbeat will be sent.</p> <p>If the current send-window size is LENGTH_UNLIMITED, 100 million is assumed as the value in the calculation.</p> <p>See Configuring the Send Window Size (Section 6.5.3.4 on page 355)</p>

Table 6.37 DDS_RtpsReliableWriterProtocol_t

Type	Field Name	Description
DDS_Duration_t	min_nack_response_delay	<p>Minimum delay to respond to an ACK/NACK.</p> <p>When a reliable <i>DataWriter</i> receives an ACK/NACK from a <i>DataReader</i>, the <i>DataWriter</i> can choose to delay a while before it sends repair DDS samples or a heartbeat. This set the value of the minimum delay.</p> <p>See Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay) (Section 10.3.4.6 on page 651).</p>
DDS_Duration_t	max_nack_response_delay	<p>Maximum delay to respond to a ACK/NACK.</p> <p>This sets the value of maximum delay between receiving an ACK/NACK and sending repair DDS samples or a heartbeat.</p> <p>A longer wait can help prevent storms of repair packets if many <i>DataReaders</i> send NACKs at the same time. However, it delays the repair, and hence increases the latency of the communication.</p> <p>See Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay) (Section 10.3.4.6 on page 651).</p>
DDS_Duration_t	nack_suppression_duration	<p>How long consecutive NACKs are suppressed.</p> <p>When a reliable <i>DataWriter</i> receives consecutive NACKs within a short duration, this may trigger the <i>DataWriter</i> to send redundant repair messages. This value sets the duration during which consecutive NACKs are ignored, thus preventing redundant repairs from being sent.</p>
DDS_Long	max_bytes_per_nack_response	<p>Maximum bytes in a repair package.</p> <p>When a reliable <i>DataWriter</i> resends DDS samples, the total package size is limited to this value. Note: The reliable <i>DataWriter</i> will always send at least one sample.</p> <p>See Controlling Packet Size for Resent DDS Samples (max_bytes_per_nack_response) (Section 10.3.4.3 on page 649).</p>
DDS_Duration_t	disable_positive_acks_min_sample_keep_duration	<p>Minimum duration that a DDS sample will be kept in the <i>DataWriter</i>'s queue for ACK-disabled <i>DataReaders</i>.</p> <p>See Disabling Positive Acknowledgements (Section 6.5.3.3 on page 354) and Disabling Positive Acknowledgements (disable_positive_acks_min_sample_keep_duration) (Section 10.3.4.7 on page 652).</p>
	disable_positive_acks_max_sample_keep_duration	<p>Maximum duration that a DDS sample will be kept in the <i>DataWriter</i>'s queue for ACK-disabled readers.</p>
DDS_Boolean	disable_positive_acks_enable_adaptive_sample_keep_duration	<p>Enables automatic dynamic adjustment of the 'keep duration' in response to network congestion.</p>

Table 6.37 DDS_RtpsReliableWriterProtocol_t

Type	Field Name	Description
DDS_ Long	disable_ positive_acks_ increase_ sample_ keep_duration_ factor	When the ‘keep duration’ is dynamically controlled, the lengthening of the ‘keep duration’ is controlled by this factor, which is expressed as a percentage. When the adaptive algorithm determines that the keep duration should be increased, this factor is multiplied with the current keep duration to get the new longer keep duration. For example, if the current keep duration is 20 milliseconds, using the default factor of 150% would result in a new keep duration of 30 milliseconds.
	disable_ positive_acks_ decrease_ sample_ keep_duration_ factor	When the ‘keep duration’ is dynamically controlled, the shortening of the ‘keep duration’ is controlled by this factor, which is expressed as a percentage. When the adaptive algorithm determines that the keep duration should be decreased, this factor is multiplied with the current keep duration to get the new shorter keep duration. For example, if the current keep duration is 20 milliseconds, using the default factor of 95% would result in a new keep duration of 19 milliseconds.
DDS_ Long	min_send_ window_ size	Minimum and maximum size for the window of outstanding DDS samples. See Configuring the Send Window Size (Section 6.5.3.4 on page 355) .
	max_send_ window_ size	
DDS_ Long	send_window_ decrease_ factor	Scales the current send-window size down by this percentage to decrease the effective send-rate in response to received negative acknowledgement. See Configuring the Send Window Size (Section 6.5.3.4 on page 355) .
DDS_ Boolean	enable_ multicast_ periodic_ heartbeat	Controls whether or not periodic heartbeat messages are sent over multicast. When enabled, if a reader has a multicast destination, the writer will send its periodic HEARTBEAT messages to that destination. Otherwise, if not enabled or the reader does not have a multicast destination, the writer will send its periodic HEARTBEATs over unicast.
DDS_ Long	multicast_ resend_ threshold	Sets the minimum number of requesting readers needed to trigger a multicast resend. See Resending Over Multicast (Section 6.5.3.7 on page 357) .
DDS_ Long	send_window_ increase_ factor	Scales the current send-window size up by this percentage to increase the effective send-rate when a duration has passed without any received negative acknowledgements. See Configuring the Send Window Size (Section 6.5.3.4 on page 355)
DDS_ Duration	send_window_ update_ period	Period in which <i>DataWriter</i> checks for received negative acknowledgements and conditionally increases the send-window size when none are received. See Configuring the Send Window Size (Section 6.5.3.4 on page 355)

6.5.3.1 High and Low Watermarks

When the number of unacknowledged DDS samples in the current send-window of a reliable *DataWriter* meets or exceeds [high_watermark \(Section on page 350\)](#), the `RELIABLE_WRITER_CACHE_`

[CHANGED Status \(DDS Extension\)](#) (Section 6.3.6.8 on page 279) will be changed appropriately, a listener callback will be triggered, and the *DataWriter* will start heartbeating its matched *DataReaders* at [fast_heartbeat_period](#) (Section on page 350)

When the number of DDS samples meets or falls below [low_watermark](#) (Section on page 350), the [RELIABLE_WRITER_CACHE_CHANGED Status \(DDS Extension\)](#) (Section 6.3.6.8 on page 279) will be changed appropriately, a listener callback will be triggered, and the heartbeat rate will return to the "normal" rate ([heartbeat_period](#) (Section on page 350)).

Having both high and low watermarks (instead of one) helps prevent rapid flickering between the rates, which could happen if the number of DDS samples hovers near the cut-off point.

Increasing the high and low watermarks will make the *DataWriters* less aggressive about seeking acknowledgments for sent data, decreasing the size of traffic spikes but slowing performance.

Decreasing the watermarks will make the *DataWriters* more aggressive, increasing both network utilization and performance.

If batching is used, [high_watermark](#) (Section on page 350) and [low_watermark](#) (Section on page 350) refer to batches, not DDS samples.

When [min_send_window_size](#) (Section on the previous page) and [max_send_window_size](#) (Section on the previous page) are not equal, the low and high watermarks are scaled down linearly to stay within the current send-window size. The value provided by configuration corresponds to the high and low watermarks for the [max_send_window_size](#) (Section on the previous page).

6.5.3.2 Normal, Fast, and Late-Joiner Heartbeat Periods

The normal [heartbeat_period](#) (Section on page 350) is used until the number of DDS samples in the reliable *DataWriter*'s queue meets or exceeds [high_watermark](#) (Section on page 350); then [fast_heartbeat_period](#) (Section on page 350) is used. Once the number of DDS samples meets or drops below [low_watermark](#) (Section on page 350), the normal rate ([heartbeat_period](#) (Section on page 350)) is used again.

- [fast_heartbeat_period](#) (Section on page 350) must be \leq [heartbeat_period](#) (Section on page 350)

Increasing [fast_heartbeat_period](#) (Section on page 350) increases the speed of discovery, but results in a larger surge of traffic when the *DataWriter* is waiting for acknowledgments.

Decreasing [heartbeat_period](#) (Section on page 350) decreases the steady state traffic on the wire, but may increase latency by decreasing the speed of repairs for lost packets when the writer does not have very many outstanding unacknowledged DDS samples.

Having two periodic heartbeat rates, and switching between them based on watermarks:

- Ensures that all *DataReaders* receive all their data as quickly as possible (the sooner they receive a heartbeat, the sooner they can send a NACK, and the sooner the *DataWriter* can send repair DDS samples);
- Helps prevent the *DataWriter* from overflowing its resource limits (as its queue starts to fill, the *DataWriter* sends heartbeats faster, prompting the *DataReaders* to acknowledge sooner, allowing the *DataWriter* to purge these acknowledged DDS samples from its queue);
- Tunes the amount of network traffic. (Heartbeats and NACKs use up network bandwidth like any other traffic; decreasing the heartbeat rates, or increasing the threshold before the fast rate starts, can smooth network traffic—at the expense of discovery performance).

The [late_joiner_heartbeat_period](#) (Section on page 350) is used when a reliable *DataReader* joins after a reliable *DataWriter* (with non-volatile Durability) has begun publishing DDS samples. Once the late-joining *DataReader* has received all cached DDS samples, it will be serviced at the same rate as other reliable *DataReaders*.

- [late_joiner_heartbeat_period](#) (Section on page 350) must be \leq [heartbeat_period](#) (Section on page 350)

6.5.3.3 Disabling Positive Acknowledgements

When strict reliable communication is not required, you can configure Connex DDS so that it does *not* send positive acknowledgements (ACKs). In this case, reliability is maintained solely based on negative acknowledgements (NACKs). The removal of ACK traffic may improve middleware performance. For example, when sending DDS samples over multicast, ACK-storms that previously may have hindered *DataWriters* and consumed overhead network bandwidth are now precluded.

By default, *DataWriters* and *DataReaders* are configured with positive ACKS enabled. To disable ACKs, either:

- Configure the *DataWriter* to disable positive ACKs for all matching *DataReaders* (by setting **disable_positive_acks** to TRUE in the [DATA_WRITER_PROTOCOL QoS Policy \(DDS Extension\)](#) (Section 6.5.3 on page 347)).
- Disable ACKs for individual *DataReaders* (by setting **disable_positive_acks** to TRUE in the [DATA_READER_PROTOCOL QoS Policy \(DDS Extension\)](#) (Section 7.6.1 on page 511)).

If ACKs are disabled, instead of the *DataWriter* holding a DDS sample in its send queue until all of its *DataReaders* have ACKed it, the *DataWriter* will hold a DDS sample for a configurable duration. This “keep-duration” starts when a DDS sample is written. When this time elapses, the DDS sample is logically considered as acknowledged by its ACK-disabled readers.

The length of the “keep-duration” can be static or dynamic, depending on how **rtps_reliable_writer.disable_positive_acks_enable_adaptive_sample_keep_duration** is set.

- When the length is static, the "keep-duration" is set to the minimum (**`rtps_reliable_writer.disable_positive_acks_min_sample_keep_duration`**).
- When the length is dynamic, the "keep-duration" is dynamically adjusted between the minimum and maximum durations (**`rtps_reliable_writer.disable_positive_acks_min_sample_keep_duration`** and **`rtps_reliable_writer.disable_positive_acks_max_sample_keep_duration`**).

Dynamic adjustment maximizes throughput and reliability in response to current network conditions: when the network is congested, durations are increased to decrease the effective send rate and relieve the congestion; when the network is not congested, durations are decreased to increase the send rate and maximize throughput.

You should configure the minimum "keep-duration" to allow at least enough time for a possible NACK to be received and processed. When a *DataWriter* has both matching ACK-disabled and ACK-enabled *DataReaders*, it holds a DDS sample in its queue until all ACK-enabled *DataReaders* have ACKed it and the "keep-duration" has elapsed.

See also: [Disabling Positive Acknowledgements \(`disable_positive_acks_min_sample_keep_duration`\) \(Section 10.3.4.7 on page 652\)](#).

6.5.3.4 Configuring the Send Window Size

When a reliable *DataWriter* writes a DDS sample, it keeps the DDS sample in its queue until it has received acknowledgements from all of its subscribing *DataReaders*. The number of these outstanding DDS samples is referred to as the *DataWriter's* "send window." Once the number of outstanding DDS samples has reached the send window size, subsequent writes will block until an outstanding DDS sample is acknowledged.

Configuration of the send window sets a minimum and maximum size, which may be unlimited. The min and max send windows can be the same. When set differently, the send window will dynamically change in response to detected network congestion, as signaled by received negative acknowledgements. When NACKs are received, the *DataWriter* responds to the slowed reader by decreasing the send window by the **`send_window_decrease_factor`** to throttle down its effective send rate. The send window will not be decreased to less than the **`min_send_window_size`**. After a period (**`send_window_update_period`**) during which no NACKs are received, indicating that the reader is catching up, the *DataWriter* will increase the send window size to increase the effective send rate by the percentage specified by **`send_window_increase_factor`**. The send window will increase to no greater than the **`max_send_window_size`**.

When both **`min_send_window_size`** and **`max_send_window_size`** are unlimited, either the resource limits **`max_samples`** in **`RESOURCE_LIMITS QosPolicy`** (Section 6.5.20 on page 405) (for non-batching) or **`max_batches`** in **`DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension)`** (Section 6.5.4 on page 359) (for batching) serves as the effective **`max_send_window_size`**.

When either **max_samples** (for non-batching) or **max_batches** (for batching) is less than **max_send_window_size**, it serves as the effective **max_send_window_size**. If it is also less than **min_send_window_size**, then effectively both min and max send-window sizes are equal to **max_samples** or **max_batches**.

6.5.3.5 Propagating Serialized Keys with Disposed-Instance Notifications

This section describes the interaction between these two fields:

- **serialize_key_with_dispose** in [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347)
- **propagate_dispose_of_unregistered_instances** in [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 7.6.1 on page 511)

RTI recommends setting **serialize_key_with_dispose** to TRUE if there are *DataReaders* with **propagate_dispose_of_unregistered_instances** also set to TRUE. However, it is permissible to set one to TRUE and the other to FALSE. The following examples will help you understand how these fields work.

See also: [Disposing of Data](#) (Section 6.3.14.2 on page 299).

Example 1

1. *DataWriter*'s **serialize_key_with_dispose** = FALSE
2. *DataReader*'s **propagate_dispose_of_unregistered_instances** = TRUE
3. *DataWriter* calls **dispose()** before writing any DDS samples
4. *DataReader* calls **take()** and receives a disposed-instance notification (without a key)
5. *DataReader* calls **get_key_value()**, which returns an error because there is no key associated with the disposed-instance notification

Example 2

1. *DataWriter*'s **serialize_key_with_dispose** = TRUE
2. *DataReader*'s **propagate_dispose_of_unregistered_instances** = FALSE
3. *DataWriter* calls **dispose()** before writing any DDS samples
4. *DataReader* calls **take()**, which does not return any DDS samples because none were written, and it does not receive any disposed-instance notifications because **propagate_dispose_of_unregistered_instances** = FALSE

Example 3

1. *DataWriter*'s **serialize_key_with_dispose** = TRUE
2. *DataReader*'s **propagate_dispose_of_unregistered_instances** = TRUE
3. *DataWriter* calls **dispose()** before writing any DDS samples
4. *DataReader* calls **take()** and receives the disposed-instance notification
5. *DataReader* calls **get_key_value()** and receives the key for the disposed-instance notification

Example 4

1. *DataWriter*'s **serialize_key_with_dispose** = TRUE
2. *DataReader*'s **propagate_dispose_of_unregistered_instances** = TRUE
3. *DataWriter* calls **write()**, which writes a DDS sample with a key
4. *DataWriter* calls **dispose()**, which writes a disposed-instance notification with a key
5. *DataReader* calls **take()** and receives a DDS sample and a disposed-instance notification; both have keys
6. *DataReader* calls **get_key_value()** with no errors

6.5.3.6 Virtual Heartbeats

Virtual heartbeats announce the availability of DDS samples with the Collaborative DataWriters feature described in [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#), where multiple *DataWriters* publish DDS samples from a common logical data-source (identified by a virtual GUID).

When [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#) **access_scope** is set to TOPIC or INSTANCE on the *Publisher*, the virtual heartbeat contains information about the DDS samples contained in the *DataWriter* queue.

When presentation **access_scope** is set to GROUP on the *Publisher*, the virtual heartbeat contains information about the DDS samples in the queues of all *DataWriters* that belong to the *Publisher*.

6.5.3.7 Resending Over Multicast

Given *DataReaders* with multicast destinations, when a *DataReader* sends a NACK to request for DDS samples to be resent, the *DataWriter* can either resend them over unicast or multicast. Though resending over multicast would save bandwidth and processing for the *DataWriter*, the potential problem is that there could be *DataReaders* of the multicast group that did not request for any resends, yet they would have to process, and drop, the resent DDS samples.

Thus, to make each multicast resend more efficient, the **multicast_resend_threshold** is set as the minimum number of *DataReaders* of the same multicast group that the *DataWriter* must receive NACKs from within a single response-delay duration. This allows the *DataWriter* to coalesce near-simultaneous unicast resends into a multicast resend, and it allows a "vote" from *DataReaders* of a multicast group to exceed a threshold before resending over multicast.

The **multicast_resend_threshold** must be set to a positive value. Note that a threshold of 1 means that all resends will be sent over multicast. Also, note that a *DataWriter* with a zero NACK response-delay (i.e., both **min_nack_response_delay** and **min_nackresponse_delay** are zero) will resend over multicast only if the threshold is 1.

6.5.3.8 Example

For information on how to use the fields in [Table 6.37 DDS_RtpsReliableWriterProtocol_t](#), see [Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy](#) (Section 10.3.4 on page 645).

The following describes a use case for when to change **push_on_write** to **DDS_BOOLEAN_FALSE**. Suppose you have a system in which the data packets being sent is very small. However, you want the data to be sent reliably, and the latency between the time that data is sent to the time that data is received is not an issue. However, the total network bandwidth between the *DataWriter* and *DataReader* applications is limited.

If the *DataWriter* sends a burst of data at a high rate, it is possible that it will overwhelm the limited bandwidth of the network. If you allocate enough space for the *DataWriter* to store the data burst being sent (see [RESOURCE_LIMITS QosPolicy](#) (Section 6.5.20 on page 405)), then you can use the **push_on_write** parameter of the `DATA_WRITER_PROTOCOL` QosPolicy to delay sending the data until the reliable *DataReader* asks for it.

By setting **push_on_write** to **DDS_BOOLEAN_FALSE**, when `write()` is called on the *DataWriter*, no data is actually sent. Instead data is stored in the *DataWriter*'s send queue. Periodically, Connex DDS will be sending heartbeats informing the *DataReader* about the data that is available. So every heartbeat period, the *DataReader* will realize that the *DataWriter* has new data, and it will send an ACK/NACK, asking for them.

When *DataWriter* receives the ACK/NACK packet, it will put together a package of data, up to the size set by the parameter **max_bytes_per_nack_response**, to be sent to the *DataReader*. This method not only self-throttles the send rate, but also uses network bandwidth more efficiently by eliminating redundant packet headers when combining several small packets into one larger one. Please note that the *DataWriter* will always send at least one sample.

6.5.3.9 Properties

This QosPolicy cannot be modified after the *DataWriter* is created.

Since it is only for *DataWriters*, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

When setting the fields in this policy, the following rules apply. If any of these are false, Connex DDS returns **DDS_RETCODE_INCONSISTENT_POLICY**:

- **min_nack_response_delay** <= **max_nack_response_delay**
- **fast_heartbeat_period** <= **heartbeat_period**
- **late_joiner_heartbeat_period** <= **heartbeat_period**
- **low_watermark** < **high_watermark**
- If batching is disabled:
 - **heartbeats_per_max_samples** <= **writer_qos.resource_limits.max_samples**
- If batching is enabled:
 - **heartbeats_per_max_samples** <= **writer_qos.resource_limits.max_batches**

6.5.3.10 Related QosPolicies

- [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 7.6.1 on page 511)
- [HISTORY QosPolicy](#) (Section 6.5.10 on page 376)
- [RELIABILITY QosPolicy](#) (Section 6.5.19 on page 400)

6.5.3.11 Applicable DDS Entities

- [DataWriters](#) (Section 6.3 on page 261)

6.5.3.12 System Resource Considerations

A high **max_bytes_per_nack_response** may increase the instantaneous network bandwidth required to send a single burst of traffic for resending dropped packets.

6.5.4 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension)

This QosPolicy defines various settings that configure how *DataWriters* allocate and use physical memory for internal resources.

It includes the members in [Table 6.38 DDS_DataWriterResourceLimitsQosPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.38 DDS_DataWriterResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	initial_concurrent_blocking_threads	Initial number of threads that are allowed to concurrently block on the write() call on the same <i>DataWriter</i> .
DDS_Long	max_concurrent_blocking_threads	Maximum number of threads that are allowed to concurrently block on write() call on the same <i>DataWriter</i> .
DDS_Long	max_remote_reader_filters	Maximum number of remote <i>DataReaders</i> for which this <i>DataWriter</i> will perform content-based filtering.
DDS_Long	initial_batches	Initial number of batches that a <i>DataWriter</i> will manage if batching is enabled.
DDS_Long	max_batches	Maximum number of batches that a <i>DataWriter</i> will manage if batching is enabled. When batching is enabled, the maximum number of DDS samples that a <i>DataWriter</i> can store is limited by this value and max_samples in RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) .
DDS_DataWriterResourceLimitsInstanceReplacementKind	instance_replacement	Sets the kinds of instances allowed to be replaced when a <i>DataWriter</i> reaches instance resource limits. (See Configuring DataWriter Instance Replacement (Section 6.5.20.2 on page 407))
DDS_Boolean	replace_empty_instances	Whether to replace empty instances during instance replacement. (See Configuring DataWriter Instance Replacement (Section 6.5.20.2 on page 407))
DDS_Boolean	autoregister_instances	Whether to register automatically instances written with non-NIL handle that are not yet registered, which will otherwise return an error. This can be especially useful if the instance has been replaced.
DDS_Long	initial_virtual_writers	Initial number of virtual writers supported by a <i>DataWriter</i> .

Table 6.38 DDS_DataWriterResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	max_virtual_writers	<p>Maximum number of virtual writers supported by a <i>DataWriter</i>.</p> <p>Sets the maximum number of unique virtual writers supported by a <i>DataWriter</i>, where virtual writers are added when DDS samples are written with the virtual writer GUID.</p> <p>This field is especially relevant in the configuration of Persistence Service¹ <i>DataWriters</i>, since they publish information on behalf of multiple virtual writers.</p>
DDS_Long	max_remote_readers	The maximum number of remote readers supported by a <i>DataWriter</i> .
DDS_Long	max_app_ack_remote_readers	The maximum number of application-level acknowledging remote readers supported by a <i>DataWriter</i> .

DataWriters must allocate internal structures to handle the simultaneous blocking of threads trying to call **write()** on the same *DataWriter*, for the storage used to batch small DDS samples, and for content-based filters specified by *DataReaders*.

Most of these internal structures start at an initial size and by default, will grow as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want to bound the amount of memory that a *DataWriter* can use. By setting the initial size to the maximum size, you will prevent Connex DDS from dynamically allocating any memory after the creation of the *DataWriter*.

When setting the fields in this policy, the following rule applies. If this is false, Connex DDS returns **DDS_RETCODE_INCONSISTENT_POLICY**:

- **max_concurrent_blocking_threads** \geq **initial_concurrent_blocking_threads**

The **initial_concurrent_blocking_threads** is used to allocate necessary initial system resources. If necessary, it will be increased automatically up to the **max_concurrent_blocking_threads** limit.

Every user thread calling **write()** on a *DataWriter* may use a semaphore that will block the thread when the *DataWriter*'s send queue is full. Because user code may set a timeout, each thread must use a different semaphore. See the **max_blocking_time** parameter of the [RELIABILITY QosPolicy \(Section 6.5.19 on](#)

¹Persistence Service is included with the Connex DDS Professional, Evaluation, and Basic package types. It saves DDS data samples so they can be delivered to subscribing applications that join the system at a later time (see [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)).

page 400). This QoS is offered so that the user application can control the dynamic allocation of system resources by Connex DDS.

If you do not mind if Connex DDS dynamically allocates semaphores when needed, then you can set the **max_concurrent_blocking_threads** parameter to some large value like **MAX_INT**. However, if you know exactly how many threads will be calling **write()** on the same *DataWriter*, and you do not want Connex DDS to allocate any system resources or memory after initialization, then you should set:

```
max_concurrent_blocking_threads = initial_concurrent_blocking_threads = NUM
```

(where *NUM* is the number of threads that could possibly block concurrently).

Each *DataWriter* can perform content-based data filtering for up to **max_remote_reader_filters** number of *DataReaders*.

Values for **max_remote_reader_filters** may be.

- **0**: The *DataWriter* will not perform filtering for any *DataReader*, which means the *DataReader* will have to filter the data itself.
- **1 to (2³¹-2)**: The *DataWriter* will filter for up to the specified number of *DataReaders*. In addition, the *Datawriter* will store the result of the filtering per DDS sample per *DataReader*.
- **DDS_LENGTH_UNLIMITED**: The *DataWriter* will filter for up to (2³¹)-2 *DataReaders*. However, in this case, the *DataWriter* will not store the filtering result per DDS sample per *DataReader*. Thus, if a DDS sample is resent (such as due to a loss of reliable communication), the DDS sample will be filtered again.

For more information, see [ContentFilteredTopics \(Section 5.4 on page 212\)](#).

6.5.4.1 Example

If there are multiple threads that can write on the same *DataWriter*, and the **write()** operation may block (based on **reliability_qos.max_blocking_time** and **HISTORY** settings), you may want to set **initial_concurrent_blocking_threads** to the most likely number of threads that will block on the same *DataWriter* at the same time, and set **max_concurrent_blocking_threads** to the maximum number of threads that could potentially block in the worst case.

6.5.4.2 Properties

This QoSPolicy cannot be modified after the *DataWriter* is created.

Since it is only for *DataWriters*, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

6.5.4.3 Related QosPolicies

- [BATCH QosPolicy \(DDS Extension\)](#) (Section 6.5.2 on page 341)
- [RELIABILITY QosPolicy](#) (Section 6.5.19 on page 400)
- [HISTORY QosPolicy](#) (Section 6.5.10 on page 376)

6.5.4.4 Applicable DDS Entities

- [DataWriters](#) (Section 6.3 on page 261)

6.5.4.5 System Resource Considerations

Increasing the values in this QosPolicy will cause more memory usage and more system resource usage.

6.5.5 DEADLINE QosPolicy

On a *DataWriter*, this QosPolicy states the maximum period in which the application expects to call **write()** on the *DataWriter*, thus publishing a new DDS sample. The application may call **write()** faster than the rate set by this QosPolicy.

On a *DataReader*, this QosPolicy states the maximum period in which the application expects to receive new values for the *Topic*. The application may receive data faster than the rate set by this QosPolicy.

The DEADLINE QosPolicy has a single member, shown in [Table 6.39 DDS_DeadlineQosPolicy](#). For the default and valid range, please refer to the API Reference HTML documentation.

Table 6.39 DDS_DeadlineQosPolicy

Type	Field Name	Description
DDS_Duration_t	period	For <i>DataWriters</i> : maximum time between writing a new value of an instance. For <i>DataReaders</i> : maximum time between receiving new values for an instance.

You can use this QosPolicy during system integration to ensure that applications have been coded to meet design specifications. You can also use it during run time to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions to prevent total system failure when data is not received in time. For topics on which data is not expected to be periodic, the deadline period should be set to an infinite value.

For keyed topics, the DEADLINE QoS applies on a per-instance basis. An application must call **write()** for each known instance of the *Topic* within the **period** specified by the DEADLINE on the *DataWriter* or receive a new value for each known instance within the **period** specified by the DEADLINE on the *DataReader*. For a *DataWriter*, the deadline period begins when the instance is first written or registered. For a *DataReader*, the deadline period begins when the first DDS sample is received.

Connex DDS will modify the *OFFERED_DEADLINE_MISSED_STATUS* and call the associated method in the **DataWriterListener** (see [OFFERED_DEADLINE_MISSED Status \(Section 6.3.6.5 on page 277\)](#)) if the application fails to **write()** a value for an instance within the period set by the DEADLINE QosPolicy of the *DataWriter*.

Similarly, Connex DDS will modify the *REQUESTED_DEADLINE_MISSED_STATUS* and call the associated method in the *DataReaderListener* (see [REQUESTED_DEADLINE_MISSED Status \(Section 7.3.7.5 on page 476\)](#)) if the application fails to receive a value for an instance within the period set by the DEADLINE QosPolicy of the *DataReader*.

For *DataReaders*, the DEADLINE QosPolicy and the [TIME_BASED_FILTER QosPolicy \(Section 7.6.4 on page 526\)](#) may interact such that even though the *DataWriter* writes DDS samples fast enough to fulfill its commitment to its own DEADLINE QosPolicy, the *DataReader* may see violations of *its* DEADLINE QosPolicy. This happens because Connex DDS will drop any packets received within the **minimum_separation** set by the TIME_BASED_FILTER—packets that could satisfy the *DataReader's* deadline.

To avoid triggering the *DataReader's* deadline even though the matched *DataWriter* is meeting its own deadline, set your QoS parameters to meet the following relationship:

```
reader deadline period >= reader minimum_separation + writer deadline period
```

Although you can set the DEADLINE QosPolicy on *Topics*, its value can only be used to initialize the DEADLINE QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

6.5.5.1 Example

Suppose you have a time-critical piece of data that should be updated at least once every second. You can set the DEADLINE **period** to 1 second on both the *DataWriter* and *DataReader*. If there is no update within that time, the *DataWriter* will get an **on_offered_deadline_missed** *Listener* callback, and the *DataReader* will get **on_requested_deadline_missed**, so that both sides can handle the error situation properly.

Note that in practice, there will be latency and jitter in the time between when data is send and when data is received. Thus even if the *DataWriter* is sending data at exactly 1 second intervals, the *DataReader* may not receive the data at exactly 1 second intervals. More likely, it will *DataReader* will receive the data at 1 second plus a small variable quantity of time. Thus you should accommodate this practical reality in choosing the DEADLINE **period** as well as the actual update period of the *DataWriter* or your application may receive false indications of failure.

The DEADLINE QosPolicy also interacts with the OWNERSHIP QosPolicy when OWNERSHIP is set to **EXCLUSIVE**. If a *DataReader* fails to receive data from the highest strength *DataWriter* within its requested DEADLINE, then the *DataReaders* can fail-over to lower strength *DataWriters*, see the [OWNERSHIP QosPolicy \(Section 6.5.15 on page 389\)](#).

6.5.5.2 Properties

This QosPolicy can be changed at any time.

The deadlines on the two sides must be compatible.

DataWriter's **DEADLINE period** \leq the *DataReader*'s **DEADLINE period**.

That is, the *DataReader* cannot expect to receive DDS samples more often than the *DataWriter* commits to sending them.

If the *DataReader* and *DataWriter* have compatible deadlines, Connex DDS monitors this “contract” and informs the application of any violations. If the deadlines are incompatible, both sides are informed and communication does not occur. The **ON_OFFERED_INCOMPATIBLE_QOS** and the **ON_REQUESTED_INCOMPATIBLE_QOS** statuses will be modified and the corresponding *Listeners* called for the *DataWriter* and *DataReader* respectively.

6.5.5.3 Related QosPolicies

- [LIVELINESS QosPolicy](#) (Section 6.5.13 on page 382)
- [OWNERSHIP QosPolicy](#) (Section 6.5.15 on page 389)
- [TIME_BASED_FILTER QosPolicy](#) (Section 7.6.4 on page 526)

6.5.5.4 Applicable DDS Entities

- [Topics](#) (Section 5.1 on page 200)
- [DataWriters](#) (Section 6.3 on page 261)
- [DataReaders](#) (Section 7.3 on page 459)

6.5.5.5 System Resource Considerations

A Connex DDS-internal thread will wake up at least by the **DEADLINE period** to check to see if the deadline was missed. It may wake up faster if the last DDS sample that was published or sent was close to the last time that the deadline was checked. Therefore a short **period** will use more CPU to wake and execute the thread checking the deadline.

6.5.6 DESTINATION_ORDER QosPolicy

When multiple *DataWriters* send data for the same topic, the order in which data from different *DataWriters* are received by the applications of different *DataReaders* may be different. Thus different *DataReaders* may not receive the same "last" value when *DataWriters* stop sending data.

This policy controls how each subscriber resolves the final value of a data instance that is written by multiple *DataWriters* (which may be associated with different *Publishers*) running on different nodes.

This QoS Policy can be used to create systems that have the property of "eventual consistency." Thus intermediate states across multiple applications may be inconsistent, but when *DataWriters* stop sending changes to the same topic, all applications will end up having the same state.

Each DDS sample includes two timestamps: a source timestamp and a destination timestamp. The source timestamp is recorded by the *DataWriter* application when the data was written. The destination timestamp is recorded by the *DataReader* application when the data was received.

This QoS includes the member in [Table 6.40 DDS_DestinationOrderQoS Policy](#).

Table 6.40 DDS_DestinationOrderQoS Policy

Type	Field Name	Description
DDS_Destination-OrderQoSPolicyKind	kind	Can be either: DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
DDS_Duration_t	source_timestamp_tolerance	Allowed tolerance between source timestamps of consecutive DDS samples. Only applies when kind (above) is DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS.

Each *DataReader* can set this QoS to:

- DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS

Assuming the OWNERSHIP_STRENGTH allows it, the latest received value for the instance should be the one whose value is kept. Data will be delivered by a *DataReader* in the order in which it was received (which may lead to inconsistent final values).

- DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS

Assuming the OWNERSHIP_STRENGTH allows it, within each instance, the **source_timestamp** shall be used to determine the most recent information. *This is the only setting that*, in the case of concurrent same-strength *DataWriters* updating the same instance, *ensures all subscribers will end up with the same final value for the instance*.

Data will be delivered by a *DataReader* in the order in which it was sent. If data arrives on the network with a source timestamp earlier than the source timestamp of the last data delivered, the new data will be dropped. This ordering therefore works best when system clocks are relatively synchronized among writing machines.

Not all data sent by multiple *DataWriters* may be delivered to a *DataReader* and not all *DataReaders* will see the same data sent by *DataWriters*. However, all *DataReaders* will see the same "final" data when *DataWriters* "stop" sending data.

- For a *DataWriter* with **kind** `DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS`:
When writing a DDS sample, its timestamp must not be less than the timestamp of the previously written DDS sample. However, if it is less than the timestamp of the previously written DDS sample but the difference is less than this tolerance, the DDS sample will use the previously written DDS sample's timestamp as its timestamp. Otherwise, if the difference is greater than this tolerance, the write will fail.

See also: [Special Instructions for deleting DataWriters if you are using the ‘Timestamp’ APIs and BY_SOURCE_TIMESTAMP Destination Order: \(Section 6.3.3.1 on page 268\)](#).
- A *DataReader* with **kind** `DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS` will accept a DDS sample only if only if the source timestamp is no farther in the future from the reception timestamp than this tolerance. Otherwise, the DDS sample is rejected.

Although you can set the `DESTINATION_ORDER` QoSPolicy on *Topics*, its value can only be used to initialize the `DESTINATION_ORDER` QoSPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#).

6.5.6.1 Properties

This QoSPolicy *cannot* be modified after the *Entity* is enabled.

This QoS must be set compatibly between the *DataWriter* and the *DataReader*. The compatible combinations are shown in [Table 6.41 Valid Reader/Writer Combinations of DestinationOrder](#).

Table 6.41 Valid Reader/Writer Combinations of DestinationOrder

Destination Order		DataReader requests:	
		BY_SOURCE	BY_RECEPTION
DataWriter offers:	BY_SOURCE	4	4
	BY_RECEPTION	incompatible	4

If this QoSPolicy is set incompatibly, the `ON_OFFERED_INCOMPATIBLE_QOS` and `ON_REQUESTED_INCOMPATIBLE_QOS` statuses will be modified and the corresponding *Listeners* called for the *DataWriter* and *DataReader* respectively.

6.5.6.2 Related QosPolicies

- [OWNERSHIP QosPolicy \(Section 6.5.15 on page 389\)](#)
- [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#)

6.5.6.3 Applicable DDS Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.6.4 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5.7 DURABILITY QosPolicy

Because the publish-subscribe paradigm is connectionless, applications can create publications and subscriptions in any way they choose. As soon as a matching pair of *DataWriters* and *DataReaders* exist, then data published by the *DataWriter* will be delivered to the *DataReader*. However, a *DataWriter* may publish data before a *DataReader* has been created. For example, before you subscribe to a magazine, there have been past issues that were published.

The DURABILITY QosPolicy controls whether or not, and how, published DDS samples are stored by the *DataWriter* application for *DataReaders* that are found after the DDS samples were initially written. *DataReaders* use this QoS to request DDS samples that were published before they were created. The analogy is for a new subscriber to a magazine to ask for issues that were published in the past. These are known as ‘historical’ DDS data samples. (Reliable *DataReaders* may wait for these historical DDS samples, see [Checking DataReader Status and StatusConditions \(Section 7.3.5 on page 468\)](#).)

This QosPolicy can be used to help ensure that *DataReaders* get all data that was sent by *DataWriters*, regardless of when it was sent. This QosPolicy can increase system tolerance to failure conditions.

Exactly how many DDS samples are stored by the *DataWriter* or requested by the *DataReader* is controlled using the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#).

For more information, please see [Mechanisms for Achieving Information Durability and Persistence \(Section Chapter 12 on page 675\)](#).

The possible settings for this QoS are:

- **DDS_VOLATILE_DURABILITY_QOS**

Connex DDSis not required to send and will not deliver any DDS data samples to *DataReaders*

that are discovered after the DDS samples were initially published.

- **DDS_TRANSIENT_LOCAL_DURABILITY_QOS**

Connex DDS will store and send previously published DDS samples for delivery to newly discovered *DataReaders* as long as the *DataWriter* still exists. For this setting to be effective, you must also set the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) **kind** to Reliable (not Best Effort). Which particular DDS samples are kept depends on other QoS settings such as [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#).

- **DDS_TRANSIENT_DURABILITY_QOS**

Connex DDS will store previously published DDS samples in memory using *Persistence Service*, which will send the stored data to newly discovered *DataReaders*. Which particular DDS samples are kept and sent by *Persistence Service* depends on the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#) of the *Persistence Service DataWriters*. These QosPolicies can be configured in the *Persistence Service* configuration file or through the [DURABILITY SERVICE QosPolicy \(Section 6.5.8 on page 372\)](#) of the *DataWriters* configured with `DDS_TRANSIENT_DURABILITY_QOS`.

- **DDS_PERSISTENT_DURABILITY_QOS**

Connex DDS will store previously published DDS samples in permanent storage, like a disk, using *Persistence Service*, which will send the stored data to newly discovered *DataReaders*. Which particular DDS samples are kept and sent by *Persistence Service* depends on the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#) in the *Persistence Service DataWriters*. These QosPolicies can be configured in the *Persistence Service* configuration file or through the [DURABILITY SERVICE QosPolicy \(Section 6.5.8 on page 372\)](#) of the *DataWriters* configured with `DDS_PERSISTENT_DURABILITY_QOS`.

This QosPolicy includes the members in [Table 6.42 DDS_DurabilityQosPolicy](#). For default settings, please refer to the API Reference HTML documentation.

Table 6.42 DDS_DurabilityQosPolicy

Type	Field Name	Description
DDS_DurabilityQosPolicyKind	kind	<p><i>DDS_VOLATILE_DURABILITY_QOS:</i> Do not save or deliver old DDS samples.</p> <p><i>DDS_TRANSIENT_LOCAL_DURABILITY_QOS:</i> Save and deliver old DDS samples if the <i>DataWriter</i> still exists.</p> <p><i>DDS_TRANSIENT_DURABILITY_QOS:</i> Save and deliver old DDS samples using a memory-based service.</p> <p><i>DDS_PERSISTENCE_DURABILITY_QOS:</i> Save and deliver old DDS samples using disk-based service.</p>
DDS_Boolean	direct_communication	<p>Whether or not a TRANSIENT or PERSISTENT <i>DataReader</i> should receive DDS samples directly from a TRANSIENT or PERSISTENT <i>DataWriter</i>.</p> <p>When TRUE, a TRANSIENT or PERSISTENT <i>DataReader</i> will receive DDS samples directly from the original <i>DataWriter</i>. The <i>DataReader</i> may also receive DDS samples from Persistence Service¹ but the duplicates will be filtered by the middleware.</p> <p>When FALSE, a TRANSIENT or PERSISTENT <i>DataReader</i> will receive DDS samples only from the <i>DataWriter</i> created by Persistence Service. This ‘relay communication’ pattern provides a way to guarantee eventual consistency.</p> <p>See RTI Persistence Service (Section 12.5.1 on page 692).</p> <p>This field only applies to <i>DataReaders</i>.</p>

With this QoS policy alone, there is no way to specify or characterize the intended consumers of the information. With TRANSIENT_LOCAL, TRANSIENT, or PERSISTENT durability a *DataWriter* can be configured to keep DDS samples around for late-joiners. However, there is no way to know when the information has been consumed by all the intended recipients.

Information durability can be combined with required subscriptions in order to guarantee that DDS samples are delivered to a set of required subscriptions. For additional details on required subscriptions see [Required Subscriptions \(Section 6.3.13 on page 294\)](#) and [AVAILABILITY QoSPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#).

6.5.7.1 Example

Suppose you have a *DataWriter* that sends data sporadically and its DURABILITY **kind** is set to **VOLATILE**. If a new *DataReader* joins the system, it won’t see any data until the next time that **write()** is called on the *DataWriter*. If you want the *DataReader* to receive any data that is valid, old or new, both

¹Persistence Service is included with the Connex DDS Professional, Evaluation, and Basic package types. It saves DDS data samples so they can be delivered to subscribing applications that join the system at a later time (see [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)).

sides should set their *DURABILITY kind* to **TRANSIENT_LOCAL**. This will ensure that the *DataReader* gets some of the previous DDS samples immediately after it is enabled.

6.5.7.2 Properties

This QosPolicy cannot be modified after the Entity has been created.

The *DataWriter* and *DataReader* must use compatible settings for this QosPolicy. To be compatible, the *DataWriter* and *DataReader* must use one of the valid combinations shown in [Table 6.43 Valid Combinations of Durability ‘kind’](#).

If this QosPolicy is found to be incompatible, the **ON_OFFERED_INCOMPATIBLE_QOS** and **ON_REQUESTED_INCOMPATIBLE_QOS** statuses will be modified and the corresponding *Listeners* called for the *DataWriter* and *DataReader* respectively.

Table 6.43 Valid Combinations of Durability ‘kind’

		DataReader requests:			
		VOLATILE	TRANSIENT_LOCAL	TRANSIENT	PERSISTENT
DataWriter offers:	VOLATILE	4	incompatible	incompatible	incompatible
	TRANSIENT_LOCAL	4	4	incompatible	incompatible
	TRANSIENT	4	4	4	incompatible
	PERSISTENT	4	4	4	4

6.5.7.3 Related QosPolicies

- [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#)
- [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#)
- [DURABILITY SERVICE QosPolicy \(Section 6.5.8 on the facing page\)](#)
- [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#)

6.5.7.4 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.7.5 System Resource Considerations

Using this policy with a setting other than **VOLATILE** will cause Connex DDS to use CPU and network bandwidth to send old DDS samples to matching, newly discovered *DataReaders*. The actual amount of resources depends on the total size of data that needs to be sent.

The maximum number of DDS samples that will be kept on the *DataWriter's* queue for late-joiners and/or required subscriptions is determined by **max_samples** in RESOURCE_LIMITS Qos Policy.

System Resource Considerations With Required Subscriptions”

By default, when TRANSIENT_LOCAL durability is used in combination with required subscriptions, a *DataWriter* configured with KEEP_ALL in the HISTORY QosPolicy (Section 6.5.10 on page 376) will keep the DDS samples in its cache until they are acknowledged by all the required subscriptions. After the DDS samples are acknowledged by the required subscriptions they will be marked as reclaimable, but they will not be purged from the *DataWriter's* queue until the *DataWriter* needs these resources for new DDS samples. This may lead to a non efficient resource utilization, specially when **max_samples** is high or even UNLIMITED.

The *DataWriter's* behavior can be changed to purge DDS samples after they have been acknowledged by all the active/matching *DataReaders* and all the required subscriptions configured on the *DataWriter*. To do so, set the **dds.data_writer.history.purge_samples_after_acknowledgment** property to 1 (see PROPERTY QosPolicy (DDS Extension) (Section 6.5.17 on page 394)).

6.5.8 DURABILITY SERVICE QosPolicy

This QosPolicy is only used if the DURABILITY QosPolicy (Section 6.5.7 on page 368) is PERSISTENT or TRANSIENT *and* you are using Persistence Service, which is included with the Connex DDS Professional, Evaluation, and Basic package types. It is used to store and possibly forward the data sent by the *DataWriter* to *DataReaders* that are created after the data was initially sent.

This QosPolicy configures certain parameters of Persistence Service when it operates on the behalf of the *DataWriter*, such as how much data to store. Specifically, this QosPolicy configures the HISTORY and RESOURCE_LIMITS used by the fictitious *DataReader* and *DataWriter* used by Persistence Service.

Note however, that by default, Persistence Service will ignore the values in the DURABILITY SERVICE QosPolicy (Section 6.5.8 above) and must be configured to use those values.

For more information, please see:

- [Mechanisms for Achieving Information Durability and Persistence \(Section Chapter 12 on page 675\)](#)
- [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)
- [Configuring Persistence Service \(Section Chapter 27 on page 934\)](#)

This QosPolicy includes the members in [Table 6.44 DDS_DurabilityServiceQosPolicy](#). For default values, please refer to the API Reference HTML documentation.

Table 6.44 DDS_DurabilityServiceQosPolicy

Type	Field Name	Description
DDS_Duration_t	service_cleanup_delay	How long to keep all information regarding an instance. Can be: Zero (default): Purge disposed instances from Persistence Service immediately. However, this will only happen if use_durability_service = 1. INFINITE: Do not purge disposed instances.
DDS_HistoryQosPolicyKind	history_kind	Settings to use for the HISTORY QosPolicy (Section 6.5.10 on page 376) when recouping durable data.
DDS_Long	history_depth	
DDS_Long	max_samples	Settings to use for the RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) when feeding data to a late joiner.
	max_instances	
	max_samples_per_instance	

The **service_cleanup_delay** in this QosPolicy controls when Persistence Service may remove all information regarding a data-instances. Information on a data-instance is maintained until all of the following conditions are met:

1. The instance has been explicitly disposed (**instance_state** = NOT_ALIVE_DISPOSED).
2. All samples for the disposed instance have been acknowledged, including the dispose sample itself.
3. A time interval longer that DurabilityService QosPolicy's **service_cleanup_delay** has elapsed since the time that Connex DDS detected that the previous two conditions were met. (Note: Only values of zero or INFINITE are currently supported for **service_cleanup_delay**.)

The **service_cleanup_delay** field is useful in the situation where your application disposes an instance and it crashes before it has a chance to complete additional tasks related to the disposition. Upon restart, your application may ask for initial data to regain its state and the delay introduced by **service_cleanup_delay** will allow your restarted application to receive the information about the disposed instance and complete any interrupted tasks.

Although you can set the `DURABILITY_SERVICE` QoSPolicy on a *Topic*, this is only useful as a means to initialize the `DURABILITY_SERVICE` QoSPolicy of a *DataWriter*. A *Topic*'s `DURABILITY_SERVICE` setting does not directly affect the operation of Connex DDS, see [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#).

6.5.8.1 Properties

This QoSPolicy cannot be modified after the Entity has been enabled.

It does not apply to *DataReaders*, so there is no requirement for setting it compatibly on the sending and receiving sides.

6.5.8.2 Related QoS Policies

- [DURABILITY QoS Policy \(Section 6.5.7 on page 368\)](#)
- [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#)
- [RESOURCE_LIMITS QoS Policy \(Section 6.5.20 on page 405\)](#)

6.5.8.3 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)

6.5.8.4 System Resource Considerations

Since this QoSPolicy configures the `HISTORY` and `RESOURCE_LIMITS` used by the fictitious *DataReader* and *DataWriter* used by Persistence Service, it does have some impact on resource usage.

6.5.9 ENTITY_NAME QoS Policy (DDS Extension)

The `ENTITY_NAME` QoSPolicy assigns a name and role name to a *DomainParticipant*, *Publisher*, *Subscriber*, *DataReader*, or *DataWriter*.

How the **name** is used is strictly application-dependent.

It is useful to attach names that are meaningful to the user. These names (except for *Publishers* and *Subscribers*) are propagated during discovery so that applications can use these names to identify, in a user-context, the entities that it discovers. Also, *Connex DDS* tools will print the names of discovered entities (except for *Publishers* and *Subscribers*).

The **role_name** identifies the role of the entity. It is used by the Collaborative *DataWriter* feature (see [Availability QoS Policy and Collaborative DataWriters \(Section 6.5.1.1 on page 338\)](#)). With Durable Subscriptions, **role_name** is used to specify to which Durable Subscription the *DataReader* belongs. (see [Availability QoS Policy and Required Subscriptions \(Section 6.5.1.2 on page 339\)](#)).

This QoSPolicy contains the members listed in [Table 6.45 DDS_EntityNameQoSPolicy](#).

Table 6.45 DDS_EntityNameQoSPolicy

Type	Field Name	Description
char *	name	A null-terminated string up to 255 characters in length. To set this in XML, see Entity Names (Section 17.4.8 on page 809) .
char *	role_name	A null-terminated string up to 255 characters in length. To set this in XML, see Entity Names (Section 17.4.8 on page 809) . For Collaborative DataWriters, this name is used to specify to which endpoint group the <i>DataWriter</i> belongs. See Availability QoS Policy and Collaborative DataWriters (Section 6.5.1.1 on page 338) . For Required and Durable Subscriptions this name is used to specify to which Subscription the <i>DataReader</i> belongs. See Required Subscriptions (Section 6.3.13 on page 294) .

These names will appear in the built-in topic for the entity (see the tables in [Built-in DataReaders \(Section 16.2 on page 773\)](#)).

Prior to `get_qos()`, if the **name** and/or **role_name** field in this QoSPolicy is not null, Connex DDS assumes the memory to be valid and big enough and may write to it. If that is not desired, set **name** and/or **role_name** to NULL before calling `get_qos()` and Connex DDS will allocate adequate memory for name.

When you call the destructor of entity's QoS structure (DomainParticipantQos, DataReaderQos, or DataWriterQos) (in C++, C++/CLI, and C#) or `<entity>Qos_finalize()` (in C), Connex DDS will attempt to free the memory used for **name** and **role_name** if it is not NULL. If this behavior is not desired, set **name** and/or **role_name** to NULL before you call the destructor of entity's QoS structure or `DomainParticipantQos_finalize()`.

6.5.9.1 Properties

This QoSPolicy cannot be modified after the entity is enabled.

6.5.9.2 Related QoS Policies

- None

6.5.9.3 Applicable Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)
- [Publishers \(Section 6.2 on page 243\)](#)
- [Subscribers \(Section 7.2 on page 440\)](#)

- [DataReaders](#) (Section 7.3 on page 459)
- [DataWriters](#) (Section 6.3 on page 261)

6.5.9.4 System Resource Considerations

If the value of **name** in this QosPolicy is not NULL, some memory will be consumed in storing the information in the database, but should not significantly impact the use of resource.

6.5.10 HISTORY QosPolicy

This QosPolicy configures the number of DDS samples that Connex DDS will store locally for *DataWriters* and *DataReaders*. For keyed *Topics*, this QosPolicy applies on a per instance basis, so that Connex DDS will attempt to store the configured value of DDS samples for every instance (see [DDS Samples, Instances, and Keys](#) (Section 2.3.1 on page 14) for a discussion of keys and instances).

It includes the members seen in [Table 6.46 DDS_HistoryQosPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.46 DDS_HistoryQosPolicy

Type	Field Name	Description
DDS_HistoryQos-PolicyKind	kind	DDS_KEEP_LAST_HISTORY_QOS: keep the last <i>depth</i> number of DDS samples per instance. DDS_KEEP_ALL_HISTORY_QOS: keep all DDS samples. ¹
DDS_Long	depth	If <i>kind</i> = DDS_KEEP_LAST_HISTORY_QOS, this is how many DDS samples to keep per instance. ² if <i>kind</i> = DDS_KEEP_ALL_HISTORY_QOS, this value is ignored.

¹Connex DDS will store up to the value of the `max_samples_per_instance` parameter of the [RESOURCE_LIMITS QosPolicy](#) (Section 6.5.20 on page 405).

²depth must be \leq `max_samples_per_instance` parameter of the [RESOURCE_LIMITS QosPolicy](#) (Section 6.5.20 on page 405)

Table 6.46 DDS_HistoryQosPolicy

Type	Field Name	Description
DDS_RefilterQos-PolicyKind	refilter	<p>Specifies how a <i>DataWriter</i> should handle previously written DDS samples for a new <i>DataReader</i>.</p> <p>When a new <i>DataReader</i> matches a <i>DataWriter</i>, the <i>DataWriter</i> can be configured to perform content-based filtering on previously written DDS samples stored in the <i>DataWriter</i> queue for the new <i>DataReader</i>.</p> <p>May be:</p> <ul style="list-style-type: none"> • DDS_NONE_REFILTER_QOS Do not filter existing DDS samples for a new <i>DataReader</i>. The <i>DataReader</i> will do the filtering. • DDS_ALL_REFILTER_QOS Filter all existing DDS samples for a newly matched <i>DataReader</i>. • DDS_ON_DEMAND_REFILTER_QOS Filter existing DDS samples only when they are requested by the <i>DataReader</i>. <p>(An extension to the DDS standard.)</p>

The **kind** determines whether or not to save a configured number of DDS samples or *all* DDS samples. It can be set to either of the following:

- **DDS_KEEP_LAST_HISTORY_QOS** Connex DDS attempts to keep the latest values of the data-instance and discard the oldest ones when the limit as set by the **depth** parameter is reached; new data will overwrite the oldest data in the queue. Thus the queue acts like a circular buffer of length **depth**.
 - For a *DataWriter*: Connex DDS attempts to keep the most recent **depth** DDS samples of each instance (identified by a unique key) managed by the *DataWriter*.
 - For a *DataReader*: Connex DDS attempts to keep the most recent **depth** DDS samples received for each instance (identified by a unique key) until the application takes them via the *DataReader*'s **take()** operation. See [Accessing DDS Data Samples with Read or Take \(Section 7.4.3 on page 493\)](#) for a discussion of the difference between **read()** and **take()**.
- **DDS_KEEP_ALL_HISTORY_QOS** Connex DDS attempts to keep all of the DDS samples of a *Topic*.
 - For a *DataWriter*: Connex DDS attempts to keep all DDS samples published by the *DataWriter*.
 - For a *DataReader*: Connex DDS attempts to keep all DDS samples received by the *DataReader* for a *Topic* (both keyed and non-keyed) until the application takes them via the *DataReader*'s **take()** operation. See [Accessing DDS Data Samples with Read or Take \(Section 7.4.3 on page 493\)](#) for a discussion of the difference between **read()** and **take()**.

tion 7.4.3 on page 493) for a discussion of the difference between **read()** and **take()**.

- The value of the **depth** parameter is ignored.

The above descriptions say “attempts to keep” because the actual number of DDS samples kept is subject to the limitations imposed by the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#). All of the DDS samples of all instances of a *Topic* share a single physical queue that is allocated for a *DataWriter* or *DataReader*. The size of this queue is configured by the RESOURCE_LIMITS QosPolicy. If there are many difference instances for a *Topic*, it is possible that the physical queue may run out of space before the number of DDS samples reaches the **depth** for all instances.

In the **KEEP_ALL** case, Connex DDS can only keep as many DDS samples for a *Topic* (independent of instances) as the size of the allocated queue. Connex DDS may or may not allocate more memory when the queue is filled, depending on the settings in the RESOURCE_LIMITS QoS Policy of the *DataWriter* or *DataReader*.

This QosPolicy interacts with the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) by controlling whether or not Connex DDS guarantees that ALL of the data sent is received or if only the last N data values sent are guaranteed to be received (a reduced level of reliability using the **KEEP_LAST** setting). However, the physical sizes of the send and receive queues are *not* controlled by the History QosPolicy. The memory allocation for the queues is controlled by the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#). Also, the amount of data that is sent to new *DataReaders* who have configured their [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) to receive previously published data is controlled by the History QosPolicy.

What happens when the physical queue is filled depends both on the setting for the HISTORY QosPolicy as well as the RELIABILITY QosPolicy.

- **DDS_KEEP_LAST_HISTORY_QOS**

- If RELIABILITY is **BEST_EFFORT**: When the number of DDS samples for an instance in the queue reaches the value of **depth**, a new DDS sample for the instance will replace the oldest DDS sample for the instance in the queue.
- If RELIABILITY is **RELIABLE**: When the number of DDS samples for an instance in the queue reaches the value of **depth**, a new DDS sample for the instance will replace the oldest DDS sample for the instance in the queue—even if the DDS sample being overwritten has not been fully acknowledged as being received by all reliable *DataReaders*. This implies that the discarded DDS sample may be lost by some reliable *DataReaders*. Thus, when using the **KEEP_LAST** setting, strict reliability is not guaranteed. See [Reliable Communications \(Section Chapter 10 on page 629\)](#) for a complete discussion on Connex DDS’s reliable protocol.

- **DDS_KEEP_ALL_HISTORY_QOS**

- If RELIABILITY is **BEST_EFFORT**: If the number of DDS samples for an instance in the queue reaches the value of the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page](#)

405)’s **max_samples_per_instance** field, a new DDS sample for the instance will replace the oldest DDS sample for the instance in the queue (regardless of instance).

- If RELIABILITY is **RELIABLE**: When the number of DDS samples for an instance in the queue reaches the value of the **RESOURCE_LIMITS** QosPolicy (Section 6.5.20 on page 405)’s **max_samples_per_instance** field, then:
 - For a *DataWriter*—a new DDS sample for the instance will replace the oldest DDS sample for the instance in the sending queue—*only* if the DDS sample being overwritten has been fully acknowledged as being received by all reliable *DataReaders*. If the oldest DDS sample for the instance has not been fully acknowledged, the **write()** operation trying to enter a new DDS sample for the instance into the sending queue will block (for the **max_blocking_time** specified in the RELIABLE QosPolicy).
 - For a *DataReader*—a new DDS sample received by the *DataReader* will be discarded. Because the *DataReader* will not acknowledge the discarded DDS sample, the *DataWriter* is forced to resend the DDS sample. Hopefully, the next time the DDS sample is received, there is space for the instance in the *DataReader*’s queue to store (and accept, thus acknowledge) the DDS sample. A DDS sample will remain in the *DataReader*’s queue for one of two reasons. The more common reason is that the user application has not removed the DDS sample using the *DataReader*’s **take()** method. Another reason is that the DDS sample has been received out of order and is not available to be taken or read by the user application until all older DDS samples have been received.

Although you can set the HISTORY QosPolicy on *Topics*, its value can only be used to initialize the HISTORY QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

6.5.10.1 Example

To achieve strict reliability, you must (1) set the *DataWriter*’s and *DataReader*’s HISTORY QosPolicy to **KEEP_ALL**, and (2) set the *DataWriter*’s and *DataReader*’s RELIABILITY QosPolicy to **RELIABLE**.

See [Reliable Communications \(Section Chapter 10 on page 629\)](#) for a complete discussion on Connex DDS’s reliable protocol.

See [Controlling Queue Depth with the History QosPolicy \(Section 10.3.3 on page 644\)](#).

6.5.10.2 Properties

This QosPolicy cannot be modified after the *Entity* has been enabled.

There is no requirement that the publishing and subscribing sides use compatible values.

6.5.10.3 Related QosPolicies

- [BATCH QosPolicy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#) Do not configure the *DataReader's* **depth** to be shallower than the *DataWriter's* maximum batch size (**batch_max_data_size**). Because batches are acknowledged as a group, a *DataReader* that cannot process an entire batch will lose the remaining DDS samples in it.
- [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#)
- [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#)

6.5.10.4 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.10.5 System Resource Considerations

While this QosPolicy does not directly affect the system resources used by Connex DDS, the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#) that must be used in conjunction with the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) will affect the amount of memory that Connex DDS will allocate for a *DataWriter* or *DataReader*.

6.5.11 LATENCYBUDGET QoS Policy

This QosPolicy can be used by a DDS implementation to change how it processes and sends data that has low latency requirements. The DDS specification does not mandate whether or how this parameter is used. Connex DDS uses it to prioritize the sending of asynchronously published data; see [ASYNCHRONOUS_PUBLISHER QosPolicy \(DDS Extension\) \(Section 6.4.1 on page 313\)](#).

This QosPolicy also applies to *Topics*. The *Topic's* setting for the policy is ignored unless you explicitly make the *DataWriter* use it.

It contains the single member listed in [Table 6.47 DDS_LatencyBudgetQosPolicy](#).

Table 6.47 DDS_LatencyBudgetQosPolicy

Type	Field Name	Description
DDS_Duration_t	duration	Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

6.5.11.1 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.12 LIFESPAN QoS Policy

The purpose of this QoS is to avoid delivering stale data to the application by specifying how long the data written by a *DataWriter* is considered valid.

Each data sample written by a *DataWriter* has an associated expiration time beyond which the data should not be delivered to any application. Once the sample expires, the data will be removed from the *DataWriter* and *DataReader* caches.

The expiration time of each sample from the *DataWriter's* cache is computed by adding the duration specified by this QoS policy to the time when the sample is added to the *DataWriter's* cache. This timestamp is not necessarily equal to the sample's source timestamp that can be provided by the user using the *DataWriter's* `write_w_timestamp()` or `write_w_params()` APIs.

The expiration time of each sample from the *DataReader's* cache is computed by adding the duration to the reception timestamp.

The Lifespan QoSPolicy can be used to control how much data is stored by Connex DDS. Even if it is configured to store "all" of the data sent or received for a topic (see the [HISTORY QoSPolicy \(Section 6.5.10 on page 376\)](#)), the total amount of data it stores may be limited by the Lifespan QoSPolicy.

You may also use the Lifespan QoSPolicy to ensure that applications do not receive or act on data, commands or messages that are too old and have "expired."

It includes the single member listed in [Table 6.48 DDS_LifespanQoSPolicy](#). For the default and valid range, please refer to the API Reference HTML documentation.

Table 6.48 DDS_LifespanQoSPolicy

Type	Field Name	Description
DDS_Duration_t	duration	Maximum duration for the data's validity.

Although you can set the LIFESPAN QoSPolicy on *Topics*, its value can only be used to initialize the LIFESPAN QoSPolicies of *DataWriters*. The Topic's setting for this QoSPolicy does not directly affect the operation of Connex DDS, see [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#).

6.5.12.1 Properties

This QoS policy can be modified after the entity is enabled.

It does not apply to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

6.5.12.2 Related QoS Policies

- [BATCH QoSPolicy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#) Be careful when configuring a *DataWriter* with a Lifespan **duration** shorter than the batch flush period (**batch_flush_delay**). If the batch does not fill up before the flush period elapses, the short **duration** will cause the DDS samples to be lost without being sent.
- [DURABILITY QoSPolicy \(Section 6.5.7 on page 368\)](#)

6.5.12.3 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)

6.5.12.4 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5.13 LIVELINESS QoSPolicy

The LIVELINESS QoSPolicy specifies how Connex DDS determines whether a *DataWriter* is “alive.” A *DataWriter*’s liveliness is used in combination with the [OWNERSHIP QoSPolicy \(Section 6.5.15 on page 389\)](#) to maintain ownership of an instance (note that the [DEADLINE QoSPolicy \(Section 6.5.5 on page 363\)](#) is also used to change ownership when a *DataWriter* is still alive). That is, for a *DataWriter* to own an instance, the *DataWriter* must still be alive as well as honoring its DEADLINE contract.

It includes the members in [Table 6.49 DDS_LivelinessQoSPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.49 DDS_LivelinessQoSPolicy

Type	Field Name	Description
DDS_LivelinessQoSPolicyKind	kind	<p>DDS_AUTOMATIC_LIVELINESS_QOS: Connex DDS will automatically assert liveliness for the <i>DataWriter</i> at least as often as the lease_duration.</p> <p>DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS: The <i>DataWriter</i> is assumed to be alive if any Entity within the same <i>DomainParticipant</i> has asserted its liveliness.</p> <p>DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS: Your application must explicitly assert the liveliness of the <i>DataWriter</i> within the lease_duration.</p>
DDS_Duration_t	lease_duration	<p>The timeout by which liveliness must be asserted for the <i>DataWriter</i> or the <i>DataWriter</i> will be considered inactive or not alive.</p> <p>Additionally, for <i>DataReaders</i>, the lease_duration also specifies the maximum period at which Connex DDS will check to see if the matching <i>DataWriter</i> is still alive.</p> <p style="background-color: #ffffcc;">A <i>DataReader</i> will consider a <i>DataWriter</i> not alive if the <i>DataWriter</i> does not assert its liveliness within the <i>DataWriter's</i> lease_duration, not the <i>DataReader's</i> lease_duration.</p>
DDS_Long	assertions_per_lease_duration	<p>The number of assertions a <i>DataWriter</i> will send during a lease_duration period.</p> <p>This field only applies to <i>DataWriters</i> using DDS_AUTOMATIC_LIVELINESS_QOS kind and it is not considered during QoS compatibility checks.</p> <p>The default value is 3. A higher value will make the liveliness mechanism more robust against packet losses, but it will also increase the network traffic.</p>

Setting a *DataWriter's* **kind** of LIVELINESS specifies the mechanism that will be used to assert liveliness for the *DataWriter*. The *DataWriter's* **lease_duration** then specifies the maximum period at which packets that indicate that the *DataWriter* is still alive are sent to matching *DataReaders*.

The various mechanisms are:

- **DDS_AUTOMATIC_LIVELINESS_QOS:**

The *DomainParticipant* is responsible for automatically sending packets to indicate that the *DataWriter* is alive; this will be done at the rate determined by the **assertions_per_lease_duration** and **lease_duration** values. This setting is appropriate when the primary failure mode is that the publishing application itself dies. It does not cover the case in which the application is still alive but in an erroneous state—allowing the *DomainParticipant* to continue to assert liveliness for the *DataWriter* but preventing threads from calling **write()** on the *DataWriter*.

As long as the internal threads spawned by Connex DDS for a *DomainParticipant* are running, then the liveliness of the *DataWriter* will be asserted regardless of the state of the rest of the application.

This setting is certainly the most convenient, if the least accurate, method of asserting liveliness for a *DataWriter*.

- **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS:**

Connex DDS will assume that as long as the user application has asserted the liveliness of at least one *DataWriter* belonging to the same *DomainParticipant* or the liveliness of the *DomainParticipant* itself, then this *DataWriter* is also alive.

This setting allows the user code to control the assertion of liveliness for an entire group of *DataWriters* with a single operation on any of the *DataWriters* or their *DomainParticipant*. Its a good balance between control and convenience.

- **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS:**

The *DataWriter* is considered alive only if the user application has explicitly called operations that assert the liveliness for that particular *DataWriter*.

This setting forces the user application to assert the liveliness for a *DataWriter* which gives the user application great control over when other applications can consider the *DataWriter* to be inactive, but at the cost of convenience.

With the *MANUAL_BY_[TOPIC,PARTICIPANT]* settings, user application code can assert the liveliness of *DataWriters* either explicitly by calling the **assert_liveliness()** operation on the *DataWriter* (as well as the *DomainParticipant* for the *MANUAL_BY_PARTICIPANT* setting) or implicitly by calling **write()** on the *DataWriter*. If the application does not use either of the methods mentioned at least once every **lease_duration**, then the subscribing application may assume that the *DataWriter* is no longer alive. Sending data *MANUAL_BY_TOPIC* will cause an assert message to be sent between the *DataWriter* and its matched *DataReaders*.

Publishing applications will monitor their *DataWriters* to make sure that they are honoring their LIVELINESS QosPolicy by asserting their liveliness at least at the period set by the **lease_duration**. If Connex DDS finds that a *DataWriter* has failed to have its liveliness asserted by its **lease_duration**, an internal thread will modify the *DataWriter*'s *LIVELINESS_LOST_STATUS* and trigger its **on_liveliness_lost()** *DataWriterListener* callback if a listener exists, see [Listeners \(Section 4.4 on page 177\)](#).

Setting the *DataReader*'s **kind** of LIVELINESS requests a specific mechanism for the publishing application to maintain the liveliness of *DataWriters*. The subscribing application may want to know that the publishing application is explicitly asserting the liveliness of the matching *DataWriter* rather than inferring its liveliness through the liveliness of its *DomainParticipant* or its sibling *DataWriters*.

The *DataReader*'s **lease_duration** specifies the maximum period at which matching *DataWriters* must have their liveliness asserted. In addition, in the subscribing application Connex DDS uses an internal thread that wakes up at the period set by the *DataReader*'s **lease_duration** to see if the *DataWriter*'s **lease_duration** has been violated.

When a matching *DataWriter* is determined to be dead (inactive), Connex DDS will modify the **LIVELINESS_CHANGED_STATUS** of each matching *DataReader* and trigger that *DataReader*'s **on_liveliness_changed()** *DataReaderListener* callback (if a listener exists).

Although you can set the **LIVELINESS** QosPolicy on *Topics*, its value can only be used to initialize the **LIVELINESS** QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

For more information on Liveliness, see [Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT \(Section 14.3.1.2 on page 724\)](#).

6.5.13.1 Example

You can use **LIVELINESS** QosPolicy during system integration to ensure that applications have been coded to meet design specifications. You can also use it during run time to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions in response to disconnected *DataWriters*.

The **LIVELINESS** QosPolicy can be used to manage fail-over when the **OWNERSHIP** QosPolicy ([Section 6.5.15 on page 389](#)) is set to **EXCLUSIVE**. This implies that the *DataReader* will only receive data from the highest strength *DataWriter* that is alive (active). When that *DataWriter*'s liveliness expires, then Connex DDS will start delivering data from the next highest strength *DataWriter* that is still alive.

6.5.13.2 Properties

This QosPolicy cannot be modified after the Entity has been enabled.

The *DataWriter* and *DataReader* must use compatible settings for this QosPolicy. To be compatible, *both* of the following conditions must be true:

The *DataWriter* and *DataReader* must use one of the valid combinations shown in [Table 6.50 Valid Combinations of Liveliness 'kind'](#).

DataWriter's **lease_duration** \leq *DataReader*'s **lease_duration**.

If this QosPolicy is found to be incompatible, the **ON_OFFERED_INCOMPATIBLE_QOS** and **ON_REQUESTED_INCOMPATIBLE_QOS** statuses will be modified and the corresponding *Listeners* called for the *DataWriter* and *DataReader* respectively.

Table 6.50 Valid Combinations of Liveliness ‘kind’

		DataReader requests:		
		MANUAL_BY_TOPIC	MANUAL_BY_PARTICIPANT	AUTOMATIC
DataWriter offers:	MANUAL_BY_TOPIC	4	4	4
	MANUAL_BY_PARTICIPANT	incompatible	4	4
	AUTOMATIC	incompatible	incompatible	4

6.5.13.3 Related QosPolicies

- [DEADLINE QosPolicy](#) (Section 6.5.5 on page 363)
- [OWNERSHIP QosPolicy](#) (Section 6.5.15 on page 389)
- [OWNERSHIP_STRENGTH QosPolicy](#) (Section 6.5.16 on page 393)

6.5.13.4 Applicable Entities

- [Topics](#) (Section 5.1 on page 200)
- [DataWriters](#) (Section 6.3 on page 261)
- [DataReaders](#) (Section 7.3 on page 459)

6.5.13.5 System Resource Considerations

An internal thread in Connex DDS will wake up periodically to check the liveliness of all the *DataWriters*. This happens both in the application that contains the *DataWriters* at the **lease_duration** set on the *DataWriters* as well as the applications that contain the *DataReaders* at the **lease_duration** set on the *DataReaders*. Therefore, as **lease_duration** becomes smaller, more CPU will be used to wake up threads and perform checks. A short **lease_duration** (or a high **assertions_per_lease_duration**) set on *DataWriters* may also use more network bandwidth because liveliness packets are being sent at a higher rate—this is especially true when LIVENESS kind is set to AUTOMATIC.

6.5.14 MULTI_CHANNEL QosPolicy (DDS Extension)

This QosPolicy is used to partition the data published by a *DataWriter* across multiple *channels*. A *channel* is defined by a filter expression and a sequence of multicast locators.

By using this QosPolicy, a *DataWriter* can be configured to send data to different multicast groups based on the content of the data. Using syntax similar to those used in Content-Based Filters, you can associate

different multicast addresses with filter expressions that operate on the values of the fields within the data. When your application's code calls `write()`, data is sent to any multicast address for which the data passes the filter.

See [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#) for complete documentation on multi-channel *DataWriters*.

Note: Durable writer history is not supported for multi-channel *DataWriters* (see [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#)); an error is reported if a multi-channel *DataWriter* tries to configure Durable Writer History.

This QosPolicy includes the members presented in [Table 6.51 DDS_MultiChannelQosPolicy](#), [Table 6.52 DDS_ChannelSettings_t](#), and [Table 6.53 DDS_TransportMulticastSettings_t](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.51 DDS_MultiChannelQosPolicy

Type	Field Name	Description
DDS_ChannelSettingsSeq	channels	A sequence of channel settings used to configure the channels' properties. If the length of the sequence is zero, the QosPolicy will be ignored. See Table 6.52 DDS_ChannelSettings_t .
char *	filter_name	Name of the filter class used to describe the filter expressions ¹ . The following values are supported: DDS_SQLFILTER_NAME (see SQL Filter Expression Notation (Section 5.4.6 on page 222)) DDS_STRINGMATCHFILTER_NAME (see STRINGMATCH Filter Expression Notation (Section 5.4.7 on page 231))

Table 6.52 DDS_ChannelSettings_t

Type	Field Name	Description
DDS_TransportMulticastSettingsSeq	multicast_settings	A sequence of multicast settings used to configure the multicast addresses associated with a channel. The sequence cannot be empty. The maximum number of multicast locators in a channel is limited to four. (A locator is defined by a transport alias, a multicast address and a port.) See Table 6.53 DDS_TransportMulticastSettings_t .

¹ In Java and C#, you can access the names of the built-in filters by using `DomainParticipant.SQLFILTER_NAME` and `DomainParticipant.STRINGMATCHFILTER_NAME`.

Table 6.52 DDS_ChannelSettings_t

Type	Field Name	Description
char *	filter_expression	A logical expression used to determine the data that will be published in the channel. This string cannot be NULL. An empty string always evaluates to TRUE. See SQL Filter Expression Notation (Section 5.4.6 on page 222) and STRINGMATCH Filter Expression Notation (Section 5.4.7 on page 231) for expression syntax.
DDS_Long	priority	A positive integer designating the relative priority of the channel, used to determine the transmission order of pending transmissions. Larger numbers have higher priority. To use publication priorities, the <i>DataWriter's</i> <code>PUBLISH_MODE</code> QosPolicy (DDS Extension) (Section 6.5.18 on page 397) must be set for asynchronous publishing and the <i>DataWriter</i> must use a FlowController that is configured for highest-priority-first (HPF) scheduling. See Prioritized DDS Samples (Section 6.6.4 on page 428) . Note: Prioritized DDS samples are not supported when using the Java, Ada, or .NET APIs. Therefore the <i>priority</i> field does not exist when using these APIs.

Table 6.53 DDS_TransportMulticastSettings_t

Type	Field Name	Description
DDS_StringSeq	transports	A sequence of transport aliases that specifies which transport should be used to publish multicast messages for this channel.
char *	receive_address	A multicast group address on which <i>DataReaders</i> subscribing to this channel will receive data.
DDS_Long	receive_port	The multicast port on which <i>DataReaders</i> subscribing to this channel will receive data.

The format of the **filter_expression** should correspond to one of the following filter classes:

- DDS_SQLFILTER_NAME (see [SQL Filter Expression Notation \(Section 5.4.6 on page 222\)](#))
- DDS_STRINGMATCHFILTER_NAME (see [STRINGMATCH Filter Expression Notation \(Section 5.4.7 on page 231\)](#))

A *DataReader* can use the ContentFilteredTopic API (see [Using a ContentFilteredTopic \(Section 5.4.5 on page 219\)](#)) to subscribe to a subset of the channels used by a *DataWriter*.

6.5.14.1 Example

See [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#).

6.5.14.2 Properties

This QosPolicy cannot be modified after the *DataWriter* is created.

It does not apply to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

6.5.14.3 Related Qos Policies

- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#)

6.5.14.4 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)

6.5.14.5 System Resource Considerations

The following fields in the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#) configure the resources associated with the channels stored in the MULTI_CHANNEL QosPolicy:

- **channel_seq_max_length**
- **channel_filter_expression_max_length**

For information about partitioning topic data across multiple channels, please refer to [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#).

6.5.15 OWNERSHIP QosPolicy

The OWNERSHIP QosPolicy specifies whether a *DataReader* receive data for an instance of a *Topic* sent by multiple *DataWriters*.

For non-keyed *Topics*, there is only one instance of the *Topic*.

This policy includes the single member shown in [Table 6.54 DDS_OwnershipQosPolicy](#).

Table 6.54 DDS_OwnershipQosPolicy

Type	Field Name	Description
DDS_OwnershipQosPolicyKind	kind	DDS_SHARED_OWNERSHIP_QOS or DDS_EXCLUSIVE_OWNERSHIP_QOS

The **kind** of OWNERSHIP can be set to one of two values:

- **SHARED Ownership**

When OWNERSHIP is **SHARED**, and multiple *DataWriters* for the *Topic* publishes the value of the same instance, all the updates are delivered to subscribing *DataReaders*. So in effect, there is no “owner;” no single *DataWriter* is responsible for updating the value of an instance. The subscribing application will receive modifications from all *DataWriters*.

- **EXCLUSIVE Ownership**

When OWNERSHIP is **EXCLUSIVE**, each instance can only be owned by one *DataWriter* at a time. This means that a single *DataWriter* is identified as the exclusive owner whose updates are allowed to modify the value of the instance for matching *DataWriters*. Other *DataWriters* may submit modifications for the instance, but only those made by the current owner are passed on to the *DataReaders*. If a non-owner *DataWriter* modifies an instance, no error or notification is made; the modification is simply ignored. The owner of the instance can change dynamically.

Note for non-keyed *Topics*, **EXCLUSIVE** ownership implies that *DataReaders* will pay attention to only one *DataWriter* at a time because there is only a single instance. For keyed *Topics*, *DataReaders* may actually receive data from multiple *DataWriters* when different *DataWriters* own different instances of the *Topic*.

This QosPolicy is often used to help users build systems that have redundant elements to safeguard against component or application failures. When systems have active and hot standby components, the Ownership QosPolicy can be used to ensure that data from standby applications are only delivered in the case of the failure of the primary.

The Ownership QosPolicy can also be used to create data channels or topics that are designed to be taken over by external applications for testing or maintenance purposes.

Although you can set the OWNERSHIP QosPolicy on *Topics*, its value can only be used to initialize the OWNERSHIP QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

6.5.15.1 How Connex DDS Selects which DataWriter is the Exclusive Owner

When OWNERSHIP is **EXCLUSIVE**, the owner of an instance at any given time is the *DataWriter* with the highest **OWNERSHIP_STRENGTH** QosPolicy (Section 6.5.16 on page 393) that is “alive” as defined by the **LIVELINESS** QosPolicy (Section 6.5.13 on page 382) and has not violated the **DEADLINE** QosPolicy (Section 6.5.5 on page 363) of the *DataReader*. **OWNERSHIP_STRENGTH** is simply an integer set by the *DataWriter*.

If the *Topic*’s data type is keyed (see **DDS Samples, Instances, and Keys** (Section 2.3.1 on page 14)), **EXCLUSIVE** ownership is determined on a per-instance basis. That is, the *DataWriter* owner of each instance is considered separately. A *DataReader* can receive values written by a lower strength *DataWriter* as long as those values are for instances that are not being written by a higher-strength *DataWriter*.

If there are multiple *DataWriters* with the same **OWNERSHIP_STRENGTH** writing to the same instance, Connex DDS resolves the tie by choosing the *DataWriter* with the smallest GUID (Globally Unique Identifier, see **Simple Participant Discovery** (Section 14.1.1 on page 710)). This means that different *DataReaders* (in different applications) of the same *Topic* will all choose the same *DataWriter* as the owner when there are multiple *DataWriters* with the same strength.

The owner of an instance can change when:

- A *DataWriter* with a higher **OWNERSHIP_STRENGTH** publishes a value for the instance.
- The **OWNERSHIP_STRENGTH** of the owning *DataWriter* is dynamically changed to be less than the strength of an existing *DataWriter* of the instance.
- The owning *DataWriter* stops asserting its **LIVELINESS** (the *DataWriter* dies).
- The owning *DataWriter* violates the **DEADLINE** QosPolicy by not updating the value of the instance within the period set by the **DEADLINE**.

Note however, the change of ownership is not synchronous across different *DataReaders* in different participants. That is, *DataReaders* in different applications may not determine that the ownership of an instance has changed at exactly the same time.

6.5.15.2 Example

OWNERSHIP is really a property that is shared between *DataReaders* and *DataWriters* of a *Topic*. However, in a system, some *Topics* will be exclusively owned and others will be shared. System requirements will determine which are which.

An example of a *Topic* that may be shared is one that is used by applications to publish alarm messages. If the application detects an anomalous condition, it will use a *DataWriter* to write a *Topic* “Alarm.” Another application that records alarms into a system log file will have a *DataReader* that subscribes to “Alarm.” In this example, any number of applications can publish the “Alarm” message. There is no concept that only

one application at a time is allowed to publish the “Alarm” message, so in this case, the **OWNERSHIP** of the *DataWriters* and *DataReaders* should be set to **SHARED**.

In a different part of the system, **EXCLUSIVE OWNERSHIP** may be used to implement redundancy in support of fault tolerance. Say, the distributed system controls a traffic system. It monitors traffic and changes the information posted on signs, the operation of metering lights, and the timing of traffic lights. This system must be tolerant to failure of any part of the system including the application that actually issues commands to change the lights at a particular intersection.

One way to implement fault tolerance is to create the system redundantly both in hardware and software. So if a piece of the running system fails, a backup can take over. In systems where failover from the primary to backup system must be seamless and transparent, the actual mechanics of failover must be fast, and the redundant component must immediately pick up where the failed component left off. For the network connections of the component, Connex DDS can provide redundant *DataWriter* and *DataReaders*.

In this case, you would not want the *DataReaders* to receive redundant messages from the redundant *DataWriters*. Instead you will want the *DataReaders* to only receive messages from the primary application and only from a backup application when a failure occurs. To continue our example, if we have redundant applications that all try to control the lights at an intersection, we would want the *DataReaders* on the light to receive messages only from the primary application. To do so, we should configure the *DataWriters* and *DataReaders* to have **EXCLUSIVE OWNERSHIP** and set the **OWNERSHIP_STRENGTH** differently on different redundant applications to distinguish between primary and backup systems.

6.5.15.3 Properties

This QoSPolicy cannot be modified after the *Entity* is enabled.

It must be set to the same **kind** on both the publishing and subscribing sides. If a *DataWriter* and *DataReader* of the same topic are found to have different **kinds** set for the **OWNERSHIP** QoS, the **ON_OFFERED_INCOMPATIBLE_QOS** and **ON_REQUESTED_INCOMPATIBLE_QOS** statuses will be modified and the corresponding *Listeners* called for the *DataWriter* and *DataReader* respectively.

6.5.15.4 Related QoS Policies

- [DEADLINE](#) QoSPolicy (Section 6.5.5 on page 363)
- [LIVELINESS](#) QoSPolicy (Section 6.5.13 on page 382)
- [OWNERSHIP_STRENGTH](#) QoSPolicy (Section 6.5.16 on the next page)

6.5.15.5 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.15.6 System Resource Considerations

This QosPolicy does not significantly impact the use of system resources.

6.5.16 OWNERSHIP_STRENGTH QosPolicy

The OWNERSHIP_STRENGTH QosPolicy is used to rank *DataWriters* of the same instance of a *Topic*, so that Connex DDS can decide which *DataWriter* will have ownership of the instance when the [OWNERSHIP QosPolicy \(Section 6.5.15 on page 389\)](#) is set to **EXCLUSIVE**.

It includes the member in [Table 6.55 DDS_OwnershipStrengthQosPolicy](#). For the default and valid range, please refer to the API Reference HTML documentation.

Table 6.55 DDS_OwnershipStrengthQosPolicy

Type	Field Name	Description
DDS_Long	value	The strength value used to arbitrate among multiple <i>DataWriters</i> .

This QosPolicy only applies to *DataWriters* when **EXCLUSIVE OWNERSHIP** is used. The strength is simply an integer value, and the *DataWriter* with the largest value is the owner. A deterministic method is used to decide which *DataWriter* is the owner when there are multiple *DataWriters* that have equal strengths. See [How Connex DDS Selects which DataWriter is the Exclusive Owner \(Section 6.5.15.1 on page 391\)](#) for more details.

6.5.16.1 Example

Suppose there are two *DataWriters* sending DDS samples of the same *Topic* instance, one as the main *DataWriter*, and the other as a backup. If you want to make sure the *DataReader* always receive from the main one whenever possible, then set the main *DataWriter* to use a higher **ownership_strength** value than the one used by the backup *DataWriter*.

6.5.16.2 Properties

This QosPolicy can be changed at any time.

It does not apply to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

6.5.16.3 Related QosPolicies

- [OWNERSHIP QosPolicy \(Section 6.5.15 on page 389\)](#)

6.5.16.4 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)

6.5.16.5 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5.17 PROPERTY QosPolicy (DDS Extension)

The PROPERTY QosPolicy stores name/value (string) pairs that can be used to configure certain parameters of ConnexT DDS that are not exposed through formal QoS policies.

It can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery. This is similar to the USER_DATA QosPolicy, except this policy uses (name, value) pairs, and you can select whether or not a particular pair should be propagated (included in the built-in topic).

It includes the member in [Table 6.56 DDS_PropertyQosPolicy](#).

Table 6.56 DDS_PropertyQosPolicy

Type	Field Name	Description
DDS_PropertySeq	value	A sequence of: (name, value) pairs and booleans that indicate whether the pair should be propagated (included in the entity's built-in topic upon discovery).

The Property QoS stores name/value pairs for an Entity. Both the name and value are strings. Certain configurable parameters for Entities that do not have a formal DDS QoS definition may be configured via this QoS by using a pre-defined name and the desired setting in string form.

You can manipulate the sequence of properties (name, value pairs) with the standard methods available for sequences. You can also use the helper class, DDSPropertyQosPolicyHelper, which provides another way to work with a PropertyQosPolicy object.

The PropertyQosPolicy may be used to configure:

- Durable writer history (see [How To Configure Durable Writer History \(Section 12.3.2 on page 683\)](#))

- Durable reader state (see [How To Configure a DataReader for Durable Reader State \(Section 12.4.4 on page 690\)](#))
- Built-in and extension Transport Plugins (see [Setting Builtin Transport Properties with the PropertyQoSPolicy \(Section 15.6 on page 748\)](#), [Setting Up a Transport with the Property QoS \(Section 25.2 on page 915\)](#), [Configuring the TCP Transport \(Section 35.1 on page 993\)](#)).
- Automatic registration of built-in types (see [Registering Built-in Types \(Section 3.2.1 on page 30\)](#))
- Clock Selection ([Section 8.6 on page 619](#))
- [Turbo Mode and Automatic Throttling for DataWriter Performance—Experimental Features \(Section 6.3.18 on page 312\)](#)
- Location or content of your license from RTI (see [License Management, in the Getting Started Guide](#))

In addition, you can add your own name/value pairs to the Property QoS of an Entity. You may also use this QoSPolicy to direct Connex DDS to propagate these name/value pairs with the discovery information for the Entity. Applications that discover the Entity can then access the user-specific name/value pairs in the discovery information of the remote Entity. This allows you to add meta-information about an Entity for application-specific use, for example, authentication/authorization certificates (which can also be done using the User or Group Data QoS).

Reasons for using the PropertyQoSPolicy include:

- Some features can only be configured through the PropertyQoSPolicy, not through other QoS or APIs. For example, Durable Reader State, Durable Writer History, Built-in Types, Monotonic Clock.
- Alternative way to configure built-in transports settings. For example, to use non-default values for the built-in transports without using the PropertyQoSPolicy, you would have to create a *DomainParticipant* disabled, change the built-in transport property settings, then enable the *DomainParticipant*. Using the PropertyQoSPolicy to configure built-in transport settings will save you the work of enabling and disabling the *DomainParticipant*. Also, transport settings are not a QoS and therefore cannot be configured through an XML file. By configuring built-in transport settings through the PropertyQoSPolicy instead, XML files can be used.

When using the Java or .NET APIs, transport configuration must take place through the PropertyQoSPolicy (not through the transport property structures).

- Alternative way to support multiple instances of built-in transports (without using Transport API).
- Alternative way to dynamically load extension transports (such as *RTI Secure WAN Transport*¹ or *RTI TCP Transport*²) or user-created transport plugins in C/C++ language bindings. If the extension or user-created transport plugin is installed using the transport API instead, the library that extra transport library/code will need to be linked into your application and may require recompilation.
- Allows full pluggable transport configuration for non-C/C++ language bindings (Java, C++/CLI, C#, etc.) The pluggable transport API is not available in those languages. Without using PropertyQoSPolicy, you cannot use extension transports (such as *RTI Secure WAN Transport*) and you cannot create your own custom transport.
- Alternative way to provide a license for platforms that do not support a file system, or if a default license location is not feasible and environment variables are not supported.

The PropertyQoSPolicyHelper operations are described in [Table 6.57 PropertyQoSPolicyHelper Operations](#). For more information, see the API Reference HTML documentation.

Table 6.57 PropertyQoSPolicyHelper Operations

Operation	Description
get_number_of_properties	Gets the number of properties in the input policy.
assert_property	Asserts the property identified by name in the input policy. (Either adds it, or replaces an existing one.)
add_property	Adds a new property to the input policy.
assert_pointer_property	Asserts the property identified by name in the input policy. Used when the property to store is a pointer.
add_pointer_property	Adds a new property to the input policy. Used when the property to store is a pointer.
lookup_property	Searches for a property in the input policy given its name.
remove_property	Removes a property from the input policy.
get_properties	Retrieves a list of properties whose names match the input prefix.

¹*RTI Secure WAN Transport* is an optional component that is installed separately.

²*RTI TCP Transport* is included with your Connex DDS distribution but is not a built-in transport and therefore not enabled by default.

6.5.17.1 Properties

This QosPolicy can be changed at any time.

There is no requirement that the publishing and subscribing sides use compatible values.

6.5.17.2 Related QosPolicies

- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#)

6.5.17.3 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)
- [DomainParticipants \(Section 8.3 on page 547\)](#)

6.5.17.4 System Resource Considerations

The [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#) contains several fields for configuring the resources associated with the properties stored in this QosPolicy.

6.5.18 PUBLISH_MODE QosPolicy (DDS Extension)

This QosPolicy determines the *DataWriter's* publishing mode, either asynchronous or synchronous.

The publishing mode controls whether data is written synchronously—in the context of the user thread when calling `write()`, or asynchronously—in the context of a separate thread internal to Connex DDS.

Note: Asynchronous *DataWriters* do not perform sender-side filtering. Any filtering, such as time-based or content-based filtering, takes place on the *DataReader* side.

Each *Publisher* spawns a single asynchronous publishing thread (set in its [ASYNCHRONOUS_PUBLISHER QosPolicy \(DDS Extension\) \(Section 6.4.1 on page 313\)](#)) to serve all its asynchronous *DataWriters*.

When data is written asynchronously, a FlowController ([FlowControllers \(DDS Extension\) \(Section 6.6 on page 422\)](#)), identified by `flow_controller_name`, can be used to shape the network traffic. The FlowController's properties determine when the asynchronous publishing thread is allowed to send data and how much.

The fastest way for Connex DDS to send data is for the user thread to execute the middleware code that actually sends the data itself. However, there are times when user applications may need or want an internal middleware thread to send the data instead. For instance, for sending large data reliably, an asyn-

chronous thread must be used (see [ASYNCHRONOUS_PUBLISHER QosPolicy \(DDS Extension\)](#) (Section 6.4.1 on page 313)).

This QosPolicy can select a FlowController to prioritize or shape the data flow sent by a *DataWriter* to *DataReaders*. Shaping a data flow usually means limiting the maximum data rates with which the middleware will send data for a *DataWriter*. The FlowController will buffer data sent faster than the maximum rate by the *DataWriter*, and then only send the excess data when the user send rate drops below the maximum rate.

If **kind** is set to `DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS`, the flow controller referred to by **flow_controller_name** must exist. Otherwise, the setting will be considered inconsistent.

This QosPolicy includes the members in [Table 6.58 DDS_PublishModeQosPolicy](#). For the defaults, please refer to the API Reference HTML documentation.

Table 6.58 DDS_PublishModeQosPolicy

Type	Field Name	Description
DDS_PublishModeQosPolicyKind	kind	<p>Either:</p> <ul style="list-style-type: none"> • <code>DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS</code> • <code>DDS_SYNCHRONOUS_PUBLISH_MODE_QOS</code>
char*	flow_controller_name	<p>Name of the associated flow controller.</p> <p>There are three built-in FlowControllers:</p> <ul style="list-style-type: none"> • <code>DDS_DEFAULT_FLOW_CONTROLLER_NAME</code> • <code>DDS_FIXED_RATE_FLOW_CONTROLLER_NAME</code> • <code>DDS_ON_DEMAND_FLOW_CONTROLLER_NAME</code> <p>You may also create your own FlowControllers.</p> <p>See FlowControllers (DDS Extension) (Section 6.6 on page 422).</p>
DDS_Long	priority	<p>A positive integer designating the relative priority of the <i>DataWriter</i>, used to determine the transmission order of pending writes.</p> <p>To use publication priorities, this QosPolicy's kind must be <code>DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS</code> and the <i>DataWriter</i> must use a FlowController with a highest-priority first (HPF) scheduling_policy.</p> <p>See Prioritized DDS Samples (Section 6.6.4 on page 428).</p> <p>Note: Prioritized DDS samples are not supported when using the Java, Ada, or .NET APIs. Therefore the priority field does not exist when using these APIs.</p>

The maximum number of DDS samples that will be coalesced depends on `NDDS_Transport_Property_t::gather_send_buffer_count_max` (each DDS sample requires at least 2-4 gather-send buffers). Performance can be improved by increasing `NDDS_Transport_Property_t::gather_send_buffer_count_max`. Note that the maximum value is operating system dependent.

Connex DDS queues DDS samples until they can be sent by the asynchronous publishing thread (as determined by the corresponding FlowController).

The number of DDS samples that will be queued is determined by the [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#): when using `KEEP_LAST`, the most recent `depth` DDS samples are kept in the queue.

Once unsent DDS samples are removed from the queue, they are no longer available to the asynchronous publishing thread and will therefore never be sent.

Unless `flow_controller_name` points to one of the built-in FlowControllers, finalizing the `DataWriterQos` will also free the string pointed to by `flow_controller_name`. Therefore, you should use `DDS_String_dup()` before passing the string to `flow_controller_name`, or reset `flow_controller_name` to `NULL` before the destructing /finalizing the QoS.

Advantages of Asynchronous Publishing:

Asynchronous publishing may increase latency, but offers the following advantages:

- The `write()` call does not make any network calls and is therefore faster and more deterministic. This becomes important when the user thread is executing time-critical code.
- When data is written in bursts or when sending large data types as multiple fragments, a flow controller can throttle the send rate of the asynchronous publishing thread to avoid flooding the network.
- Asynchronously written DDS samples for the same destination will be coalesced into a single network packet which reduces bandwidth consumption.

6.5.18.1 Properties

This QoS Policy cannot be modified after the *Publisher* is created.

Since it is only for *DataWriters*, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

6.5.18.2 Related QoS Policies

- [ASYNCHRONOUS_PUBLISHER QoS Policy \(DDS Extension\) \(Section 6.4.1 on page 313\)](#)
- [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#)

6.5.18.3 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)

6.5.18.4 System Resource Considerations

See [Configuring Resource Limits for Asynchronous DataWriters \(Section 6.5.20.1 on page 406\)](#).

System resource usage depends on the settings in the corresponding FlowController (see [FlowControllers \(DDS Extension\) \(Section 6.6 on page 422\)](#)).

6.5.19 RELIABILITY QoSPolicy

This RELIABILITY QoSPolicy determines whether or not data published by a *DataWriter* will be reliably delivered by Connex DDS to matching *DataReaders*. The reliability protocol used by Connex DDS is discussed in [Reliable Communications \(Section Chapter 10 on page 629\)](#).

The reliability of a connection between a *DataWriter* and *DataReader* is entirely user configurable. It can be done on a per *DataWriter/DataReader* connection. A connection may be configured to be "best effort" which means that Connex DDS will not use any resources to monitor or guarantee that the data sent by a *DataWriter* is received by a *DataReader*.

For some use cases, such as the periodic update of sensor values to a GUI displaying the value to a person, "best effort" delivery is often good enough. It is certainly the fastest, most efficient, and least resource-intensive (CPU and network bandwidth) method of getting the newest/latest value for a topic from *DataWriters* to *DataReaders*. But there is no guarantee that the data sent will be received. It may be lost due to a variety of factors, including data loss by the physical transport such as wireless RF or even Ethernet. Packets received out of order are dropped and a [SAMPLE_LOST Status \(Section 7.3.7.7 on page 478\)](#) is generated.

However, there are data streams (topics) in which you want an absolute guarantee that all data sent by a *DataWriter* is received reliably by *DataReaders*. This means that Connex DDS must check whether or not data was received, and repair any data that was lost by resending a copy of the data as many times as it takes for the *DataReader* to receive the data.

Connex DDS uses a reliability protocol configured and tuned by these QoS policies:

- [HISTORY QoSPolicy \(Section 6.5.10 on page 376\)](#),
- [DATA_WRITER_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#),
- [DATA_READER_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#),
- [RESOURCE_LIMITS QoSPolicy \(Section 6.5.20 on page 405\)](#)

The Reliability QoS policy is simply a switch to turn on the reliability protocol for a *DataWriter/DataReader* connection. The level of reliability provided by Connex DDS is determined by the configuration of the aforementioned QoS policies.

You can configure Connex DDS to deliver ALL data in the order they were sent (also known as absolute or strict reliability). Or, as a trade-off for less memory, CPU, and network usage, you can choose a reduced level of reliability where only the last N values are guaranteed to be delivered reliably to *DataReaders* (where N is user-configurable). With the reduced level of reliability, there are no guarantees that the data sent before the last N are received. Only the last N data packets are monitored and repaired if necessary.

It includes the members in [Table 6.59 DDS_ReliabilityQoSPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.59 DDS_ReliabilityQoSPolicy

Type	Field Name	Description
DDS_ReliabilityQoSPolicyKind	kind	<p>Can be either:</p> <ul style="list-style-type: none"> • DDS_BEST_EFFORT_RELIABILITY_QOS: DDS data samples are sent once and missed samples are acceptable. • DDS_RELIABLE_RELIABILITY_QOS: Connex DDS will make sure that data sent is received and missed DDS samples are resent.
DDS_Duration_t	max_blocking_time	How long a <i>DataWriter</i> can block on a write() when the send queue is full due to unacknowledged messages. (Has no meaning for <i>DataReaders</i> .)
DDS_ReliabilityQoSPolicy-AcknowledgmentModeKind	acknowledgment_kind	<p>Kind of reliable acknowledgment.</p> <p>Only applies when <i>kind</i> is RELIABLE.</p> <p>Sets the kind of acknowledgments supported by a <i>DataWriter</i> and sent by <i>DataReader</i>.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • DDS_PROTOCOL_ACKNOWLEDGMENT_MODE • DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE • DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE <p>See Application Acknowledgment Kinds (Section 6.3.12.1 on page 289)</p>

The **kind** of RELIABILITY can be either:

- **BEST_EFFORT**

Connex DDS will send DDS data samples only once to *DataReaders*. No effort or resources are spent to track whether or not sent DDS samples are received. Minimal resources are used. This is the most deterministic method of sending data since there is no indeterministic delay that can be introduced by buffering or resending data. DDS data samples may be lost. This setting is good for periodic data.

- **RELIABLE**

Connex DDS will send DDS samples reliably to *DataReaders*—buffering sent data until they have been acknowledged as being received by *DataReaders* and resending any DDS samples that may have been lost during transport. Additional resources configured by the HISTORY and RESOURCE_LIMITS QosPolicies may be used. Extra packets will be sent on the network to query (heartbeat) and acknowledge the receipt of DDS samples by the *DataReader*. This setting is a good choice when guaranteed data delivery is required; for example, sending events or commands.

To send *large* data reliably, you will also need to set the [PUBLISH_MODE QosPolicy \(DDS Extension\) \(Section 6.5.18 on page 397\)](#) **kind** to DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS. *Large* in this context means that the data cannot be sent as a single packet by a transport (for example, data larger than 63K when using UDP/IP).

While a *DataWriter* sends data reliably, the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#) determine how many DDS samples can be stored while waiting for acknowledgements from *DataReaders*. A DDS sample that is sent reliably is entered in the *DataWriter*'s send queue awaiting acknowledgement from *DataReaders*. How many DDS samples that the *DataWriter* is allowed to store in the send queue for a data-instance depends on the **kind** of the HISTORY QoS as well as the **max_samples_per_instance** and **max_samples** parameter of the RESOURCE_LIMITS QoS.

If the HISTORY **kind** is **KEEP_LAST**, then the *DataWriter* is allowed to have the HISTORY **depth** number of DDS samples per instance of the *Topic* in the send queue. Should the number of unacknowledged DDS samples in the send queue for a data-instance reach the HISTORY **depth**, then the next DDS sample written by the *DataWriter* for the instance will overwrite the oldest DDS sample for the instance in the queue. This implies that an unacknowledged DDS sample may be overwritten and thus lost. So even if the RELIABILITY **kind** is **RELIABLE**, if the HISTORY **kind** is **KEEP_LAST**, it is possible that some data sent by the *DataWriter* will not be delivered to the *DataReader*. What is guaranteed is that if the *DataWriter* stops writing, the last *N* DDS samples that the *DataWriter* wrote will be delivered reliably; where *n* is the value of the HISTORY **depth**.

However, if the HISTORY **kind** is **KEEP_ALL**, then when the send queue is filled with acknowledged DDS samples (either due to the number of unacknowledged DDS samples for an instance reaching the RESOURCE_LIMITS **max_samples_per_instance** value or the total number of unacknowledged DDS samples have reached the size of the send queue as specified by RESOURCE_LIMITS **max_samples**),

the next `write()` operation on the *DataWriter* will block until either a DDS sample in the queue has been fully acknowledged by *DataReaders* and thus can be overwritten or a timeout of RELIABILITY `max_blocking_period` has been reached.

If there is still no space in the queue when `max_blocking_time` is reached, the `write()` call will return a failure with the error code `DDS_RETCODE_TIMEOUT`.

Thus for strict reliability—a guarantee that all DDS data samples sent by a *DataWriter* are received by *DataReaders*—you must use a RELIABILITY `kind` of `RELIABLE` and a HISTORY `kind` of `KEEP_ALL` for both the *DataWriter* and the *DataReader*.

Although you can set the RELIABILITY QosPolicy on *Topics*, its value can only be used to initialize the RELIABILITY QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

6.5.19.1 Example

This QosPolicy is used to achieve reliable communications, which is discussed in [Reliable Communications \(Section Chapter 10 on page 629\)](#) and [Enabling Reliability \(Section 10.3.1 on page 637\)](#).

6.5.19.2 Properties

This QosPolicy *cannot* be modified after the Entity has been enabled.

The *DataWriter* and *DataReader* must use compatible settings for this QosPolicy. To be compatible, the *DataWriter* and *DataReader* must use one of the valid combinations for the Reliability `kind` (see [Table 6.60 Valid Combinations of Reliability ‘kind’](#)), and one of the valid combinations for the `acknowledgment_kind` (see [Table 6.61 Valid Combinations of Reliability ‘acknowledgment_kind’](#)):

Table 6.60 Valid Combinations of Reliability ‘kind’

		DataReader requests:	
		BEST_EFFORT	RELIABLE
DataWriter offers:	BEST_EFFORT	4	incompatible
	RELIABLE	4	4

Table 6.61 Valid Combinations of Reliability 'acknowledgment_kind'

		DataReader requests:		
		PROTOCOL	APPLICATION_AUTO	APPLICATION_EXPLICIT
DataWriter offers:	PROTOCOL	4	incompatible	incompatible
	APPLICATION_AUTO	4	4	4
	APPLICATION_EXPLICIT	4	4	4

If this QosPolicy is found to be incompatible, statuses **ON_OFFERED_INCOMPATIBLE_QOS** and **ON_REQUESTED_INCOMPATIBLE_QOS** will be modified and the corresponding *Listeners* called for the *DataWriter* and *DataReader*, respectively.

There are no compatibility issues regarding the value of **max_blocking_wait**, since it does not apply to *DataReaders*.

6.5.19.3 Related QosPolicies

- [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#)
- [PUBLISH_MODE QosPolicy \(DDS Extension\) \(Section 6.5.18 on page 397\)](#)
- [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on the next page\)](#)

6.5.19.4 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.19.5 System Resource Considerations

Setting the **kind** to **RELIABLE** will cause Connex DDS to use up more resources to monitor and maintain a reliable connection between a *DataWriter* and all of its reliable *DataReaders*. This includes the use of extra CPU and network bandwidth to send and process heartbeat, ACK/NACK, and repair packets (see [Reliable Communications \(Section Chapter 10 on page 629\)](#)).

Setting **max_blocking_time** to a non-zero number may block the sending thread when the **RELIABILITY** kind is **RELIABLE**.

6.5.20 RESOURCE_LIMITS QosPolicy

For the reliability protocol (and the [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#)), this QosPolicy determines the actual maximum queue size when the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) is set to KEEP_ALL.

In general, this QosPolicy is used to limit the amount of system memory that Connex DDS can allocate. For embedded real-time systems and safety-critical systems, pre-determination of maximum memory usage is often required. In addition, dynamic memory allocation could introduce non-deterministic latencies in time-critical paths.

This QosPolicy can be set such that an entity does not dynamically allocate any more memory after its initialization phase.

It includes the members in [Table 6.62 DDS_ResourceLimitsQosPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 6.62 DDS_ResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	max_samples	Maximum number of live DDS samples that Connex DDS can store for a <i>DataWriter/DataReader</i> . This is a physical limit.
DDS_Long	max_instances	Maximum number of instances that can be managed by a <i>DataWriter/DataReader</i> . For <i>DataReaders</i> , max_instances must be \leq max_total_instances in the DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2 on page 517) . See also: Example (Section 6.5.20.3 on page 407) .
DDS_Long	max_samples_per_instance	Maximum number of DDS samples of any one instance that Connex DDS will store for a <i>DataWriter/DataReader</i> . For keyed types and <i>DataReaders</i> , this value only applies to DDS samples with an instance state of DDS_ALIVE_INSTANCE_STATE. If a keyed <i>Topic</i> is not used, then max_samples_per_instance must equal max_samples .
DDS_Long	initial_samples	Initial number of DDS samples that Connex DDS will store for a <i>DataWriter/DataReader</i> . (DDS extension)
DDS_Long	initial_instances	Initial number of instances that can be managed by a <i>DataWriter/DataReader</i> . (DDS extension)
DDS_Long	instance_hash_buckets	Number of hash buckets, which are used by Connex DDS to facilitate instance lookup. (DDS extension).

One of the most important fields is **max_samples**, which sets the size and causes memory to be allocated for the send or receive queues. For information on how this policy affects reliability, see [Tuning Queue Sizes and Other Resource Limits \(Section 10.3.2 on page 638\)](#).

When a *DataWriter* or *DataReader* is created, the **initial_instances** and **initial_samples** parameters determine the amount of memory first allocated for the those Entities. As the application executes, if more space is needed in the send/receive queues to store DDS samples or as more instances are created, then Connex DDS will automatically allocate memory until the limits of **max_instances** and **max_samples** are reached.

You may set **initial_instances = max_instances** and **initial_samples = max_samples** if you do not want Connex DDS to dynamically allocate memory after initialization.

For keyed *Topics*, the **max_samples_per_instance** field in this policy represents maximum number of DDS samples with the same key that are allowed to be stored by a *DataWriter* or *DataReader*. This is a logical limit. The hard physical limit is determined by **max_samples**. However, because the theoretical number of instances may be quite large (as set by **max_instances**), you may not want Connex DDS to allocate the total memory needed to hold the maximum number of DDS samples per instance for all possible instances (**max_samples_per_instance * max_instances**) because during normal operations, the application will never have to hold that much data for the Entity.

So it is possible that an Entity will hit the physical limit **max_samples** before it hits the **max_samples_per_instance** limit for a particular instance. However, Connex DDS must be able to store **max_samples_per_instance** for at least one instance. Therefore, **max_samples_per_instance must be <= max_samples**.

If a keyed data type is not used, there is only a single instance of the *Topic*, so **max_samples_per_instance must equal max_samples**.

Once a physical or logical limit is hit, then how Connex DDS deals with new DDS data samples being sent or received for a *DataWriter* or *DataReader* is described in the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) setting of **DDS_KEEP_ALL_HISTORY_QOS**. It is closely tied to whether or not a reliable connection is being maintained.

Although you can set the **RESOURCE_LIMITS** QosPolicy on *Topics*, its value can only be used to initialize the **RESOURCE_LIMITS** QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

6.5.20.1 Configuring Resource Limits for Asynchronous DataWriters

When using an asynchronous *Publisher*, if a call to **write()** is blocked due to a resource limit, the block will last until the timeout period expires, which will prevent others from freeing the resource. To avoid this situation, make sure that the *DomainParticipant's* **outstanding_asynchronous_sample_allocation** in the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS](#) QosPolicy (DDS Extension) (Section 8.5.4 on page 593) is always greater than the sum of all asynchronous *DataWriters'* **max_samples**.

6.5.20.2 Configuring DataWriter Instance Replacement

When the **max_instances** limit is reached, a *DataWriter* will try to make space for a new instance by replacing an existing instance according to the instance replacement kind set in **instance_replacement**. For the sake of instance replacement, an instance is considered to be unregistered, disposed, or alive. The oldest instance of the specified kind, if such an instance exists, would be replaced with the new instance. Also, all DDS samples of a replaced instance must already have been acknowledged, such that removing the instance would not deprive any existing reader from receiving them.

Since an unregistered instance is one that a *DataWriter* will not update any further, unregistered instances are replaced before any other instance kinds. This applies for all **instance_replacement** kinds; for example, the **ALIVE_THEN_DISPOSED** kind would first replace unregistered, then alive, and then disposed instances. The rest of the kinds specify one or two kinds (e.g. **DISPOSED** and **ALIVE_OR_DISPOSED**). For the single kind, if no unregistered instances are replaceable, and no instances of the specified kind are replaceable, then the instance replacement will fail. For the others specifying multiple kinds, it either specifies to look for one kind first and then another kind (e.g. **ALIVE_THEN_DISPOSED**), meaning if the first kind is found then that instance will be replaced, or it will replace either of the kinds specified (e.g. **ALIVE_OR_DISPOSED**), whichever is older as determined by the time of instance registering, writing, or disposing.

If an acknowledged instance of the specified kind is found, the *DataWriter* will reclaim its resources for the new instance. It will also invoke the *DataWriterListener*'s **on_instance_replaced()** callback (if installed) and notify the user with the handle of the replaced instance, which can then be used to retrieve the instance key from within the callback. If no replaceable instances are found, the new instance will fail to be registered; the *DataWriter* may block, if the instance registration was done in the context of a write, or it may return with an out-of-resources return code.

In addition, **replace_empty_instances** (in the [DATA_WRITER_RESOURCE_LIMITS QosPolicy \(DDS Extension\)](#) (Section 6.5.4 on page 359)) configures whether instances with no DDS samples are eligible to be replaced. If this is set, then a *DataWriter* will first try to replace empty instances, even before replacing unregistered instances.

6.5.20.3 Example

If you want to be able to store **max_samples_per_instance** for every instance, then you should set

```
max_samples >= max_instances * max_samples_per_instance
```

But if you want to save memory and you do not expect that the running application will ever reach the case where it will see **max_instances** of instances, then you may use a smaller value for **max_samples** to save memory.

In any case, there is a lower limit for **max_samples**:

```
max_samples >= max_samples_per_instance
```

If the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#)'s **kind** is set to **KEEP_LAST**, then you should set:

```
max_samples_per_instance = HISTORY.depth
```

6.5.20.4 Properties

This QosPolicy cannot be modified after the Entity is enabled.

There are no requirements that the publishing and subscribing sides use compatible values.

6.5.20.5 Related QosPolicies

- [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#)
- [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#)
- For *DataReaders*, **max_instances** must be \leq **max_total_instances** in the [DATA_READER_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 7.6.2 on page 517\)](#)

6.5.20.6 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.20.7 System Resource Considerations

Larger **initial_*** numbers will increase the initial system memory usage. Larger **max_*** numbers will increase the worst-case system memory usage.

Increasing **instance_hash_buckets** speeds up instance-lookup time but also increases memory usage.

6.5.21 SERVICE QosPolicy (DDS Extension)

The SERVICE QosPolicy is intended for use by RTI infrastructure services. User applications should not modify its value. It includes the member in [Table 6.63 DDS_ServiceQosPolicy](#).

Table 6.63 DDS_ServiceQosPolicy

Type	Field Name	Description
DDS_ServiceQosPolicyKind	kind	Kind of service associated with the entity. Possible values: DDS_NO_SERVICE_QOS, DDS_PERSISTENCE_SERVICE_QOS, DDS_QUEUEING_SERVICE_QOS, DDS_ROUTING_SERVICE_QOS, DDS_RECORDING_SERVICE_QOS, DDS_REPLAY_SERVICE_QOS, DDS_DATABASE_INTEGRATION_SERVICE_QOS

An application can determine the kind of service associated with a discovered *DataWriter* and *DataReader* by looking at the **service** field in the *PublicationBuiltinTopicData* and *SubscriptionBuiltinTopicData* structures (see [Chapter 16: Built-In Topics](#)).

6.5.21.1 Properties

This QosPolicy cannot be modified after the Entity is enabled.

There are no requirements that the publishing and subscribing sides use compatible values.

6.5.21.2 Related QosPolicies

None

6.5.21.3 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3\)](#)

6.5.21.4 System Resource Considerations

None.

6.5.22 TRANSPORT_PRIORITY QosPolicy

The TRANSPORT_PRIORITY QosPolicy is optional and only partially supported on certain OSs and transports by RTI. However, its intention is to allow you to specify on a per-*DataWriter* or per-*DataReader* basis that the data sent by a *DataWriter* or *DataReader* is of a different priority.

DDS does not specify how a DDS implementation shall treat data of different priorities. It is often difficult or impossible for DDS implementations to treat data of higher priority differently than data of lower priority, especially when data is being sent (delivered to a physical transport) directly by the thread that called *DataWriter's* **write()** operation. Also, many physical network transports themselves do not have an end-user controllable level of data packet priority.

In Connex DDS, for the UDPv4 built-in transport, the value set in the `TRANSPORT_PRIORITY` QosPolicy is used in a `setsockopt` call to set the TOS (type of service) bits of the IPv4 header for datagrams sent by a *DataWriter* or *DataReader*. It is platform dependent on how and whether or not the `setsockopt` has an effect. On some platforms such as Windows and Linux, external permissions must be given to the user application in order to set the TOS bits.

It is incorrect to assume that using the `TRANSPORT_PRIORITY` QosPolicy will have any effect at all on the end-to-end delivery of data between a *DataWriter* and *DataReader*. All network elements such as switches and routers must have the capability and be enabled to actually use the TOS bits to treat higher-priority packets differently. Thus the ability to use the `TRANSPORT_PRIORITY` QosPolicy must be designed and configured at a system level; just turning it on in an application may have no effect at all.

It includes the member in [Table 6.64 DDS_TransportPriorityQosPolicy](#). For the default and valid range, please refer to the API Reference HTML documentation.

Table 6.64 DDS_TransportPriorityQosPolicy

Type	Field Name	Description
DDS_Long	value	Hint as to how to set the priority.

Connex DDS will propagate the **value** set on a per-*DataWriter* or per-*DataReader* basis to the transport when the *DataWriter* publishes data. It is up to the implementation of the transport to do something with the **value**, if anything.

You can set the `TRANSPORT_PRIORITY` QosPolicy on a *Topic* and use its **value** to initialize the `TRANSPORT_PRIORITY` QosPolicies of *DataWriters* and *DataReaders*. The `TRANSPORT_PRIORITY` QosPolicy of a *Topic* does not directly affect the operation of Connex DDS, see [Setting Topic QosPolicies \(Section 5.1.3 on page 204\)](#).

6.5.22.1 Example

Should Connex DDS be configured with a transport that can use and will honor the concept of a prioritized message, then you would be able to create a *DataWriter* of a *Topic* whose DDS data samples, when published, will be sent at a higher priority than other *DataWriters* that use the same transport.

6.5.22.2 Properties

This QosPolicy cannot be modified after the entity is created.

6.5.22.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

6.5.22.4 Applicable Entities

- [Topics \(Section 5.1 on page 200\)](#)
- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.22.5 System Resource Considerations

The use of this policy does not significantly impact the use of resources. However, if a transport is implemented to use the value set by this policy, then there may be transport-specific issues regarding the resources that the transport implementation itself uses.

6.5.23 TRANSPORT_SELECTION QosPolicy (DDS Extension)

The TRANSPORT_SELECTION QosPolicy allows you to select the transports that have been installed with the *DomainParticipant* to be used by the *DataWriter* or *DataReader*.

An application may be simultaneously connected to many different physical transports, e.g., Ethernet, Infiniband, shared memory, VME backplane, and wireless. By default, the middleware will use up to 4 transports to deliver data from a *DataWriter* to a *DataReader*.

This QosPolicy can be used to both limit and control which of the application's available transports may be used by a *DataWriter* to send data or by a *DataReader* to receive data.

It includes the member in [Table 6.65 DDS_TransportSelectionQosPolicy](#). For more information, please refer to the API Reference HTML documentation.

Table 6.65 DDS_TransportSelectionQosPolicy

Type	Field Name	Description
DDS_StringSeq	enabled_transports	A sequence of aliases for the transports that may be used by the <i>DataWriter</i> or <i>DataReader</i> .

Connex DDS allows you to configure the transports that it uses to send and receive messages. A number of built-in transports, such as UDPv4 and shared memory, are available as well as custom ones that you may implement and install. Each transport will be installed in the *DomainParticipant* with one or more *aliases*.

To enable a *DataWriter* or *DataReader* to use a particular transport, add the *alias* to the **enabled_transports** sequence of this QosPolicy. An empty sequence is a special case, and indicates that all transports installed in the *DomainParticipant* can be used by the *DataWriter* or *DataReader*.

For more information on configuring and installing transports, please see the API Reference HTML documentation (from the **Modules** page, select **RTI DDS API Reference, Pluggable Transports**).

6.5.23.1 Example

Suppose a *DomainParticipant* has both UDPv4 and shared memory transports installed. If you want a particular *DataWriter* to publish its data only over shared memory, then you should use this QosPolicy to specify that restriction.

6.5.23.2 Properties

This QosPolicy cannot be modified after the *Entity* is created.

It can be set differently for the *DataWriter* and the *DataReader*.

6.5.23.3 Related QosPolicies

- [TRANSPORT_UNICAST QosPolicy \(DDS Extension\) \(Section 6.5.24 below\)](#)
- [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#)
- [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\) \(Section 8.5.7 on page 606\)](#)

6.5.23.4 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)

6.5.23.5 System Resource Considerations

By restricting *DataWriters* from sending or *DataReaders* from receiving over certain transports, you may decrease the load on those transports.

6.5.24 TRANSPORT_UNICAST QosPolicy (DDS Extension)

The TRANSPORT_UNICAST QosPolicy allows you to specify unicast network addresses to be used by *DomainParticipant*, *DataWriters* and *DataReaders* for receiving messages.

Connex DDS may send data to a variety of *Entities*, not just *DataReaders*. *DomainParticipants* receive messages to support the discovery process discussed in [Discovery \(Section Chapter 14 on page 709\)](#). *DataWriters* may receive ACK/NACK messages to support the reliable protocol discussed in [Reliable Communications \(Section Chapter 10 on page 629\)](#).

During discovery, each *Entity* announces to remote applications a list of (up to 4) unicast addresses to which the remote application should send data (either user-data packets or reliable protocol meta-data such as ACK/NACK and Heartbeats).

By default, the list of addresses is populated automatically with values obtained from the enabled transport plugins allowed to be used by the Entity (see the [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\)](#) (Section 8.5.7 on page 606) and [TRANSPORT_SELECTION QosPolicy \(DDS Extension\)](#) (Section 6.5.23 on page 411)). Also, the associated ports are automatically determined (see [Inbound Ports for User Traffic](#) (Section 14.5.2 on page 740)).

Use `TRANSPORT_UNICAST QosPolicy` to manually set the receive address list for an Entity. You may optionally set a port to use a non-default receive port as well. Only the first 4 addresses will be used. Connext DDS will create a receive thread for every unique port number that it encounters (on a per transport basis).

The QosPolicy structure includes the members in [Table 6.66 DDS_TransportUnicastQosPolicy](#). For more information and default values, please refer to the API Reference HTML documentation.

Table 6.66 DDS_TransportUnicastQosPolicy

Type	Field Name	Description
DDS_TransportUnicastSettingsSeq (see Table 6.67 DDS_TransportUnicastSettings_t)	value	A sequence of up to 4 unicast settings that should be used by remote entities to address messages to be sent to this <i>Entity</i> .

Table 6.67 DDS_TransportUnicastSettings_t

Type	Field Name	Description
DDS_StringSeq	transports	A sequence of transport aliases that specifies which transports should be used to receive unicast messages for this <i>Entity</i> .
DDS_Long	receive_port	The port that should be used in the addressing of unicast messages destined for this <i>Entity</i> . A value of 0 will cause Connext DDS to use a default port number based on <i>domain</i> and participant ids. See Ports Used for Discovery (Section 14.5 on page 738).

A message sent to a unicast address will be received by a single node on the network (as opposed to a multicast address where a single message may be received by multiple nodes). This policy sets the unicast addresses and ports that remote entities should use when sending messages to the Entity on which the `TRANSPORT_UNICAST QosPolicy` is set.

Up to four “return” unicast addresses may be configured for an *Entity*. Instead of specifying addresses directly, you use the **transports** field of the **DDS_TransportUnicastSetting_t** to select the transports (using their aliases) on which remote entities should send messages destined for this Entity. The addresses of the selected transports will be the “return” addresses. See the API Reference HTML documentation about

configuring transports and aliases (from the **Modules** page, select **RTI Connex DDS API Reference, Pluggable Transports**).

Note, a single transport may have more than one unicast address. For example, if a node has multiple network interface cards (NICs), then the UDPv4 transport will have an address for each NIC. When using the TRANSPORT_UNICAST QoSPolicy to set the return addresses, a single **value** for the **DDS_TransportUnicastSettingsSeq** may provide more than the four return addresses that Connex DDS currently uses.

Whether or not you are able to configure the network interfaces that are allowed to be used by a transport is up to the implementation of the transport. For the built-in UDPv4 transport, you may restrict an instance of the transport to use a subset of the available network interfaces. See the API Reference HTML documentation for the built-in UDPv4 transport for more information.

For a *DomainParticipant*, this QoS policy sets the default list of addresses used by other applications to send user data for local *DataReaders*.

For a reliable *DataWriter*, if set, the other applications will use the specified list of addresses to send reliable protocol packets (ACKS/NACKS) on the behalf of reliable *DataReaders*. Otherwise, if not set, the other applications will use the addresses set by the *DomainParticipant*.

For a *DataReader*, if set, then other applications will use the specified list of addresses to send user data (and reliable protocol packets for reliable *DataReaders*). Otherwise, if not set, the other applications will use the addresses set by the *DomainParticipant*.

For a *DataReader*, if the port number specified by this QoS is the same as a port number specified by a TRANSPORT_MULTICAST QoS, then the transport may choose to process data received both via multicast and unicast with a single thread. Whether or not a transport must use different threads to process data received via multicast or unicast for the same port number depends on the implementation of the transport.

To use this QoSPolicy, you also need to specify a port number. A port number of 0 will cause Connex DDS to automatically use a default value. As explained in [Ports Used for Discovery \(Section 14.5 on page 738\)](#), the default port number for unicast addresses is based on the domain and participant IDs. Should you choose to use a different port number, then for every unique port number used by Entities in your application, depending on the transport, Connex DDS may create a thread to process messages received for that port on that transport. See [Connex DDS Threading Model \(Section Chapter 19 on page 837\)](#) for more about threads.

Threads are created on a per-transport basis, so if this QoSPolicy specifies multiple **transports** for a **receive_port**, then a thread may be created for each transport for that unique port. Some transports may be able to share a single thread for different ports, others can not. Different *Entities* can share the same port number, and thus, the same thread will process all of the data for all of the *Entities* sharing the same port number for a transport.

Note: If a *DataWriter* is using the [MULTI_CHANNEL QosPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386), the unicast addresses specified in the `TRANSPORT_UNICAST` QosPolicy are ignored by that *DataWriter*. The *DataWriter* will not publish DDS samples on those locators.

6.5.24.1 Example

You may use this QosPolicy to restrict an *Entity* from receiving data through a particular transport. For example, on a multi-NIC (network interface card) system, you may install different transports for different NICs. Then you can balance the network load between network cards by using different values for the `TRANSPORT_UNICAST` QosPolicy for different *DataReaders*. Thus some *DataReaders* will receive their data from one NIC and other *DataReaders* will receive their data from another.

6.5.24.2 Properties

This QosPolicy cannot be modified after the Entity is created.

It can be set differently for the *DomainParticipant*, the *DataWriter* and the *DataReader*.

6.5.24.3 Related QosPolicies

- [MULTI_CHANNEL QosPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386)
- [TRANSPORT_SELECTION QosPolicy \(DDS Extension\)](#) (Section 6.5.23 on page 411)
- [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\)](#) (Section 7.6.5 on page 529)
- [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\)](#) (Section 8.5.7 on page 606)

6.5.24.4 Applicable Entities

- [DomainParticipants](#) (Section 8.3 on page 547)
- [DataWriters](#) (Section 6.3 on page 261)
- [DataReaders](#) (Section 7.3 on page 459)

6.5.24.5 System Resource Considerations

Because this QosPolicy changes the transports on which messages are received for different Entities, the bandwidth used on the different transports may be affected.

Depending on the implementation of a transport, Connex DDS may need to create threads to receive and process data on a unique-port-number basis. Some transports can share the same thread to process data received for different ports; others like UDPv4 must have different threads for different ports. In addition, if the same port is used for both unicast and multicast, the transport implementation will determine whether or not the same thread can be used to process both unicast and multicast data. For UDPv4, only one thread is needed per port— independent of whether the data was received via unicast or multicast data. See [Receive Threads](#) (Section 19.3 on page 839) for more information.

6.5.25 TYPESUPPORT QosPolicy (DDS Extension)

This policy can be used to modify the code generated by *RTI Code Generator* so that the [de]serialization routines act differently depending on the information passed in via the object pointer. This policy also determines if padding bytes are set to zero during serialization.

It includes the members in [Table 6.68 DDS_TypeSupportQosPolicy](#).

Table 6.68 DDS_TypeSupportQosPolicy

Type	Field Name	Description
void *	plugin_data	Value to pass into the type plug-in's serialization/deserialization function. See Note below.
DDS_CdrPaddingKind	cdr_padding_kind	<p>Determines whether or not the padding bytes will be set to zero during CDR serialization.</p> <p>For a <i>DomainParticipant</i>: Configures how padding bytes are set when serializing data for the builtin topic <i>DataWriters</i> and <i>DataReaders</i>.</p> <p>For <i>DataWriters</i> and <i>DataReaders</i>: Configures how padding bytes are set when serializing data for that entity.</p> <p>May be:</p> <ul style="list-style-type: none"> • ZERO_CDR_PADDING (Padding bytes will be set to zero during CDR serialization) • NOT_SET_CDR_PADDING (Padding bytes will not be set to any value during CDR serialization) • AUTO_CDR_PADDING (For a <i>DomainParticipant</i>, the default behavior is NOT_SET_CDR_PADDING. For a <i>DataWriter</i> or <i>DataReader</i>, the behavior is to inherit the value from the <i>DomainParticipant</i>.)

Note: RTI generally recommends that you treat generated source files as compiler outputs (analogous to object files) and that you do not modify them. RTI cannot support user changes to generated source files. Furthermore, such changes would make upgrading to newer versions of Connex DDS more difficult, as this generated code is considered to be a part of the middleware implementation and consequently does change from version to version. *The plugin_data field in this QoS policy should be considered a back door, only to be used after careful design consideration, testing, and consultation with your RTI representative.*

6.5.25.1 Properties

This QoS policy may be modified after the *DataWriter* or *DataReader* is enabled.

It can be set differently for the *DataWriter* and *DataReader*.

6.5.25.2 Related QoS Policies

None.

6.5.25.3 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)
- [DomainParticipants \(Section 8.3 on page 547\)](#)

6.5.25.4 System Resource Considerations

None.

6.5.26 USER_DATA QoSPolicy

This QoSPolicy provides an area where your application can store additional information related to a *DomainParticipant*, *DataWriter*, or *DataReader*. This information is passed between applications during discovery (see [Discovery \(Section Chapter 14 on page 709\)](#)) using built-in-topics (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)). How this information is used will be up to user code. Connex DDS does not do anything with the information stored as USER_DATA except to pass it to other applications.

Use cases are usually for application-to-application identification, authentication, authorization, and encryption purposes. For example, applications can use Group or User Data to send security certificates to each other for RSA-type security.

The value of the USER_DATA QoSPolicy is sent to remote applications when they are first discovered, as well as when the *DomainParticipant*, *DataWriter* or *DataReader*'s `set_qos()` methods are called after changing the value of the USER_DATA. User code can set listeners on the built-in *DataReaders* of the built-in *Topics* used by Connex DDS to propagate discovery information. Methods in the built-in topic listeners will be called whenever new *DomainParticipants*, *DataReaders*, and *DataWriters* are found. Within the user callback, you will have access to the USER_DATA that was set for the associated *Entity*.

Currently, USER_DATA of the associated *Entity* is only propagated with the information that declares a *DomainParticipant*, *DataWriter* or *DataReader*. Thus, you will need to access the value of USER_DATA through `DDS_ParticipantBuiltinTopicData`, `DDS_PublicationBuiltinTopicData` or `DDS_SubscriptionBuiltinTopicData` (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)).

The structure for the USER_DATA QoSPolicy includes just one field, as seen in [Table 6.69 DDS_UserDataQoSPolicy](#). The field is a sequence of octets that translates to a contiguous buffer of bytes whose contents and length is set by the user. The maximum size for the data are set in the `DOMAIN_PARTICIPANT_RESOURCE_LIMITS` QoSPolicy (DDS Extension) ([Section 8.5.4 on page 593](#)).

Table 6.69 DDS_UserDataQosPolicy

Type	Field Name	Description
DDS_OctetSeq	value	Default: empty

This policy is similar to the [GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#) and [TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#) that apply to other types of Entities.

6.5.26.1 Example

One possible use of USER_DATA is to pass some credential or certificate that your subscriber application can use to accept or reject communication with the *DataWriters* (or vice versa, where the publisher application can validate the permission of *DataReaders* to receive its data). Using the same method, an application (*DomainParticipant*) can accept or reject all connections from another application. The value of the USER_DATA of the *DomainParticipant* is propagated in the ‘user_data’ field of the **DDS_ParticipantBuiltinTopicData** that is sent with the declaration of each *DomainParticipant*. Similarly, the value of the USER_DATA of the *DataWriter* is propagated in the ‘user_data’ field of the **DDS_PublicationBuiltinTopicData** that is sent with the declaration of each *DataWriter*, and the value of the USER_DATA of the *DataReader* is propagated in the ‘user_data’ field of the **DDS_SubscriptionBuiltinTopicData** that is sent with the declaration of each *DataReader*.

When Connex DDS discovers a *DomainParticipant/DataWriter/DataReader*, the application can be notified of the discovery of the new entity and retrieve information about the *Entity*’s QoS by reading the **DCPSParticipant**, **DCPSPublication** or **DCPSSubscription** built-in topics (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)). The user application can then examine the USER_DATA field in the built-in *Topic* and decide whether or not the remote *Entity* should be allowed to communicate with the local *Entity*. If communication is not allowed, the application can use the *DomainParticipant*’s **ignore_participant()**, **ignore_publication()** or **ignore_subscription()** operation to reject the newly discovered remote entity as one with which the application allows Connex DDS to communicate. See [Built-in DataReaders \(Section 16.2 on page 773\)](#) for an example of how to do this.

6.5.26.2 Properties

This QosPolicy can be modified at any time. A change in the QosPolicy will cause Connex DDS to send packets containing the new USER_DATA to all of the other applications in the DDS domain.

It can be set differently on the publishing and subscribing sides.

6.5.26.3 Related QosPolicies

- [TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#)
- [GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)

- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#)

6.5.26.4 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)
- [DataReaders \(Section 7.3 on page 459\)](#)
- [DomainParticipants \(Section 8.3 on page 547\)](#)

6.5.26.5 System Resource Considerations

The maximum size of the USER_DATA is set in the **participant_user_data_max_length**, **writer_user_data_max_length**, and **reader_user_data_max_length** fields of the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#). Because Connex DDS will allocate memory based on this value, you should only increase this value if you need to. If your system does not use USER_DATA, then you can set this value to 0 to save memory. Setting the value of the USER_DATA QosPolicy to hold data longer than the value set in the **[participant,writer,reader]_user_data_max_length** field will result in failure and an **INCONSISTENT_QOS_POLICY** return code.

However, should you decide to change the maximum size of USER_DATA, you *must* make certain that all applications in the DDS domain have changed the value of **[participant,writer,reader]_user_data_max_length** to be the same. If two applications have different limits on the size of USER_DATA, and one application sets the USER_DATA QosPolicy to hold data that is greater than the maximum size set by another application, then the *DataWriters* and *DataReaders* between the two applications will *not* connect. The *DomainParticipants* may also reject connections from each other entirely. This is also true for the GROUP_DATA ([GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)) and TOPIC_DATA ([TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#)) QosPolicies.

6.5.27 WRITER_DATA_LIFECYCLE QoS Policy

This QoS policy controls how a *DataWriter* handles the lifecycle of the instances (keys) that the *DataWriter* is registered to manage. This QoS policy includes the members in [Table 6.70 DDS_WriterDataLifecycleQosPolicy](#).

Table 6.70 DDS_WriterDataLifecycleQosPolicy

Type	Field Name	Description
DDS_Boolean	autodispose_unregistered_instances	RTI_TRUE (default): Instance is disposed when unregistered. RTI_FALSE: Instance is not disposed when unregistered.

Table 6.70 DDS_WriterDataLifecycleQoSPolicy

Type	Field Name	Description
struct DDS_ Duration_ t	autopurge_ unregistered_ instance_ delay	<p>Determines how long the <i>DataWriter</i> will maintain information regarding an instance that has been unregistered.</p> <p>By default, the <i>DataWriter</i> resources associated with an instance (e.g., the space needed to remember the Instance Key or KeyHash) are released lazily. This means the resources are only reclaimed when the space is needed for another instance because <code>max_instances</code> (Section on page 405) (see <code>RESOURCE_LIMITS QoSPolicy</code> (Section 6.5.20 on page 405)) is exceeded. This behavior can be changed by setting <code>autopurge_unregistered_instance_delay</code> to a value other than <code>INFINITE</code>.</p> <p>After this time elapses, the <i>DataWriter</i> will purge all internal information regarding the instance, including historical DDS samples even if <code>max_instances</code> (Section on page 405) has not been reached.</p>

You may use the *DataWriter*'s `unregister()` operation ([Registering and Unregistering Instances \(Section 6.3.14.1 on page 297\)](#)) to indicate that the *DataWriter* no longer wants to send data for a Topic. This QoS controls whether or not Connex DDS automatically also calls `dispose()` ([Disposing of Data \(Section 6.3.14.2 on page 299\)](#)) on the behalf of the *DataWriter* for the data.

Unregistering vs. Disposing:

- When an instance is unregistered, it means *this particular DataWriter* has no more information/data on this instance.
- When an instance is disposed, it means the instance is "dead"—there will no more information/data from *any DataWriter* on this instance.

The behavior controlled by this QoS applies on a per instance (key) basis for keyed Topics, so when a *DataWriter* unregisters an instance, Connex DDS also automatically disposes that instance. This is the default behavior since `autodispose_unregistered_instances` defaults to TRUE.

Use Cases for Unregistering without Disposing:

There are situations in which you may want to set `autodispose_unregistered_instances` to FALSE, so that unregistering will not automatically dispose the instance. For example:

- In many cases where the ownership of a Topic is EXCLUSIVE (see the [OWNERSHIP QoSPolicy \(Section 6.5.15 on page 389\)](#)), *DataWriters* may want to relinquish ownership of a particular instance of the Topic to allow other *DataWriters* to send updates for the value of that instance. In this case, you may want a *DataWriter* to just unregister an instance—without disposing it (since there are other writers). Unregistering an instance implies that the *DataWriter* no longer owns that instance, but it is a stronger statement to say that instance no longer exists.

- User applications may be coded to trigger on the disposal of instances, thus the ability to unregister without disposing may be useful to properly maintain the semantic of disposal.

When you delete a *DataWriter* ([Creating DataWriters \(Section 6.3.1 on page 266\)](#)), all of the instances managed by the *DataWriter* are automatically unregistered. Therefore, this QoS policy determines whether or not all of the instances are disposed when the *DataWriter* is deleted when you call one of these operations:

- *Publisher's delete_datawriter()* (see [Creating DataWriters \(Section 6.3.1 on page 266\)](#))
- *Publisher's delete_contained_entities()* (see [Deleting Contained DataWriters \(Section 6.2.3.1 on page 251\)](#))
- *DomainParticipant's delete_contained_entities()* (see [Deleting Contained Entities \(Section 8.3.3 on page 559\)](#))

When **autodispose_unregistered_instances** is TRUE, the middleware will clean up all the resources associated with an unregistered instance (most notably, the DDS sample history of non-volatile *DataWriters*) when all the instance's DDS samples have been acknowledged by all its live *DataReaders*, including the DDS sample that indicates the unregistration. By default, **autopurge_unregistered_instances_delay** is disabled (the delay is INFINITE). If the delay is set to zero, the *DataWriter* will clean up as soon as all the DDS samples are acknowledged after the call to **unregister()**. A non-zero value for the delay can be useful in two ways:

- To keep the historical DDS samples for late-joiners for a period of time.
- In the context of discovery, if the applications temporarily lose the connection before the unregistration (which represents the remote entity destruction), to provide the DDS samples that indicate the dispose and unregister actions once the connection is reestablished.

*This delay can also be set for discovery data through these fields in the **DISCOVERY_CONFIG QoSPolicy (DDS Extension)** (Section 8.5.3 on page 585):*

- **publication_writer_data_lifecycle.autopurge_unregistered_instances_delay**
- **subscription_writer_data_lifecycle.autopurge_unregistered_instances_delay**

6.5.27.1 Properties

It does not apply to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

This QoS policy may be modified after the *DataWriter* is enabled.

6.5.27.2 Related QoS Policies

- None.

6.5.27.3 Applicable Entities

- [DataWriters \(Section 6.3 on page 261\)](#)

6.5.27.4 System Resource Considerations

None.

6.6 FlowControllers (DDS Extension)

This section does not apply when using the separate add-on product, *Ada Language Support*, which does not support FlowControllers.

A FlowController is the object responsible for shaping the network traffic by determining when attached asynchronous *DataWriters* are allowed to write data.

You can use one of the built-in FlowControllers (and optionally modify their properties), create a custom FlowController by using the *DomainParticipant's* `create_flowcontroller()` operation (see [Creating and Deleting FlowControllers \(Section 6.6.6 on page 433\)](#)), or create a custom FlowController by using the *DomainParticipant's* `PROPERTY QoSPolicy (DDS Extension)` (Section 6.5.17 on page 394); see [Creating and Configuring Custom FlowControllers with Property QoS \(Section 6.6.5 on page 431\)](#).

To use a FlowController, you provide its name in the *DataWriter's* `PUBLISH_MODE QoSPolicy (DDS Extension)` (Section 6.5.18 on page 397).

- **DDS_DEFAULT_FLOW_CONTROLLER_NAME**

By default, flow control is disabled. That is, the built-in `DDS_DEFAULT_FLOW_CONTROLLER_NAME` flow controller does not apply any flow control. Instead, it allows data to be sent asynchronously as soon as it is written by the *DataWriter*.

- **DDS_FIXED_RATE_FLOW_CONTROLLER_NAME**

The `FIXED_RATE` flow controller shapes the network traffic by allowing data to be sent only once every second. Any accumulated DDS samples destined for the same destination are coalesced into as few network packets as possible.

- **DDS_ON_DEMAND_FLOW_CONTROLLER_NAME**

The `ON_DEMAND` flow controller allows data to be sent only when you call the FlowController's

trigger_flow() operation. With each trigger, all accumulated data since the previous trigger is sent (across all *Publishers* or *DataWriters*). In other words, the network traffic shape is fully controlled by the user. Any accumulated DDS samples destined for the same destination are coalesced into as few network packets as possible.

This external trigger source is ideal for users who want to implement some form of closed-loop flow control or who want to only put data on the wire every so many DDS samples (e.g., with the number of DDS samples based on `NDDS_Transport_Property_t`'s **gather_send_buffer_count_max**).

The default property settings for the built-in FlowControllers are described in the API Reference HTML documentation.

DDS samples written by an asynchronous *DataWriter* are not sent in the context of the **write()** call. Instead, Connex DDS puts the DDS samples in a queue for future processing. The FlowController associated with each asynchronous *DataWriter* determines when the DDS samples are actually sent.

Each FlowController maintains a separate FIFO queue for each unique destination (remote application). DDS samples written by asynchronous *DataWriters* associated with the FlowController are placed in the queues that correspond to the intended destinations of the DDS sample.

When tokens become available, a FlowController must decide which queue(s) to grant tokens first. This is determined by the FlowController's **scheduling_policy** property (see [Table 6.71 DDS_FlowControllerProperty_t](#)). Once a queue has been granted tokens, it is serviced by the asynchronous publishing thread. The queued up DDS samples will be coalesced and sent to the corresponding destination. The number of DDS samples sent depends on the data size and the number of tokens granted.

[Table 6.71 DDS_FlowControllerProperty_t](#) lists the properties for a FlowController.

Table 6.71 DDS_FlowControllerProperty_t

Type	Field Name	Description
DDS_FlowControllerSchedulingPolicy	scheduling_policy	Round robin, earliest deadline first, or highest priority first. See Flow Controller Scheduling Policies (Section 6.6.1 on the facing page) .
DDS_FlowControllerTokenBucketProperty_t	token_bucket	See Token Bucket Properties (Section 6.6.3 on page 426) .

[Table 6.72 FlowController Operations](#) lists the operations available for a FlowController.

Table 6.72 FlowController Operations

Operation	Description	Reference
get_property	Get and Set the FlowController properties.	Getting/Setting Properties for a Specific FlowController (Section 6.6.8 on page 435)
set_property		
trigger_flow	Provides an external trigger to the FlowController.	Adding an External Trigger (Section 6.6.9 on page 435)
get_name	Returns the name of the FlowController.	Other FlowController Operations (Section 6.6.10 on page 435)
get_participant	Returns the <i>DomainParticipant</i> to which the FlowController belongs.	

6.6.1 Flow Controller Scheduling Policies

- **Round Robin**

(DDS_RR_FLOW_CONTROLLER_SCHED_POLICY) Perform flow control in a round-robin (RR) fashion.

Whenever tokens become available, the FlowController distributes the tokens uniformly across all of its (non-empty) destination queues. No destinations are prioritized. Instead, all destinations are treated equally and are serviced in a round-robin fashion.

- **Earliest Deadline First**

(DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY) Perform flow control in an earliest-deadline-first (EDF) fashion.

A DDS sample's deadline is determined by the time it was written plus the latency budget of the *DataWriter* at the time of the write call (as specified in the `DDS_LatencyBudgetQosPolicy`). The relative priority of a flow controller's destination queue is determined by the earliest deadline across all DDS samples it contains.

When tokens become available, the FlowController distributes tokens to the destination queues in order of their priority. In other words, the queue containing the DDS sample with the earliest deadline is serviced first. The number of tokens granted equals the number of tokens required to send the first DDS sample in the queue. Note that the priority of a queue may change as DDS samples are sent (i.e., removed from the queue). If a DDS sample must be sent to multiple destinations or two DDS samples have an equal deadline value, the corresponding destination queues are serviced in a round-robin fashion.

With the default **duration** of 0 in the `LatencyBudgetQosPolicy`, using an `EDF_FLOW_CONTROLLER_SCHED_POLICY` `FlowController` preserves the order in which you call `write()` across the `DataWriters` associated with the `FlowController`.

Since the `LatencyBudgetQosPolicy` is mutable, a DDS sample written second may contain an earlier deadline than the DDS sample written first if the `DDS_LatencyBudgetQosPolicy`'s **duration** is sufficiently decreased in between writing the two DDS samples. In that case, if the first DDS sample is not yet written (still in queue waiting for its turn), it inherits the priority corresponding to the (earlier) deadline from the second DDS sample.

In other words, the priority of a destination queue is always determined by the earliest deadline among all DDS samples contained in the queue. This priority inheritance approach is required in order to both honor the updated **duration** and to adhere to the `DataWriter` in-order data delivery guarantee.

- **Highest Priority First**

(`DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY`) Perform flow control in an highest-priority-first (HPF) fashion.

Note: Prioritized DDS samples are not supported when using the Java, Ada, or .NET APIs. Therefore the Highest Priority First scheduling policy is not supported when using these APIs.

The next destination queue to service is determined by the publication priority of the `DataWriter`, the channel of a multi-channel `DataWriter`, or individual DDS sample.

The relative priority of a flow controller's destination queue is determined by the highest publication priority of all the DDS samples it contains.

When tokens become available, the `FlowController` distributes tokens to the destination queues in order of their publication priority. The queue containing the DDS sample with the highest publication priority is serviced first. The number of tokens granted equals the number of tokens required to send the first DDS sample in the queue. Note that a queue's priority may change as DDS samples are sent (i.e., as they are removed from the queue). If a DDS sample must be sent to multiple destinations or two DDS samples have the same publication priority, the corresponding destination queues are serviced in a round-robin fashion.

This priority inheritance approach is required to both honor the designated publication priority and adhere to the `DataWriter`'s in-order data delivery guarantee.

See also: [Prioritized DDS Samples \(Section 6.6.4 on page 428\)](#).

6.6.2 Managing Fast DataWriters When Using a FlowController

If a *DataWriter* is writing DDS samples faster than its attached FlowController can throttle, Connex DDS may drop DDS samples on the writer's side. This happens because the DDS samples may be removed from the queue before the asynchronous publisher's thread has a chance to send them. To work around this problem, either:

- Use reliable communication to block the **write()** call and thereby throttle your application.
- Do not allow the queue to fill up in the first place.

The queue should be sized large enough to handle expected write bursts, so that no DDS samples are dropped. Then in steady state, the FlowController will smooth out these bursts and the queue will ideally have only one entry.

6.6.3 Token Bucket Properties

FlowControllers use a token-bucket approach for open-loop network flow control. The flow control characteristics are determined by the token bucket properties. The properties are listed in [Table 6.73 DDS_FlowControllerTokenBucketProperty_t](#); see the API Reference HTML documentation for their defaults and valid ranges.

Table 6.73 DDS_FlowControllerTokenBucketProperty_t

Type	Field Name	Description
DDS_Long	max_tokens	Maximum number of tokens than can accumulate in the token bucket. See max_tokens (Section 6.6.3.1 on the next page) .
DDS_Long	tokens_added_per_period	The number of tokens added to the token bucket per specified period. See tokens_added_per_period (Section 6.6.3.2 on the next page) .
DDS_Long	tokens_leaked_per_period	The number of tokens removed from the token bucket per specified period. See tokens_leaked_per_period (Section 6.6.3.3 on the next page) .
DDS_Duration_t	period	Period for adding tokens to and removing tokens from the bucket. See period (Section 6.6.3.4 on the next page) .
DDS_Long	bytes_per_token	Maximum number of bytes allowed to send for each token available. See bytes_per_token (Section 6.6.3.5 on page 428) .

Asynchronously published DDS samples are queued up and transmitted based on the token bucket flow control scheme. The token bucket contains tokens, each of which represents a number of bytes. DDS samples can be sent only when there are sufficient tokens in the bucket. As DDS samples are sent, tokens are consumed. The number of tokens consumed is proportional to the size of the data being sent. Tokens are replenished on a periodic basis.

The rate at which tokens become available and other token bucket properties determine the network traffic flow.

Note that if the same DDS sample must be sent to multiple destinations, separate tokens are required for each destination. Only when multiple DDS samples are destined to the same destination will they be coalesced and sent using the same token(s). In other words, each token can only contribute to a single network packet.

6.6.3.1 max_tokens

The maximum number of tokens in the bucket will never exceed this value. Any excess tokens are discarded. This property value, combined with **bytes_per_token**, determines the maximum allowable data burst.

Use `DDS_LENGTH_UNLIMITED` to allow accumulation of an unlimited amount of tokens (and therefore potentially an unlimited burst size).

6.6.3.2 tokens_added_per_period

A FlowController transmits data only when tokens are available. Tokens are periodically replenished. This field determines the number of tokens added to the token bucket with each periodic replenishment.

Available tokens are distributed to associated *DataWriters* based on the **scheduling_policy**. Use `DDS_LENGTH_UNLIMITED` to add the maximum number of tokens allowed by **max_tokens**.

6.6.3.3 tokens_leaked_per_period

When tokens are replenished and there are sufficient tokens to send all DDS samples in the queue, this property determines whether any or all of the leftover tokens remain in the bucket.

Use `DDS_LENGTH_UNLIMITED` to remove all excess tokens from the token bucket once all DDS samples have been sent. In other words, no token accumulation is allowed. When new DDS samples are written after tokens were purged, the earliest point in time at which they can be sent is at the next periodic replenishment.

6.6.3.4 period

This field determines the period by which tokens are added or removed from the token bucket.

The special value `DDS_DURATION_INFINITE` can be used to create an on-demand FlowController, for which tokens are no longer replenished periodically. Instead, tokens must be added explicitly by calling the FlowController's **trigger_flow()** operation. This external trigger adds **tokens_added_per_period** tokens each time it is called (subject to the other property settings).

Once **period** is set to `DDS_DURATION_INFINITE`, it can no longer be reverted to a finite period.

6.6.3.5 bytes_per_token

This field determines the number of bytes that can actually be transmitted based on the number of tokens.

Tokens are always consumed in whole by each *DataWriter*. That is, in cases where **bytes_per_token** is greater than the DDS sample size, multiple DDS samples may be sent to the same destination using a single token (regardless of the **scheduling_policy**).

Where fragmentation is required, the fragment size will be either (a) **bytes_per_token** or (b) the minimum of the largest message sizes across all transports installed with the *DataWriter*, whichever is less.

Use `DDS_LENGTH_UNLIMITED` to indicate that an unlimited number of bytes can be transmitted per token. In other words, a single token allows the recipient *DataWriter* to transmit all its queued DDS samples to a single destination. A separate token is required to send to each additional destination.

6.6.4 Prioritized DDS Samples

Note: This feature is not supported when using the Ada API.

The *Prioritized DDS Samples* feature allows you to prioritize traffic that is in competition for transmission resources. The granularity of this prioritization may be by *DataWriter*, by instance, or by individual DDS sample.

Prioritized DDS Samples can improve latency in the following cases:

- Low-Availability Links

With low-availability communication, unsent DDS samples may accumulate while the link is unavailable. When the link is restored, a large number of DDS samples may be waiting for transmission. High priority DDS samples will be sent first.

- Low-Bandwidth Links

With low-bandwidth communication, a temporary backlog may occur or the link may become congested with large DDS samples. High-priority DDS samples will be sent at the first available gap, between the fragments of a large low-priority DDS sample.

- Prioritized Topics

With limited bandwidth communication, some topics may be deemed to be of higher priority than others on an ongoing basis, and DDS samples written to some topics should be given precedence over others on transmission.

- High Priority Events

Due to external rules or content analysis (e.g., perimeter violation or identification as a threat), the priority of DDS samples is dynamically determined, and the priority assigned a given DDS sample will reflect the urgency of its delivery.

To configure a *DataWriter* to use prioritized DDS samples:

- Create a FlowController with the **scheduling_policy** property set to `DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY`.
- Create a *DataWriter* with the **PUBLISH_MODE** QosPolicy (DDS Extension) (Section 6.5.18 on page 397) **kind** set to `ASYNCHRONOUS` and **flow_controller_name** set to the name of the FlowController.

A single FlowController may perform traffic shaping for multiple *DataWriters* and multiple *DataWriter* channels. The FlowController's configuration determines how often publication resources are scheduled, how much data may be sent per period, and other transmission characteristics that determine the ultimate performance of prioritized DDS samples.

When working with prioritized DDS samples, you should use these operations, which allow you to specify priority:

- **write_w_params()** (see [Writing Data \(Section 6.3.8 on page 283\)](#))
- **unregister_instance_w_params()** (see [Registering and Unregistering Instances \(Section 6.3.14.1 on page 297\)](#))
- **dispose_w_params()** (see [Disposing of Data \(Section 6.3.14.2 on page 299\)](#))

If you use **write()**, **unregister()**, or **dispose()** instead of the **_w_params()** versions, the affected DDS sample is assigned priority 0 (undefined priority). If you are using a multi-channel *DataWriter* with a priority filter, and you have no channel for priority 0, the DDS sample will be discarded.

6.6.4.1 Designating Priorities

For *DataWriters* and *DataWriter* channels, valid publication priority values are:

- `DDS_PUBLICATION_PRIORITY_UNDEFINED`
- `DDS_PUBLICATION_PRIORITY_AUTOMATIC`
- Positive integers excluding zero

For individual DDS samples, valid publication priority values are 0 and positive integers.

There are three ways to set the publication priority of a *DataWriter* or *DataWriter* channel:

1. For a *DataWriter*, publication priority is set in the **priority** field of its **PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.18 on page 397)**. For a multi-channel *DataWriter* (see **MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.14 on page 386)**), this value will be the default publication priority for any member channel that has not been assigned a specific value.
2. For a channel of a Multi-channel *DataWriter*, publication priority can be set in the *DataWriter*'s **MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.14 on page 386)** in **channels[].priority**.
3. If a *DataWriter* or a channel of a Multi-channel *DataWriter* is configured for publication priority inheritance (DDS_PUBLICATION_PRIORITY_AUTOMATIC), its publication priority is the highest priority among all the DDS samples currently in the publication queue. When using publication priority inheritance, the publication priorities of individual DDS samples are set by calling the **write_w_params()** operation, which takes a **priority** parameter.

The *effective* publication priority is determined from the interaction of the *DataWriter*, channel, and DDS sample publication priorities, as shown in [Table 6.74 Effective Publication Priority](#) .

Table 6.74 Effective Publication Priority

	Priority Setting Combinations				
Writer Priority	Undefined	Don't care	AUTOMATIC	Don't care	Designated positive integer > 0
Channel Priority	Undefined	AUTOMATIC	Undefined	Designated positive integer > 0	Undefined
DDS Sample Priority	Don't care	Designated positive integer > 0	Designated positive integer > 0	Don't care	Don't care
Effective Priority	Lowest Priority	DDS Sample Priority ¹	DDS Sample Priority ²	Channel Priority	Writer Priority

6.6.4.2 Priority-Based Filtering

The configuration methods explained above are sufficient to create multiple *DataWriters*, each with its own assigned priority, all using the same *FlowController* configured for *publication priority*-based schedul-

¹Highest sample priority among all DDS samples currently in the publication queue.

²Highest sample priority among all DDS samples currently in the publication queue.

ing. Such a configuration is sufficient to assign different priorities to individual topics, but it does not allow different *publication priorities* to be assigned to published data *within* a *Topic*.

To assign different priorities to data within a *DataWriter*, you will need to use a Multi-channel *DataWriter* and configure the channels with different priorities. Configuring the publication priorities of *DataWriter* channels is explained above. To associate different priorities of data with different publication channels, configure the **channel[]**.**filter_expression** in the *DataWriter's* [MULTI_CHANNEL QoSPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386). The filtering criteria that is available for evaluation by each channel is determined by the filter type, which is configured with the *DataWriter's* **filter_name** (also in the [MULTI_CHANNEL QoSPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386)).

For example, using the built-in SQL-based content filter allows channel membership to be determined based on the content of each DDS sample.

If you do not want to embed priority criteria within each DDS sample, you can use a built-in filter named `DDS_PRIFILTER_NAME` that uses the publication priority that is provided when you call **write_w_params()** (see [Writing Data \(Section 6.3.8 on page 283\)](#)). The filter's expression syntax is:

```
@priority OP VAL
```

where OP can be `<`, `<=`, `>`, `>=`, `=`, or `<>` (standard relational operators), and VAL is a positive integer.

The filter supports multiple expressions, combined with the conjunctions AND and OR. You can use parentheses to disambiguate combinations of AND and OR in the same expression. For example:

```
@priority = 2 OR (@priority > 6 AND @priority < 10)
```

6.6.5 Creating and Configuring Custom FlowControllers with Property QoS

You can create and configure FlowControllers using the [PROPERTY QoSPolicy \(DDS Extension\)](#) (Section 6.5.17 on page 394). The properties must have a prefix of “`dds.flow_controller.token_bucket`”, followed by the name of the FlowController being created or configured. For example, if you want to create/configure a FlowController named **MyFC**, all the properties for **MyFC** should have the prefix “`dds.flow_controller.token_bucket.MyFC`”.

[Table 6.75 FlowController Properties](#) lists the properties that can be set for FlowControllers in the *DomainParticipant's* [PROPERTY QoSPolicy \(DDS Extension\)](#) (Section 6.5.17 on page 394). A FlowController with the name “`dds.flow_controller.token_bucket.<your flow controllername>`” will be implicitly created when at least one property using that prefix is specified. Then, to link a *DataWriter* to your FlowController, use “`dds.flow_controller.token_bucket.<your flow controllername>`” in the *DataWriter's* **publish_mode.flow_controller_name**.

Table 6.75 FlowController Properties

Property Name prefix with 'dds.flow_ controller.token_bucket.' <your flow controller name>	Property Value Description
scheduling_policy	Specifies the scheduling policy to be used. (See Flow Controller Scheduling Policies (Section 6.6.1 on page 424)) May be: DDS_RR_FLOW_CONTROLLER_SCHED_POLICY DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY
token_bucket.max_tokens	Maximum number of tokens than can accumulate in the token bucket. Use -1 for unlimited.
token_bucket.tokens_added_per_period	Number of tokens added to the token bucket per specified period. Use -1 for unlimited.
token_bucket.tokens_leaked_per_period	Number of tokens removed from the token bucket per specified period. Use -1 for unlimited.
token_bucket.period.sec	Period for adding tokens to and removing tokens from the bucket in seconds.
token_bucket.period.nanosec	Period for adding tokens to and removing tokens from the bucket in nanoseconds.
token_bucket.bytes_per_token	Maximum number of bytes allowed to send for each token available.

6.6.5.1 Example

The following example shows how to set FlowController properties.

Note: Some lines in this example, such as **dds.flow_controller.token_bucket.MyFlowController.scheduling_policy**, are too long to fit on the page as one line; however in your XML file, they each need to be on a single line.

```

<participant_qos>
  <property>
    <value>
      <element>
        <name>
dds.flow_controller.token_bucket.MyFlowController.scheduling_policy
        </name>
        <value>DDS_RR_FLOW_CONTROLLER_SCHED_POLICY</value>
      </element>
      <element>
        <name>
dds.flow_controller.token_bucket.MyFlowController.token_bucket.period.sec

```

```

        </name>
        <value>100</value>
    </element>
    <element>
        <name>
dds.flow_controller.token_bucket.MyFlowController.
token_bucket.period.nanosec
        </name>
        <value>0</value>
    </element>
    <element>
        <name>
dds.flow_controller.token_bucket.MyFlowController.token_bucket.tokens_added_per_period
        </name>
        <value>2</value>
    </element>
    <element>
        <name>
dds.flow_controller.token_bucket.MyFlowController.token_bucket.tokens_leaked_per_period
        </name>
        <value>2</value>
    </element>
    <element>
        <name>
dds.flow_controller.token_bucket.MyFlowController.token_bucket.bytes_per_token
        </name>
        <value>1024</value>
    </element>
</value>
</property>
</participant_qos>
<datawriter_qos>
    <publish_mode>
        <flow_controller_name>
            dds.flow_controller.token_bucket.MyFlowController
        </flow_controller_name>
        <kind>ASYNCHRONOUS_PUBLISH_MODE_QOS</kind>
    </publish_mode>
</datawriter_qos>

```

6.6.6 Creating and Deleting FlowControllers

(Note: in the Modern C++ API FlowControllers have reference semantics, see [Creating and Deleting Entities](#))

If you do not want to use one of the three built-in FlowControllers described in [FlowControllers \(DDS Extension\)](#) (Section 6.6 on page 422), you can create your own with the *DomainParticipant's* **create_flowcontroller()** operation:

```

DDSFlowController* create_flowcontroller
    (const char * name,
     const DDS_FlowControllerProperty_t & property)

```

To associate a FlowController with a *DataWriter*, you set the FlowController's name in the [PUBLISH_MODE QosPolicy \(DDS Extension\) \(Section 6.5.18 on page 397\)](#) (`flow_controller_name`).

A single FlowController may service multiple *DataWriters*, even if they belong to a different *Publisher*. The FlowController's **property** structure determines how the FlowController shapes the network traffic.

name Name of the FlowController to create. A *DataWriter* is associated with a DDSFlowController by name. Limited to 255 characters.

property Properties to be used for creating the FlowController. The special value `DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT` can be used to indicate that the FlowController should be created with the default `DDS_FlowControllerProperty_t` set in the *DomainParticipant*.

Note: If you use `DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT`, it is *not* safe to create the FlowController while another thread may be simultaneously calling `set_default_flowcontroller_property()` or looking for that FlowController with `lookup_flowcontroller()`.

To delete an existing FlowController, use the *DomainParticipant*'s `delete_flowcontroller()` operation:

```
DDS_ReturnCode_t delete_flowcontroller (DDSFlowController * fc)
```

The FlowController must belong this the *DomainParticipant* and not have any attached *DataWriters* or the delete call will return an error (`PRECONDITION_NOT_MET`).

6.6.7 Getting/Setting Default FlowController Properties

To get the default `DDS_FlowControllerProperty_t` values, use this operation on the *DomainParticipant*:

```
DDS_ReturnCode_t get_default_flowcontroller_property
(DDS_FlowControllerProperty_t & property)
```

The retrieved property will match the set of values specified on the last successful call to the *DomainParticipant*'s `set_default_flowcontroller_property()`, or if the call was never made, the default values listed in `DDS_FlowControllerProperty_t`.

To change the default `DDS_FlowControllerProperty_t` values used when a new FlowController is created, use this operation on the *DomainParticipant*:

```
DDS_ReturnCode_t set_default_flowcontroller_property
(const DDS_FlowControllerProperty_t & property)
```

The special value `DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT` may be passed for the **property** to indicate that the default property should be reset to the default values the factory would use if `set_default_flowcontroller_property()` had never been called.

Note: It is not safe to set the default FlowController properties while another thread may be simultaneously calling `get_default_flowcontroller_property()`, `set_default_flowcontroller_property()`, or `create_`

flowcontroller() with `DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT` as the **qos** parameter. It is also not safe to get the default FlowController properties while another thread may be simultaneously calling **get_default_flowcontroller_property()**.

6.6.8 Getting/Setting Properties for a Specific FlowController

To get the properties of a FlowController, use the FlowController's **get_property()** operation:

```
DDS_ReturnCode_t DDSFlowController::get_property
    (struct DDS_FlowControllerProperty_t & property)
```

To change the properties of a FlowController, use the FlowController's **set_property()** operation:

```
DDS_ReturnCode_t DDSFlowController::set_property
    (const struct DDS_FlowControllerProperty_t & property)
```

Once a FlowController has been instantiated, only its **token_bucket** property can be changed. The **scheduling_policy** is immutable. A new **token.period** only takes effect at the next scheduled token distribution time (as determined by its previous value).

The special value `DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT` can be used to match the current default properties set in the *DomainParticipant*.

6.6.9 Adding an External Trigger

Typically, a FlowController uses an internal trigger to periodically replenish its tokens. The period by which this trigger is called is determined by the **period** property setting.

The **trigger_flow()** function provides an additional, external trigger to the FlowController. This trigger adds **tokens_added_per_period** tokens each time it is called (subject to the other property settings of the FlowController).

```
DDS_ReturnCode_t trigger_flow ()
```

An on-demand FlowController can be created with a `DDS_DURATION_INFINITE` as **period**, in which case the only trigger source is external (i.e. the FlowController is solely triggered by the user on demand).

trigger_flow() can be called on both a strict on-demand FlowController and a hybrid FlowController (internally and externally triggered).

6.6.10 Other FlowController Operations

If you have the FlowController object and need its name, call the FlowController's **get_name()** operation:

```
const char* DDSFlowController::get_name ( )
```

Conversely, if you have the name of the FlowController and need the FlowController object, call the *DomainParticipant*'s **lookup_flowcontroller()** operation:


```
DDSFlowController* lookup_flowcontroller (const char * name)
```

To get a FlowController's *DomainParticipant*, call the FlowController's **get_participant()** operation:

```
DDSDomainParticipant* get_participant ( )
```

Note: It is not safe to lookup a FlowController description while another thread is creating that FlowController

Chapter 7 Receiving Data

This section discusses how to create, configure, and use *Subscribers* and *DataReaders* to receive data. It describes how these objects interact, as well as the types of operations that are available for them.

This section includes:

The goal of this section is to help you become familiar with the Entities you need for receiving data. For up-to-date details such as formal parameters and return codes on any mentioned operations, please see the *Connex DDS API Reference HTML* documentation.

7.1 Preview: Steps to Receiving Data

There are three ways to receive data:

- Your application can explicitly check for new data by calling a *DataReader*'s **read()** or **take()** operation. This method is also known as *polling for data*.
- Your application can be notified asynchronously whenever new DDS data samples arrive—this is done with a *Listener* on either the *Subscriber* or the *DataReader*. Connex DDS will invoke the *Listener*'s callback routine when there is new data. Within the callback routine, user code can access the data by calling **read()** or **take()** on the *DataReader*. This method is the way for your application to receive data with the least amount of latency.
- Your application can wait for new data by using *Conditions* and a *WaitSet*, then calling **wait()**. Connex DDS will block your application's thread until the criteria (such as the arrival of DDS samples, or a specific status) set in the *Condition* becomes true. Then your application resumes and can access the data with **read()** or **take()**.

The *DataReader*'s **read()** operation gives your application a copy of the data and leaves the data in the *DataReader*'s receive queue. The *DataReader*'s **take()** operation removes data from the receive queue before giving it to your application.

See [Using DataReaders to Access Data \(Read & Take\) \(Section 7.4 on page 491\)](#) for details on using *DataReaders* to access received data.

See [Conditions and WaitSets \(Section 4.6 on page 187\)](#) for details on using *Conditions* and *WaitSets*.

To prepare to receive data, create and configure the required Entities:

1. Create a *DomainParticipant*.
2. Register user data types¹ with the *DomainParticipant*. For example, the '**FooDataType**'.
3. Use the *DomainParticipant* to create a *Topic* with the registered data type.
4. Optionally², use the *DomainParticipant* to create a *Subscriber*.
5. Use the *Subscriber* or *DomainParticipant* to create a *DataReader* for the *Topic*.
6. Use a type-safe method to cast the generic *DataReader* created by the *Subscriber* to a type-specific *DataReader*. For example, '**FooDataReader**'.

Then use one of the following mechanisms to receive data.

- To receive DDS data samples by polling for new data:
 - Using a **FooDataReader**, use the **read()** or **take()** operations to access the DDS data samples that have been received and stored for the *DataReader*. These operations can be invoked at any time, even if the receive queue is empty.
 - To receive DDS data samples asynchronously:
 - Install a *Listener* on the *DataReader* or *Subscriber* that will be called back by an internal Connext DDS thread when new DDS data samples arrive for the *DataReader*.
1. Create a *DDSDataReaderListener* for the *FooDataReader* or a *DDSSubscriberListener* for *Subscriber*. In C++, C++/CLI, C# and Java, you must derive your own *Listener* class from those base classes. In C, you must create the individual functions and store them in a structure.

If you created a *DDSDataReaderListener* with the **on_data_available()** callback enabled: **on_data_available()** will be called when new data arrives for that *DataReader*.

If you created a *DDSSubscriberListener* with the **on_data_on_readers()** callback enabled: **on_data_on_readers()** will be called when data arrives for any *DataReader* created by the *Subscriber*.

¹Type registration is not required for built-in types (see [Registering Built-in Types \(Section 3.2.1 on page 30\)](#)).

²You are not required to explicitly create a *Subscriber*; instead, you can use the 'implicit *Subscriber*' created from the *DomainParticipant*. See [Creating Subscribers Explicitly vs. Implicitly \(Section 7.2.1 on page 444\)](#).

2. Install the *Listener* on either the **FooDataReader** or *Subscriber*.

For the *DataReader*, the *Listener* should be installed to handle changes in the **DATA_AVAILABLE** status.

For the *Subscriber*, the *Listener* should be installed to handle changes in the **DATA_ON_READERS** status.

3. Only 1 *Listener* will be called back when new data arrives for a *DataReader*.

Connex DDS will call the *Subscriber's Listener* if it is installed. Otherwise, the *DataReader's Listener* is called if it is installed. That is, the **on_data_on_readers()** operation takes precedence over the **on_data_available()** operation.

If neither *Listeners* are installed or neither *Listeners* are enabled to handle their respective statuses, then Connex DDS will not call any user functions when new data arrives for the *DataReader*.

4. In the **on_data_available()** method of the *DDSDataReaderListener*, invoke **read()** or **take()** on the **FooDataReader** to access the data.

If the **on_data_on_readers()** method of the *DDSSubscriberListener* is called, the code can invoke **read()** or **take()** directly on the *Subscriber's DataReaders* that have received new data. Alternatively, the code can invoke the *Subscriber's notify_datareaders()* operation. This will in turn call the **on_data_available()** methods of the *DataReaderListeners* (if installed and enabled) for each of the *DataReaders* that have received new DDS data samples.

To wait (block) until DDS data samples arrive:

1. Use the *DataReader* to create a *ReadCondition* that describes the DDS samples for which you want to wait. For example, you can specify that you want to wait for never-before-seen DDS samples from *DataReaders* that are still considered to be 'alive.'

Alternatively, you can create a *StatusCondition* that specifies you want to wait for the **ON_DATA_AVAILABLE** status.

2. Create a *WaitSet*.
3. Attach the *ReadCondition* or *StatusCondition* to the *WaitSet*.
4. Call the *WaitSet's wait()* operation, specifying how long you are willing to wait for the desired DDS samples. When **wait()** returns, it will indicate that it timed out, or that the attached *Condition* become true (and therefore the desired DDS samples are available).
5. Using a **FooDataReader**, use the **read()** or **take()** operations to access the DDS data samples that have been received and stored for the *DataReader*.

7.2 Subscribers

An application that intends to subscribe to information needs the following Entities: *DomainParticipant*, *Topic*, *Subscriber*, and *DataReader*. All Entities have a corresponding specialized *Listener* and a set of QosPolicies. The *Listener* is how Connex DDS notifies your application of status changes relevant to the Entity. The QosPolicies allow your application to configure the behavior and resources of the Entity.

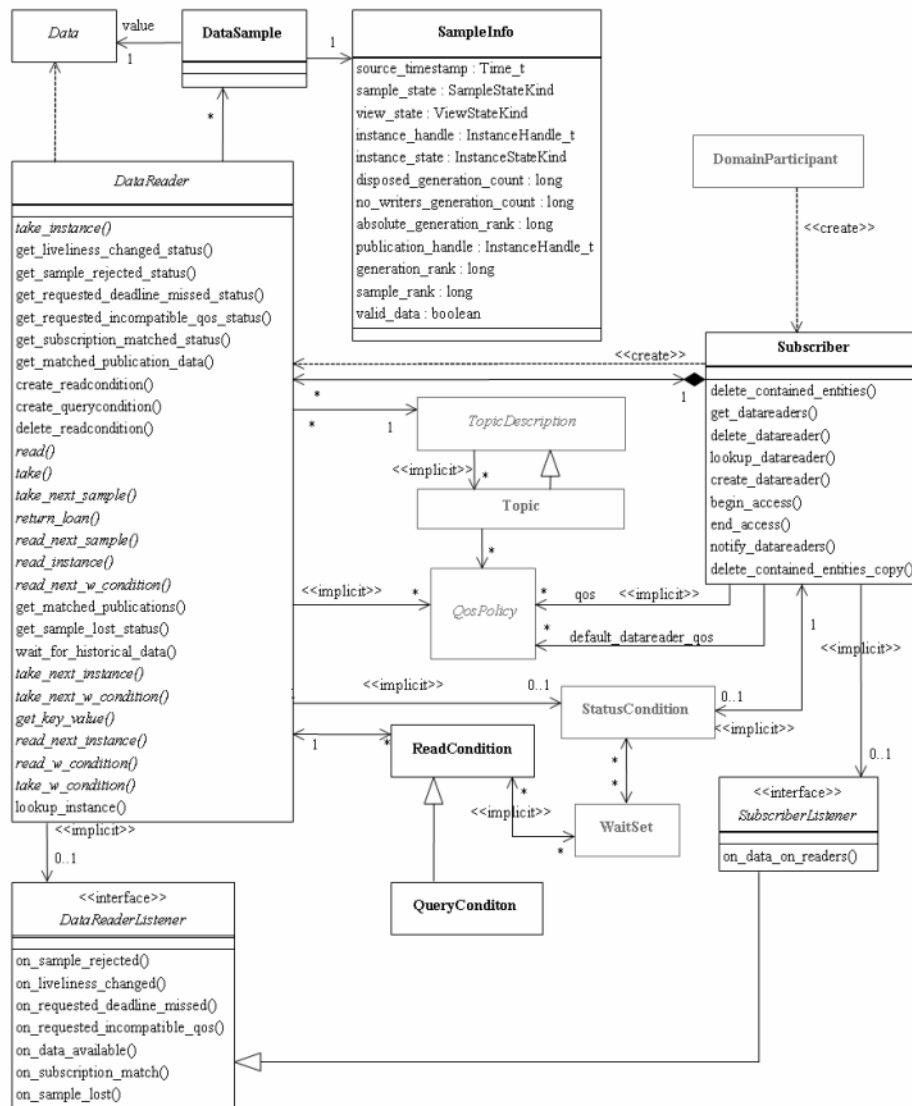
- The *DomainParticipant* defines the DDS domain on which the information will be available.
- The *Topic* defines the name of the data to be subscribed, as well as the type (format) of the data itself.
- The *DataReader* is the Entity used by the application to subscribe to updated values of the data. The *DataReader* is bound at creation time to a *Topic*, thus specifying the named and typed data stream to which it is subscribed. The application uses the *DataWriter*'s **read()** or **take()** operation to access DDS data samples received for the *Topic*.
- The *Subscriber* manages the activities of several *DataReader* entities. The application receives data using a *DataReader* that belongs to a *Subscriber*. However, the *Subscriber* will determine when the data received from applications is actually available for access through the *DataReader*. Depending on the settings of various QosPolicies of the *Subscriber* and *DataReader*, data may be buffered until DDS data samples for associated *DataReaders* are also received. By default, the data is available to the application as soon as it is received.

For more information, see [Creating Subscribers Explicitly vs. Implicitly \(Section 7.2.1 on page 444\)](#).

The UML diagram in [Subscription Module \(Section Figure 7.1 on the facing page\)](#) shows how these *Entities* are related as well as the methods defined for each Entity.

Subscribers are used to perform the operations listed in [Table 7.1 Subscriber Operations](#). For details such as formal parameters and return codes, please see the API Reference HTML documentation. Otherwise, you can find more information about the operations by looking in the section listed under the [Reference \(Section on page 442\)](#) column.

Figure 7.1 Subscription Module



Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

Table 7.1 Subscriber Operations

Working with ...	Operation	Description	Reference
DataReaders	begin_access	Indicates that the application is about to access the DDS data samples in the <i>DataReaders</i> of the <i>Subscriber</i> .	Beginning and Ending Group-Ordered Access (Section 7.2.5 on page 453)
	create_datareader	Creates a <i>DataReader</i> .	Creating DataReaders (Section 7.3.1 on page 463)
	create_datareader_with_profile	Creates a <i>DataReader</i> with QoS from a specified QoS profile.	
	copy_from_topic_qos	Copies relevant QoS Policies from a <i>Topic</i> into a <i>DataReaderQoS</i> structure.	Subscriber QoS-Related Operations (Section 7.2.4.6 on page 453)
DataReaders cont'd	delete_contained_entities	Deletes all the <i>DataReaders</i> that were created by the <i>Subscriber</i> . Also deletes the corresponding <i>ReadConditions</i> created by the contained <i>DataReaders</i> .	Deleting Contained DataReaders (Section 7.2.3.1 on page 447)
	delete_datareader	Deletes a specific <i>DataReader</i> .	Deleting DataReaders (Section 7.3.3 on page 466)
	end_access	Indicates that the application is done accessing the DDS data samples in the <i>DataReaders</i> of the <i>Subscriber</i> .	Beginning and Ending Group-Ordered Access (Section 7.2.5 on page 453)
	get_all_datareaders	Retrieves all the <i>DataReaders</i> created from this <i>Subscriber</i> .	Getting All DataReaders (Section 7.3.2 on page 465)
	get_datareaders	Returns a list of <i>DataReaders</i> that contain DDS samples with the specified sample_states , view_states and instance_states .	Getting DataReaders with Specific DDS Samples (Section 7.2.7 on page 456)
	get_default_datareader_qos	Copies the <i>Subscriber</i> 's default <i>DataReaderQoS</i> values into a <i>DataReaderQoS</i> structure.	Setting Subscriber QoS Policies (Section 7.2.4 on page 447)

Table 7.1 Subscriber Operations

Working with ...	Operation	Description	Reference
DataReaders cont'd	get_status_changes	Gets all status changes.	Getting Status and Status Changes (Section 4.1.4 on page 157)
	lookup_datareader	Retrieves a <i>DataReader</i> previously created for a specific <i>Topic</i> .	Finding a Subscriber's Related Entities (Section 7.2.8 on page 457)
	notify_datareaders	Invokes the on_data_available() operation for attached <i>Listeners</i> of <i>DataReaders</i> that have new DDS data samples.	Setting Up SubscriberListeners (Section 7.2.6 on page 454)
	set_default_datareader_qos	Sets or changes the <i>Subscriber</i> 's default <i>DataReaderQoS</i> values.	Setting Subscriber QoS Policies (Section 7.2.4 on page 447)
Libraries and Profiles	get_default_library	Gets the <i>Subscriber</i> 's default QoS profile library.	Getting and Settings Subscriber's Default QoS Profile and Library (Section 7.2.4.4 on page 451)
	get_default_profile	Gets the <i>Subscriber</i> 's default QoS profile.	
	get_default_profile_library	Gets the library that contains the <i>Subscriber</i> 's default QoS profile.	
	set_default_library	Sets the default library for a <i>Subscriber</i> .	
	set_default_profile	Sets the default profile for a <i>Subscriber</i> .	
Participants	get_participant	Gets the <i>Subscriber</i> 's <i>DomainParticipant</i> .	Finding a Subscriber's Related Entities (Section 7.2.8 on page 457)

Table 7.1 Subscriber Operations

Working with ...	Operation	Description	Reference
Subscribers	enable	Enables the <i>Subscriber</i> .	Enabling DDS Entities (Section 4.1.2 on page 154)
	equals	Compares two <i>Subscriber</i> 's QoS structures for equality.	Comparing QoS Values (Section 7.2.4.2 on page 450)
	get_listener	Gets the currently installed <i>Listener</i> .	Setting Up SubscriberListeners (Section 7.2.6 on page 454)
	get_qos	Gets the <i>Subscriber</i> 's current QoSPolicy settings. This is most often used in preparation for calling set_qos.	Changing QoS Settings After Subscriber Has Been Created (Section 7.2.4.3 on page 450)
	set_listener	Sets the <i>Subscriber</i> 's <i>Listener</i> . If you created the <i>Subscriber</i> without a <i>Listener</i> , you can use this operation to add one later.	Setting Up SubscriberListeners (Section 7.2.6 on page 454)
	set_qos	Sets the <i>Subscriber</i> 's QoS. You can use this operation to change the values for the <i>Subscriber</i> 's QoS Policies. Note, however, that not all QoS Policies can be changed after the <i>Subscriber</i> has been created.	Changing QoS Settings After Subscriber Has Been Created (Section 7.2.4.3 on page 450)
	set_qos_with_profile	Sets the <i>Subscriber</i> 's QoS based on a QoS profile.	Changing QoS Settings After Subscriber Has Been Created (Section 7.2.4.3 on page 450)

7.2.1 Creating Subscribers Explicitly vs. Implicitly

To receive data, your application must have a *Subscriber*. However, you are not required to explicitly create a *Subscriber*. If you do not create one, the middleware will implicitly create a *Subscriber* the first time you create a *DataReader* using the *DomainParticipant*'s operations. It will be created with default QoS (DDS_SUBSCRIBER_QOS_DEFAULT) and no *Listener*. The 'implicit *Subscriber*' can be accessed using the *DomainParticipant*'s `get_implicit_subscriber()` operation (see [Getting the Implicit Publisher or Subscriber \(Section 8.3.9 on page 569\)](#)). You can use this 'implicit *Subscriber*' just like any other *Subscriber* (it has the same operations, QoS Policies, etc.). So you can change the mutable QoS and set a *Listener* if desired.

A *Subscriber* (implicit or explicit) gets its own default QoS and the default QoS for its child *DataReaders* from the *DomainParticipant*. These default QoS are set when the *Subscriber* is created. (This is true for *Publishers* and *DataWriters*, too.)

DataReaders are created by calling `create_datareader()` or `create_datareader_with_profile()`—these operations exist for *DomainParticipants* and *Subscribers*¹. If you use the *DomainParticipant* to create a

¹In the Modern C++ API, you always use a *DataReader* constructor.

DataReader, it will belong to the implicit *Subscriber*. If you use a *Subscriber* to create a *DataReader*, it will belong to that *Subscriber*.

The middleware will use the same implicit *Subscriber* for all *DataReaders* that are created using the *DomainParticipant*'s operations.

Having the middleware implicitly create a *Subscriber* allows you to skip the step of creating a *Subscriber*. However, having all your *DataReaders* belong to the same *Subscriber* can reduce the concurrency of the system because all the read operations will be serialized.

7.2.2 Creating Subscribers

Before you can explicitly create a *Subscriber*, you need a *DomainParticipant* ([DomainParticipants \(Section 8.3 on page 547\)](#)). To create a *Subscriber*, use the *DomainParticipant*'s `create_subscriber()` or `create_subscriber_with_profile()` operation.

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Note: the Modern C++ API provides *Subscriber* constructors whose first, and only required argument is the *DomainParticipant*.

```
DDSSubscriber* create_subscriber(
    const DDS_SubscriberQos &qos,
    DDSSubscriberListener * listener,
    DDS_StatusMask mask)
DDSSubscriber* create_subscriber_with_profile (
    const char * library_name,
    const char * profile_name,
    DDSSubscriberListener * listener,
    DDS_StatusMask mask )
```

Where:

- qos** If you want the default QoS settings (described in the API Reference HTML documentation), use `DDS_SUBSCRIBER_QOS_DEFAULT` for this parameter (see [Figure 7.2 Creating a Subscriber with Default QoS Policies on the next page](#)). If you want to customize any of the QoS Policies, supply a QoS structure (see [Creating a Subscriber with Non-Default QoS Policies \(not from a profile\) \(Section Figure 7.3 on page 449\)](#)). The QoS structure for a *Subscriber* is described in [Subscriber QoS Policies \(Section 7.5 on page 510\)](#).

Note: If you use `DDS_SUBSCRIBER_QOS_DEFAULT`, it is not safe to create the *Subscriber* while another thread may be simultaneously calling `set_default_subscriber_qos()`.

- listener** *Listeners* are callback routines. Connex DDS uses them to notify your application when specific events (new DDS data samples arrive and status changes) occur with respect to the *Subscriber* or the *DataReaders* created by the *Subscriber*. The *listener* parameter may be set to NULL if you do not want to install a *Listener*. If you use NULL, the *Listener* of the *DomainParticipant* to which the *Subscriber* belongs will be used instead (if it is set). For more information on *SubscriberListeners*, see [Setting Up SubscriberListeners \(Section 7.2.6 on page 454\)](#).
- mask** This bit-mask indicates which status changes will cause the *Subscriber's Listener* to be invoked. The bits set in the mask must have corresponding callbacks implemented in the *Listener*. If you use NULL for the *Listener*, use DDS_STATUS_MASK_NONE for this parameter. If the *Listener* implements all callbacks, use DDS_STATUS_MASK_ALL. For information on Status, see [Listeners \(Section 4.4 on page 177\)](#).
- This bit-mask indicates which status changes will cause the *Subscriber's Listener* to be invoked. The bits set in the mask must have corresponding callbacks implemented in the *Listener*. If you use NULL for the *Listener*, use DDS_STATUS_MASK_NONE for this parameter. If the *Listener* implements all callbacks, use DDS_STATUS_MASK_ALL. For information on Status, see [Listeners \(Section 4.4 on page 177\)](#).
- library_name** A QoS Library is a named set of QoS profiles. See [URL Groups \(Section 17.8 on page 814\)](#).
- profile_name** A QoS profile groups a set of related QoS, usually one per entity. See [URL Groups \(Section 17.8 on page 814\)](#).

Figure 7.2 Creating a Subscriber with Default QoS Policies

```
// create the subscriber
DDSSubscriber* subscriber =
    participant->create_subscriber(
        DDS_SUBSCRIBER_QOS_DEFAULT,
        NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}
```

For more examples, see [Configuring QoS Settings when the Subscriber is Created \(Section 7.2.4.1 on page 448\)](#).

After you create a *Subscriber*, the next step is to use the *Subscriber* to create a *DataReader* for each *Topic*, see [Creating DataReaders \(Section 7.3.1 on page 463\)](#). For a list of operations you can perform with a *Subscriber*, see [Table 7.1 Subscriber Operations](#).

7.2.3 Deleting Subscribers

(Note: in the Modern C++ API, *Entities* are automatically destroyed, see [Creating and Deleting DDS Entities \(Section 4.1.1 on page 153\)](#))

This section applies to both implicitly and explicitly created *Subscribers*.

To delete a *Subscriber*:

1. You must first delete all *DataReaders* that were created with the *Subscriber*. Use the *Subscriber*'s **delete_datareader()** operation ([Creating DataReaders \(Section 7.3.1 on page 463\)](#)) to delete them one at a time, or use the **delete_contained_entities()** operation ([Deleting Contained DataReaders \(Section 7.2.3.1 below\)](#)) to delete them all at the same time.

```
DDS_ReturnCode_t delete_datareader (DDSDataReader *a_datareader)
```

2. Delete the *Subscriber* by using the *DomainParticipant*'s **delete_subscriber()** operation ().

Note: A *Subscriber* cannot be deleted within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

7.2.3.1 Deleting Contained DataReaders

The *Subscriber*'s **delete_contained_entities()** operation deletes all the *DataReaders* that were created by the *Subscriber*. It also deletes the *ReadConditions* created by each contained *DataReader*.

```
DDS_ReturnCode_t DDSSubscriber::delete_contained_entities ()
```

After this operation returns successfully, the application may delete the *Subscriber* (see [Deleting Subscribers \(Section 7.2.3 on the previous page\)](#)).

The operation will return **PRECONDITION_NOT_MET** if any of the contained entities cannot be deleted. This will occur, for example, if a contained *DataReader* cannot be deleted because the application has called **read()** but has not called the corresponding **return_loan()** operation to return the loaned DDS samples.

7.2.4 Setting Subscriber QoS Policies

A *Subscriber*'s QoS Policies control its behavior. Think of the policies as the configuration and behavior 'properties' for the *Subscriber*. The **DDS_SubscriberQos** structure has the following format:

```
struct DDS_SubscriberQos {
    DDS_PresentationQosPolicy    presentation;
    DDS_PartitionQosPolicy      partition;
    DDS_GroupDataQosPolicy      group_data;
    DDS_EntityFactoryQosPolicy  entity_factory;
    DDS_ExclusiveAreaQosPolicy  exclusive_area;
    DDS_EntityNameQosPolicy     subscriber_name;
};
```

Note: `set_qos()` cannot always be used by a *Listener*, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

[Table 7.2 Subscriber QoS Policies](#) summarizes the meaning of each policy. *Subscribers* have the same set of QoS Policies as *Publishers*; they are described in detail in [Publisher/Subscriber QoS Policies \(Section 6.4 on page 312\)](#). For information on *why* you would want to change a particular QoS Policy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation for each policy.

Table 7.2 Subscriber QoS Policies

QoS Policy	Description
ENTITYFACTORY QoS Policy (Section 6.4.2 on page 315)	Whether or not new entities created from this entity will start out as 'enabled.'
ENTITY_NAME QoS Policy (DDS Extension) (Section 6.5.9 on page 374)	Assigns a name and <code>role_name</code> to a <i>Subscriber</i> .
EXCLUSIVE_AREA QoS Policy (DDS Extension) (Section 6.4.3 on page 318)	Whether or not the entity uses a multi-thread safe region with deadlock protection.
GROUP_DATA QoS Policy (Section 6.4.4 on page 320)	A place to pass group-level information among applications. Usage is application-dependent.
PARTITION QoS Policy (Section 6.4.5 on page 323)	Set of strings that introduces a logical partition among Topics visible by Publisher/Subscriber.
PRESENTATION QoS Policy (Section 6.4.6 on page 330)	The order in which instance changes are presented to the Subscriber. By default, no order is used.

7.2.4.1 Configuring QoS Settings when the Subscriber is Created

As described in [Creating Subscribers \(Section 7.2.2 on page 445\)](#), there are different ways to create a *Subscriber*, depending on how you want to specify its QoS (with or without a QoS Profile).

- In [Creating Subscribers \(Section 7.2.2 on page 445\)](#) is an example of how to explicitly create a *Subscriber* with default QoS Policies. It used the special constant, `DDS_SUBSCRIBER_QOS_DEFAULT`, which indicates that the default QoS values for a *Subscriber* should be used. The default Subscriber QoS Policies are configured in the *DomainParticipant*; you can change them with the *DomainParticipant*'s `set_default_subscriber_qos()` or `set_default_subscriber_qos_with_profile()` operation (see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on page 568\)](#)).
- To create a *Subscriber* with non-default QoS settings, without using a QoS profile, see [Figure 7.3 Creating a Subscriber with Non-Default QoS Policies \(not from a profile\) on the facing page](#). It uses the *DomainParticipant*'s `get_default_subscriber_qos()` method to initialize a `DDS_Sub-`

subscriberQos structure. Then the policies are modified from their default values before the QoS structure is passed to `create_subscriber()`.

- You can also create a *Subscriber* and specify its QoS settings via a QoS Profile. To do so, call `create_subscriber_with_profile()`, as seen in [Figure 7.4 Creating a Subscriber with a QoS Profile below](#).
- If you want to use a QoS profile, but then make some changes to the QoS before creating the *Subscriber*, call `get_subscriber_qos_from_profile()`, modify the QoS and use the modified QoS structure when calling `create_subscriber()`, as seen in [Figure 7.5 Getting QoS Values from a Profile, Changing QoS Values, Creating a Subscriber with Modified QoS Values on the next page](#).

For more information, see [Creating Subscribers \(Section 7.2.2 on page 445\)](#) and [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 7.3 Creating a Subscriber with Non-Default QoS Policies (not from a profile)

```
DDS_SubscriberQos subscriber_qos;1
// get defaults
if (participant->get_default_subscriber_qos(subscriber_qos) !=
    DDS_RETCODE_OK) {
// handle error
}
// make QoS changes here. for example, this changes the ENTITY_FACTORY QoS
subscriber_qos.entity_factory.autoenable_created_entities=DDS_BOOLEAN_FALSE;
// create the subscriber
DDSSubscriber * subscriber = participant->create_subscriber(subscriber_qos,
    NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
// handle error
}
```

Figure 7.4 Creating a Subscriber with a QoS Profile

```
// create the subscriber with QoS profile
DDSSubscriber * subscriber = participant->create_subscriber_with_profile(
    "MySubscriberLibrary", "MySubscriberProfile", NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
// handle error
}
```

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

Figure 7.5 Getting QoS Values from a Profile, Changing QoS Values, Creating a Subscriber with Modified QoS Values

```

DDS_SubscriberQos subscriber_qos;1
// Get subscriber QoS from profile
retcode = factory->get_subscriber_qos_from_profile(subscriber_qos,
    "SubscriberLibrary", "SubscriberProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
// for example, this changes the ENTITY_FACTORY QoS
subscriber_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_TRUE;
// create the subscriber with modified QoS
DDSPublisher* subscriber = participant->create_subscriber(
    "Example Foo", type_name, subscriber_qos,
    NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}

```

7.2.4.2 Comparing QoS Values

The **equals()** operation compares two *Subscriber*'s `DDS_SubscriberQoS` structures for equality. It takes two parameters for the two *Subscriber*'s QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

7.2.4.3 Changing QoS Settings After Subscriber Has Been Created

There are 2 ways to change an existing *Subscriber*'s QoS after it has been created—again depending on whether or not you are using a QoS Profile.

- To change an existing *Subscriber*'s QoS programmatically (that is, without using a QoS profile), **get_qos()** and **set_qos()**. See the example code in [Figure 7.6 Changing the QoS of an Existing Subscriber on the facing page](#). It retrieves the current values by calling the *Subscriber*'s **get_qos()** operation. Then it modifies the value and calls **set_qos()** to apply the new value. Note, however, that some QoS Policies cannot be changed after the *Subscriber* has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.
- You can also change a *Subscriber*'s (and all other Entities') QoS by using a QoS Profile and calling **set_qos_with_profile()**. For an example, see [Figure 7.7 Changing the QoS of an Existing](#)

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

[Subscriber with a QoS Profile on the facing page](#). For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 7.6 Changing the QoS of an Existing Subscriber

```
DDS_SubscriberQos subscriber_qos;
// Get current QoS. subscriber points to an existing DDSSubscriber.
if (subscriber->get_qos(subscriber_qos) != DDS_RETCODE_OK) {
    // handle error
}
// make changes
// New entity_factory autoenable_created_entities will be true
subscriber_qos.entity_factory.autoenable_created_entities =
    DDS_BOOLEAN_TRUE;
// Set the new QoS
if (subscriber->set_qos(subscriber_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 7.7 Changing the QoS of an Existing Subscriber with a QoS Profile

```
retcode = subscriber->set_qos_with_profile(
    "SubscriberProfileLibrary", "SubscriberProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

7.2.4.4 Getting and Settings Subscriber's Default QoS Profile and Library

You can retrieve the default QoS profile used to create *Subscribers* with the `get_default_profile()` operation. You can also get the default library for *Subscribers*, as well as the library that contains the *Subscriber's* default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

```
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()
```

There are also operations for setting the *Subscriber's* default library and profile:

```
DDS_ReturnCode_t set_default_library (
    const char * library_name)
DDS_ReturnCode_t set_default_profile (
    const char * library_name,
    const char * profile_name)
```


These operations only affect which library/profile will be used as the default the next time a default *Subscriber* library/profile is needed during a call to one of this *Subscriber*'s operations.

When calling a *Subscriber* operation that requires a **profile_name** parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

If the default library/profile is not set, the *Subscriber* inherits the default from the *DomainParticipant*.

set_default_profile() does not set the default QoS for *DataReaders* created by the *Subscriber*; for this functionality, use the *Subscriber*'s **set_default_datareader_qos_with_profile()**, see [Getting and Setting Default QoS for DataReaders \(Section 7.2.4.5 below\)](#) (you may pass in NULL after having called the *Subscriber*'s **set_default_profile()**).

set_default_profile() does not set the default QoS for newly created *Subscribers*; for this functionality, use the *DomainParticipant*'s **set_default_subscriber_qos_with_profile()** operation, see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on page 568\)](#).

7.2.4.5 Getting and Setting Default QoS for DataReaders

These operations *set* the default QoS that will be used for new *DataReaders* if **create_datareader()** is called with DDS_DATAREADER_QOS_DEFAULT as the 'qos' parameter:

```
DDS_ReturnCode_t set_default_datareader_qos (const DDS_DataReaderQos &qos)

DDS_ReturnCode_t set_default_datareader_qos_with_profile (
    const char *library_name, const char *profile_name)
```

The above operations may potentially allocate memory, depending on the sequences contained in some QoS policies.

To *get* the default QoS that will be used for creating *DataReaders* if **create_datareader()** is called with DDS_DATAREADER_QOS_DEFAULT as the 'qos' parameter:

```
DDS_ReturnCode_t get_default_datareader_qos (DDS_DataReaderQos & qos)
```

The above operation gets the QoS settings that were specified on the last successful call to **set_default_datareader_qos()** or **set_default_datareader_qos_with_profile()**, or if the call was never made, the default values listed in DDS_DataReaderQos.

Note: It is not safe to set the default *DataReader* QoS values while another thread may be simultaneously calling **get_default_datareader_qos()**, **set_default_datareader_qos()** or **create_datareader()** with DDS_DATAREADER_QOS_DEFAULT as the **qos** parameter. It is also not safe to get the default *DataReader* QoS values while another thread may be simultaneously calling **set_default_datareader_qos()**.

7.2.4.6 Subscriber QoS-Related Operations

- **Copying a Topic's QoS into a DataReader's QoS**

This method is provided as a convenience for setting the values in a *DataReaderQos* structure before using that structure to create a *DataReader*. As explained in [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#), most of the policies in a *TopicQos* structure do not apply directly to the *Topic* itself, but to the associated *DataWriters* and *DataReaders* of that *Topic*. The *TopicQos* serves as a single container where the values of QoS Policies that must be set compatibly across matching *DataWriters* and *DataReaders* can be stored.

Thus instead of setting the values of the individual QoS Policies that make up a *DataReaderQos* structure every time you need to create a *DataReader* for a *Topic*, you can use the *Subscriber's* **copy_from_topic_qos()** operation to “import” the *Topic's* QoS Policies into a *DataReaderQos* structure. This operation copies the relevant policies in the *TopicQos* to the corresponding policies in the *DataReaderQos*.

This copy operation will often be used in combination with the *Subscriber's* **get_default_datareader_qos()** and the *Topic's* **get_qos()** operations. The *Topic's* QoS values are merged on top of the *Subscriber's* default *DataReader* QoS Policies with the result used to create a new *DataReader*, or to set the QoS of an existing one (see [Setting DataReader QoS Policies \(Section 7.3.8 on page 482\)](#)).

- **Copying a Subscriber's QoS**

In the C API users should use the **DDS_SubscriberQos_copy()** operation rather than using structure assignment when copying between two QoS structures. The **copy()** operation will perform a deep copy so that policies that allocate heap memory such as sequences are copied correctly. In C++, C++/CLI, C# and Java, a copy constructor is provided to take care of sequences automatically.

- **Clearing QoS-Related Memory**

Some QoS Policies contain sequences that allocate memory dynamically as they grow or shrink. The C API's **DDS_SubscriberQos_finalize()** operation frees the memory used by sequences but otherwise leaves the QoS unchanged. C users should call **finalize()** on all **DDS_SubscriberQos** objects before they are freed, or for QoS structures allocated on the stack, before they go out of scope. In C++, C++/CLI, C# and Java, the memory used by sequences is freed in the destructor.

7.2.5 Beginning and Ending Group-Ordered Access

The *Subscriber's* **begin_access()** operation indicates that the application is about to access the DDS data samples in any of the *DataReaders* attached to the *Subscriber*.

If the *Subscriber's* **access_scope** (in the [PRESENTATION QoS Policy \(Section 6.4.6 on page 330\)](#)) is **GROUP** or **HIGHEST_OFFERED** and **ordered_access** (also in the [PRESENTATION QoS Policy](#)

([Section 6.4.6 on page 330](#)) is TRUE, the application is required to use this operation to access the DDS samples in order across *DataWriters* of the same group (*Publisher* with `access_scope` GROUP).

In the above case, `begin_access()` must be called prior to calling any of the sample-accessing operations: `get_datareaders()` on the *Subscriber*, and `read()`, `take()`, `read_w_condition()`, and `take_w_condition()` on any *DataReader*.

Once the application has finished accessing the DDS data samples, it must call `end_access()`.

The application is not required to call `begin_access()` and `end_access()` to access the DDS samples in order if the *Publisher's* `access_scope` is something other than GROUP. In this case, calling `begin_access()` and `end_access()` is not considered an error and has no effect.

Calls to `begin_access()` and `end_access()` may be nested and must be balanced. That is, `end_access()` close a previous call to `begin_access()`.

7.2.6 Setting Up SubscriberListeners

Like all Entities, *Subscribers* may optionally have *Listeners*. *Listeners* are user-defined objects that implement a DDS-defined interface (i.e. a pre-defined set of callback functions). *Listeners* provide the means for Connex DDS to notify applications of any changes in *Statuses* (events) that may be relevant to it. By writing the callback functions in the *Listener* and installing the *Listener* into the *Subscriber*, applications can be notified to handle the events of interest. For more general information on *Listeners* and *Statuses*, see [Listeners \(Section 4.4 on page 177\)](#).

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

As illustrated in [Subscription Module \(Section Figure 7.1 on page 441\)](#), the *SubscriberListener* interface extends the *DataReaderListener* interface. In other words, the *SubscriberListener* interface contains all the functions in the *DataReaderListener* interface. In addition, a *SubscriberListener* has an additional function: `on_data_on_readers()`, corresponding to the *Subscriber's* `DATA_ON_READERS` status. This is the only status that is specific to a *Subscriber*. This status is closely tied to the `DATA_AVAILABLE` status ([DATA_AVAILABLE Status \(Section 7.3.7.1 on page 471\)](#)) of *DataReaders*.

The *Subscriber's* `DATA_ON_READERS` status is set whenever the `DATA_AVAILABLE` status is set for any of the *DataReaders* created by the *Subscriber*. This implies that one of its *DataReaders* has received new DDS data samples. When the `DATA_ON_READERS` status is set, the *SubscriberListener's* `on_data_on_readers()` method will be invoked.

The `DATA_ON_READERS` status of a *Subscriber* takes precedence over the `DATA_AVAILABLE` status of any of its *DataReaders*. Thus, when data arrives for a *DataReader*, the `on_data_on_readers()` operation of the *SubscriberListener* will be called instead of the `on_data_available()` operation of the *DataReaderListener*—assuming that the *Subscriber* has a *Listener* installed that is enabled to handle changes in the `DATA_ON_READERS` status. (Note however, that in the *SubscriberListener's* `on_`

data_on_readers() operation, you may choose to call **notify_datareaders()**, which in turn may cause the *DataReaderListener*'s **on_data_available()** operation to be called.)

All of the other methods of a *SubscriberListener* will be called back for changes in the *Statuses* of *Subscriber*'s *DataReaders* only if the *DataReader* is not set up to handle the statuses itself.

If you want a *Subscriber* to handle status events for its *DataReaders*, you can set up a *SubscriberListener* during the *Subscriber*'s creation or use the **set_listener()** method after the *Subscriber* is created. The last parameter is a bit-mask with which you should set which *Status* events that the *SubscriberListener* will handle. For example,

```
DDS_StatusMask mask =
    DDS_REQUESTED_DEADLINE_MISSED_STATUS |
    DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS;
subscriber = participant->create_subscriber(
    DDS_SUBSCRIBER_QOS_DEFAULT, listener, mask);
```

or

```
DDS_StatusMask mask =
    DDS_REQUESTED_DEADLINE_MISSED_STATUS |
    DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS;
subscriber->set_listener(listener, mask);
```

As previously mentioned, the callbacks in the *SubscriberListener* act as ‘default’ callbacks for all the *DataReaders* contained within. When Connex DDS wants to notify a *DataReader* of a relevant *Status* change (for example, **SUBSCRIPTION_MATCHED**), it first checks to see if the *DataReader* has the corresponding *DataReaderListener* callback enabled (such as the **on_subscription_matched()** operation). If so, Connex DDS dispatches the event to the *DataReaderListener* callback. Otherwise, Connex DDS dispatches the event to the corresponding *SubscriberListener* callback.

NOTE, the reverse is true for the **DATA_ON_READERS/DATA_AVAILABLE** status. When **DATA_AVAILABLE** changes for any *DataReaders* of a *Subscriber*, Connex DDS first checks to see if the *SubscriberListener* has **DATA_ON_READERS** enabled. If so, Connex DDS will invoke the **on_data_on_readers()** callback. Otherwise, Connex DDS dispatches the event to the *Listener* (**on_data_available()**) of the *DataReader* whose **DATA_AVAILABLE** status actually changed.

A particular callback in a *DataReader* is *not* enabled if either:

- The application installed a NULL *DataReaderListener* (meaning there are *no* callbacks for the *DataReader* at all).
- The application has disabled the callback for a *DataReaderListener*. This is done by turning off the associated status bit in the *mask* parameter passed to the **set_listener()** or **create_datareader()** call

when installing the *DataReaderListener* on the *DataReader*. For more information on *DataReaderListener*, see [Setting Up DataReaderListeners \(Section 7.3.4 on page 466\)](#).

Similarly, the callbacks in the *DomainParticipantListener* act as ‘default’ callbacks for all the *Subscribers* that belong to it. For more information on *DomainParticipantListeners*, see [Setting Up DomainParticipantListeners \(Section 8.3.5 on page 560\)](#).

The *Subscriber* also provides an operation called **notify_datareaders()** that can be used to invoke the **on_data_available()** callbacks of *DataReaders* who have new DDS data samples in their receive queues. Often **notify_datareaders()** will be used in the **on_data_on_readers()** callback to pass off the real processing of data from the *SubscriberListener* to the individual *DataReaderListeners*.

Calling **notify_datareaders()** causes the **DATA_ON_READERS** status to be reset.

[Simple SubscriberListener \(Section Figure 7.8 below\)](#) shows a *SubscriberListener* that simply notifies its *DataReaders* when new data arrives.

Figure 7.8 Simple SubscriberListener

```
class MySubscriberListener : public DDSSubscriberListener {
public:
    void on_data_on_readers(DDSSubscriber *);
    /* For this example we take no action other operations */
};
void MySubscriberListener::on_data_on_readers (DDSSubscriber *subscriber)
{
    // do global processing
    ...
    // now dispatch data arrival event to specific DataReaders
    subscriber->notify_datareaders();
}
```

7.2.7 Getting DataReaders with Specific DDS Samples

The *Subscriber*’s **get_datareaders()** operation retrieves a list of *DataReaders* that have DDS samples with specific **sample_states**, **view_states**, and **instance_states**.

If the application is outside a **begin_access()/end_access()** block, or if the *Subscriber*’s **access_scope** (in the [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#)) is **INSTANCE** or **TOPIC**, or **ordered_access** (also in the [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#)) is **FALSE**, the returned collection is a ‘set’ containing each *DataReader* at most once, in no specified order.

If the application is within a **begin_access()/end_access()** block, and the *Subscriber*’s **access_scope** is **GROUP** or **HIGHEST_OFFERED**, and **ordered_access** is **TRUE**, the returned collection is a ‘list’ of *DataReaders*, where a *DataReader* may appear more than one time.

To retrieve the DDS samples in the order in which they were published across *DataWriters* of the same group (a *Publisher* configured with GROUP **access_scope**), the application should **read()/take()** from each *DataReader* in the same order as appears in the output sequence. The application will move to the next *DataReader* when the **read()/take()** operation fails with NO_DATA.

```
DDS_ReturnCode_t get_datareaders (DDSDataReaderSeq & readers,
    DDS_SampleStateMask    sample_states,
    DDS_ViewStateMask     view_states,
    DDS_InstanceStateMask instance_states)
```

For more information, see [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#).

7.2.8 Finding a Subscriber's Related Entities

These *Subscriber* operations are useful for obtaining a handle to related entities:

- **get_participant()**: Gets the *DomainParticipant* with which a *Subscriber* was created.
- **lookup_datareader()**: Finds a *DataReader* created by the *Subscriber* with a *Topic* of a particular name. Note that if multiple *DataReaders* were created by the same *Subscriber* with the same *Topic*, any one of them may be returned by this method.

You can use this operation on a built-in *Subscriber* to access the built-in *DataReaders* for the built-in topics. The built-in *DataReader* is created when this operation is called on a built-in topic for the first time.

If you are going to modify the transport properties for the built-in *DataReaders*, do so *before* using this operation. Built-in transports are implicitly registered when the *DomainParticipant* is enabled or the first *DataWriter/DataReader* is created. To ensure that built-in *DataReaders* receive all the discovery traffic, you should lookup the *DataReader* before the *DomainParticipant* is enabled. Therefore the suggested sequence when looking up built-in *DataReaders* is:

1. Create a disabled *DomainParticipant* (see [ENTITYFACTORY QosPolicy \(Section 6.4.2 on page 315\)](#)).
 2. If you want to use non-default values, modify the built-in transport properties (see [Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties\(\) \(Section 15.5 on page 746\)](#)).
 3. Call **get_builtin_subscriber()** (see [Built-in DataReaders \(Section 16.2 on page 773\)](#)).
 4. Call **lookup_datareader()**.
 5. Call **enable()** on the *DomainParticipant* (see [Enabling DDS Entities \(Section 4.1.2 on page 154\)](#)).
- **DDS_Subscriber_as_Entity()**: This method is provided for C applications and is necessary when invoking the parent class *Entity* methods on *Subscribers*. For example, to call the *Entity* method **get_**

`status_changes()` on a *Subscriber*, `my_sub`, do the following:

```
DDS_Entity_get_status_changes(DDS_Subscriber_as_Entity(my_sub))
```

- **DDS_Subscriber_as_Entity()** is not provided in the C++, C++/CLI, C# and Java APIs because the object-oriented features of those languages make it unnecessary.

7.2.9 Statuses for Subscribers

The status indicators for a *Subscriber* are the same as those available for its *DataReaders*, with one additional status: **DATA_ON_READERS** ([DATA_ON_READERS Status \(Section 7.2.9.1 below\)](#)). The following statuses can be monitored by the *SubscriberListener*.

- [DATA_ON_READERS Status \(Section 7.2.9.1 below\)](#)
- [DATA_AVAILABLE Status \(Section 7.3.7.1 on page 471\)](#)
- [LIVELINESS_CHANGED Status \(Section 7.3.7.4 on page 475\)](#)
- [REQUESTED_DEADLINE_MISSED Status \(Section 7.3.7.5 on page 476\)](#)
- [REQUESTED_INCOMPATIBLE_QOS Status \(Section 7.3.7.6 on page 477\)](#)
- [SAMPLE_LOST Status \(Section 7.3.7.7 on page 478\)](#)
- [SAMPLE_REJECTED Status \(Section 7.3.7.8 on page 479\)](#)
- [SUBSCRIPTION_MATCHED Status \(Section 7.3.7.9 on page 482\)](#)

You can access *Subscriber* status by using a *SubscriberListener* or its inherited `get_status_changes()` operation (see [Getting Status and Status Changes \(Section 4.1.4 on page 157\)](#)), which can be used to explicitly poll for the **DATA_ON_READERS** status of the *Subscriber*.

7.2.9.1 DATA_ON_READERS Status

The **DATA_ON_READERS** status, like the **DATA_AVAILABLE** status for *DataReaders*, is a *read* communication status, which makes it somewhat different from other *plain* communication statuses. (See [Types of Communication Status \(Section 4.3.1 on page 170\)](#) for more information on statuses and the difference between *read* and *plain* statuses.) In particular, there is no status-specific data structure; the status is either changed or not, there is no additional associated information.

The **DATA_ON_READERS** status indicates that there is new data available for one or more *DataReaders* that belong to this *Subscriber*. The **DATA_AVAILABLE** status for each such *DataReader* will also be updated.

The **DATA_ON_READERS** status is reset (the corresponding bit in the bitmask is turned off) when you call `read()`, `take()`, or one of their variations on *any* of the *DataReaders* that belong to the *Subscriber*. This

is true even if the *DataReader* on which you call read/take is not the same *DataReader* that caused the **DATA_ON_READERS** status to be set in the first place. This status is also reset when you call **notify_datareaders()** on the *Subscriber*, or after **on_data_on_readers()** is invoked.

If a *SubscriberListener* has both **on_data_on_readers()** and **on_data_available()** callbacks enabled (by turning on both status bits), only **on_data_on_readers()** is called.

7.3 DataReaders

To create a *DataReader*, you need a *DomainParticipant*, a *Topic*, and optionally, a *Subscriber*. You need at least one *DataReader* for each *Topic* whose DDS data samples you want to receive.

After you create a *DataReader*, you will be able to use the operations listed in [Table 7.3 DataReader Operations](#). You are likely to use many of these operations from within your *DataReader's Listener*, which is invoked when there are status changes or new DDS data samples. For more details on all operations, see the API reference HTML documentation. The *DataReaderListener* is described in [Setting Up DataReaderListeners](#) (Section 7.3.4 on page 466).

DataReaders are created by using operations on a *DomainParticipant* or a *Subscriber*, as described in [Creating Subscribers Explicitly vs. Implicitly](#) (Section 7.2.1 on page 444). If you use the *DomainParticipant's* operations, the *DataReader* will belong to an implicit *Subscriber* that is automatically created by the middleware. If you use a *Subscriber's* operations, the *DataReader* will belong to that *Subscriber*. So either way, the *DataReader* belongs to a *Subscriber*.

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks](#) (Section 4.5.1 on page 185).

Table 7.3 DataReader Operations

Purpose	Operation	Description	Reference
Configuring the DataReader	enable	Enables the <i>DataReader</i> .	Enabling DDS Entities (Section 4.1.2 on page 154)
	equals	Compares two <i>DataReader's</i> QoS structures for equality.	Comparing QoS Values (Section 7.3.8.2 on page 487)
	get_qos	Gets the QoS.	Setting DataReader QoS Policies (Section 7.3.8 on page 482)
	set_qos	Modifies the QoS.	
	set_qos_with_profile	Modifies the QoS based on a QoS profile.	
	get_listener	Gets the currently installed <i>Listener</i> .	Setting Up DataReaderListeners (Section 7.3.4 on page 466)
	set_listener	Replaces the <i>Listener</i> .	

Table 7.3 DataReader Operations

Purpose	Operation	Description	Reference
Accessing DDS Data Samples with “Read” (Use FooData-Reader, see Accessing DDS Data Samples with Read or Take (Section 7.4.3 on page 493))	read	Reads (copies) a collection of DDS data samples from the <i>DataReader</i> .	Accessing DDS Data Samples with Read or Take (Section 7.4.3 on page 493)
	read_instance	Identical to read, but all DDS samples returned belong to a single instance, which you specify as a parameter.	read_instance and take_instance (Section 7.4.3.4 on page 497)
	read_instance_w_condition	Identical to read_instance, but all DDS samples returned belong to a single instance <i>and</i> satisfy a specific ReadCondition.	read_instance_w_condition and take_instance_w_condition (Section 7.4.3.7 on page 500)
	read_next_instance	Similar to read_instance, but the actual instance is not directly specified as a parameter. Instead, the DDS samples will all belong to instance ordered after the one previously read.	read_next_instance and take_next_instance (Section 7.4.3.5 on page 498)
	read_next_instance_w_condition	Accesses a collection of DDS data samples of the next instance that match a specific set of <i>ReadConditions</i> , from the <i>DataReader</i> .	read_next_instance_w_condition and take_next_instance_w_condition (Section 7.4.3.8 on page 501)
	read_next_sample	Reads the next not-previously-accessed data value from the <i>DataReader</i> .	read_next_sample and take_next_sample (Section 7.4.3.3 on page 497)
	read_w_condition	Accesses a collection of DDS data samples from the <i>DataReader</i> that match specific <i>ReadCondition</i> criteria.	read_w_condition and take_w_condition (Section 7.4.3.6 on page 500)

Table 7.3 DataReader Operations

Purpose	Operation	Description	Reference
Accessing DDS Data Samples with “Take” (Use FooData-Reader, see Accessing DDS Data Samples with Read or Take (Section 7.4.3 on page 493))	take	Like read, but the DDS samples are removed from the <i>DataReader</i> 's receive queue.	Accessing DDS Data Samples with Read or Take (Section 7.4.3 on page 493)
	take_instance	Identical to take, but all DDS samples returned belong to a single instance, which you specify as a parameter.	read_instance and take_instance (Section 7.4.3.4 on page 497)
	take_instance_w_condition	Identical to take_instance, but all DDS samples returned belong to a single instance <i>and</i> satisfy a specific ReadCondition.	read_instance_w_condition and take_instance_w_condition (Section 7.4.3.7 on page 500)
	take_next_instance	Like read_next_instance, but the DDS samples are removed from the <i>DataReader</i> 's receive queue.	read_next_instance and take_next_instance (Section 7.4.3.5 on page 498)
	take_next_instance_w_condition	Accesses (and removes) a collection of DDS data samples of the next instance that match a specific set of <i>ReadConditions</i> , from the <i>DataReader</i> .	read_next_instance_w_condition and take_next_instance_w_condition (Section 7.4.3.8 on page 501)
	take_next_sample	Like read_next_sample, but the DDS samples are removed from the <i>DataReader</i> 's receive queue.	read_next_sample and take_next_sample (Section 7.4.3.3 on page 497)
Working with DDS Data Samples and FooData-Reader (Use FooData-Reader, see Accessing DDS Data Samples with Read or Take (Section 7.4.3 on page 493))	take_w_condition	Accesses (and removes) a collection of DDS data samples from the <i>DataReader</i> that match specific <i>ReadCondition</i> criteria.	read_w_condition and take_w_condition (Section 7.4.3.6 on page 500)
	narrow	A type-safe way to cast a pointer. This takes a DDSDataReader pointer and ‘narrows’ it to a ‘FooDataReader’ where ‘Foo’ is the related data type.	Using a Type-Specific DataReader (FooDataReader) (Section 7.4.1 on page 491)
	return_loan	Returns buffers loaned in a previous read or take call.	Loaning and Returning Data and SampleInfo Sequences (Section 7.4.2 on page 492)
	get_key_value	Gets the key for an instance handle.	Getting the Key Value for an Instance (Section 7.3.9.5 on page 491)
	lookup_instance	Gets the instance handle that corresponds to an instance key.	Looking Up an Instance Handle (Section 7.3.9.4 on page 490)

Table 7.3 DataReader Operations

Purpose	Operation	Description	Reference
Acknowledging DDS Samples	acknowledge_all	Acknowledge all previously accessed DDS samples.	Acknowledging DDS Samples (Section 7.4.4 on page 502)
	acknowledge_sample	Acknowledge a single DDS sample.	
Checking Status	get_liveliness_changed_status	Gets LIVELINESS_CHANGED_STATUS status.	Statuses for DataReaders (Section 7.3.7 on page 470)
	get_requested_deadline_missed_status	Gets REQUESTED_DEADLINE_MISSED_STATUS status.	
	get_requested_incompatible_qos_status	Gets REQUESTED_INCOMPATIBLE_QOS_STATUS status.	
	get_sample_lost_status	Gets SAMPLE_LOST_STATUS status.	
	get_sample_rejected_status	Gets SAMPLE_REJECTED_STATUS status.	
	get_subscription_matched_status	Gets SUBSCRIPTION_MATCHED_STATUS status.	Getting Status and Status Changes (Section 4.1.4 on page 157)
	get_status_changes	Gets a list of statuses that changed since last time the application read the status or the listeners were called.	
	get_datareader_cache_status	Gets DATA_READER_CACHE_STATUS status.	
	get_datareader_protocol_status	Gets DATA_READER_PROTOCOL_STATUS status.	Checking DataReader Status and StatusConditions (Section 7.3.5 on page 468)
	get_matched_publication_datareader_protocol_status	Get the protocol status for this <i>DataReader</i> , per matched publication identified by the publication_handle.	Statuses for DataReaders (Section 7.3.7 on page 470)

Table 7.3 DataReader Operations

Purpose	Operation	Description	Reference
Navigating Relationships	get_instance_handle	Returns the DDS_InstanceHandle_t associated with the Entity.	Getting an Entity's Instance Handle (Section 4.1.3 on page 157)
	get_matched_publication_data	Gets information on a publication with a matching Topic and compatible QoS.	Finding Matching Publications (Section 7.3.9.1 on page 489)
	get_matched_publications	Gets a list of publications that have a matching Topic and compatible QoS. These are the publications currently associated with the <i>DataReader</i> .	
	get_matched_publication_participant_data	Gets information on a DomainParticipant of a matching publication.	Finding the Matching Publication's ParticipantBuiltinTopicData (Section 7.3.9.2 on page 490)
	get_subscriber	Gets the <i>Subscriber</i> that created the <i>DataReader</i> .	Finding a DataReader's Related Entities (Section 7.3.9.3 on page 490)
	get_topicdescription	Gets the Topic associated with the <i>DataReader</i> .	
Working with Conditions	create_querycondition	Creates a <i>QueryCondition</i> .	ReadConditions and QueryConditions (Section 4.6.7 on page 195)
	create_readcondition	Creates a <i>ReadCondition</i> .	
	delete_readcondition	Deletes a <i>ReadCondition/QueryCondition</i> attached to the <i>DataReader</i> .	
	delete_contained_entities	Deletes all the <i>ReadConditions/QueryConditions</i> that were created by means of the "create" operations on the <i>DataReader</i> .	Deleting Contained ReadConditions (Section 7.3.3.1 on page 466)
	get_statuscondition	Gets the StatusCondition associated with the Entity.	StatusConditions (Section 4.6.8 on page 197)
Waiting for Historical Data	wait_for_historical_data	Waits until all "historical" (previously sent) data is received. Only valid for Reliable <i>DataReaders</i> with non-VOLATILE DURABILITY.	Waiting for Historical Data (Section 7.3.6 on page 469)

7.3.1 Creating DataReaders

Before you can create a *DataReader*, you need a *DomainParticipant* and a *Topic*.

DataReaders are created by calling **create_datareader()** or **create_datareader_with_profile()**—these operations exist for *DomainParticipants* and *Subscribers*. If you use the *DomainParticipant* to create a

DataReader, it will belong to the implicit *Subscriber* described in [Creating Subscribers Explicitly vs. Implicitly \(Section 7.2.1 on page 444\)](#). If you use a *Subscriber*'s operations to create a *DataReader*, it will belong to that *Subscriber*.

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Note: In the Modern C++ API, *DataReaders* provide constructors whose first argument is a *Subscriber*. The only required arguments are the subscriber and the topic.

```
DDSDataReader* create_datareader(
    DDSTopicDescription *topic,
    const DDS_DataReaderQos &qos,
    DDSDataReaderListener *listener,
    DDS_StatusMask mask);

DDSDataReader * create_datareader_with_profile (
    DDSTopicDescription * topic,
    const char * library_name,

    const char * profile_name,
    DDSDataReaderListener * listener,
    DDS_StatusMask mask)
```

Where:

- topic** The *Topic* to which the *DataReader* is subscribing. This must have been previously created by the same *DomainParticipant*.

- qos** If you want the default QoS settings (described in the API Reference HTML documentation), use `DDS_DATAREADER_QOS_DEFAULT` for this parameter (see [Creating a DataReader with Default QoS Policies \(Section Figure 7.9 on the facing page\)](#)). If you want to customize any of the QoS Policies, supply a QoS structure (see [Setting DataReader QoS Policies \(Section 7.3.8 on page 482\)](#)).

- Note:** If you use `DDS_DATAREADER_QOS_DEFAULT` for the **qos** parameter, it is not safe to create the *DataReader* while another thread may be simultaneously calling the *Subscriber*'s `set_default_datareader_qos()` operation.

- listener** A *DataReader*'s *Listener* is where you define the callback routine that will be notified when new DDS data samples arrive. Connex DDS also uses this *Listener* to notify your application of specific events (status changes) that may occur with respect to the *DataReader*. For more information, see [Setting Up DataReaderListeners \(Section 7.3.4 on page 466\)](#) and [Statuses for DataReaders \(Section 7.3.7 on page 470\)](#).

- The *listener* parameter is optional; you may use NULL instead. In that case, the *Subscriber*'s *Listener* (or if that is NULL, the *DomainParticipant*'s *Listener*) will receive the notifications instead. See [Setting Up DataReaderListeners \(Section 7.3.4 on page 466\)](#) for more on *DataReaderListeners*.

- mask** This bit mask indicates which status changes will cause the *Listener* to be invoked. The bits set in the mask must have corresponding callbacks implemented in the *Listener*. If you use NULL for the *Listener*, use `DDS_STATUS_MASK_NONE` for this parameter. If the *Listener* implements all callbacks, use `DDS_STATUS_MASK_ALL`. For information on statuses, see [Listeners \(Section 4.4 on page 177\)](#).
- library_name** A QoS Library is a named set of QoS profiles. See [URL Groups \(Section 17.8 on page 814\)](#).
- profile_name** A QoS profile groups a set of related QoS, usually one per entity. See [URL Groups \(Section 17.8 on page 814\)](#).

After you create a *DataReader*, you can use it to retrieve received data. See [Using DataReaders to Access Data \(Read & Take\) \(Section 7.4 on page 491\)](#).

Note: When a *DataReader* is created, only those transports already registered are available to the *DataReader*. The built-in transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataReader* is created, or (c) you lookup a built-in *DataReader*, whichever happens first.

[Creating a DataReader with Default QoS Policies \(Section Figure 7.9 below\)](#) shows an example of how to create a *DataReader* with default QoS Policies.

Figure 7.9 Creating a DataReader with Default QoS Policies

```
// MyReaderListener is user defined, extends DDSDataReaderListener
DDSDataReaderListener *reader_listener = new MyReaderListener();
DataReader* reader = subscriber->create_datareader(topic,
    DDS_DATAREADER_QOS_DEFAULT,
    reader_listener, DDS_STATUS_MASK_ALL);
if (reader == NULL) {
    // ... error
}
// narrow it into your specific data type
FooDataReader* foo_reader = FooDataReader::narrow(reader);
```

For more examples on how to create a *DataWriter*, see [Configuring QoS Settings when the DataReader is Created \(Section 7.3.8.1 on page 485\)](#)

7.3.2 Getting All DataReaders

To retrieve all the *DataReaders* created by the *Subscriber*, use the *Subscriber's* `get_all_datareaders()` operation:

```
DDS_ReturnCode_t get_all_datareaders(
    DDS_Subscriber* self,
    struct DDS_DataReaderSeq* readers);
```

In the Modern C++ API, use the freestanding function `rti::sub::find_datareaders()`.

7.3.3 Deleting DataReaders

(Note: in the Modern C++ API, *Entities* are automatically destroyed, see [Creating and Deleting DDS Entities \(Section 4.1.1 on page 153\)](#))

To delete a *DataReader*:

Delete any *ReadConditions* and *QueryConditions* that were created with the *DataReader*. Use the *DataReader*'s `delete_readcondition()` operation to delete them one at a time, or use the `delete_contained_entities()` operation ([Deleting Contained ReadConditions \(Section 7.3.3.1 below\)](#)) to delete them all at the same time.

```
DDS_ReturnCode_t delete_readcondition (DDSReadCondition *condition)
```

Delete the *DataReader* by using the *Subscriber*'s `delete_datareader()` operation ([Deleting Subscribers \(Section 7.2.3 on page 446\)](#)).

Note: A *DataReader* cannot be deleted within its own reader listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

To delete all of a *Subscriber*'s *DataReaders*, use the *Subscriber*'s `delete_contained_entities()` operation (see [Deleting Contained DataReaders \(Section 7.2.3.1 on page 447\)](#)).

7.3.3.1 Deleting Contained ReadConditions

The *DataReader*'s `delete_contained_entities()` operation deletes all the *ReadConditions* and *QueryConditions* ([ReadConditions and QueryConditions \(Section 4.6.7 on page 195\)](#)) that were created by the *DataReader*.

```
DDS_ReturnCode_t delete_contained_entities ()
```

After this operation returns successfully, the application may delete the *DataReader* (see [Deleting DataReaders \(Section 7.3.3 above\)](#)).

7.3.4 Setting Up DataReaderListeners

DataReaders may optionally have *Listeners*. A *DataReaderListener* is a collection of callback methods; these methods are invoked by Connex DDS when DDS data samples are received or when there are status changes for the *DataReader*.

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

If you do not implement a *DataReaderListener*, the associated *Subscriber*'s *Listener* is used instead. If that *Subscriber* does not have a *Listener* either, then the *DomainParticipant*'s *Listener* is used if one exists (see [Setting Up SubscriberListeners \(Section 7.2.6 on page 454\)](#) and [Setting Up DomainParticipantListeners \(Section 8.3.5 on page 560\)](#)).

If you do not require asynchronous notification of data availability or status changes, you do not need to set a *Listener* for the *DataReader*. In that case, you will need to periodically call one of the **read()** or **take()** operations described in [Using DataReaders to Access Data \(Read & Take\)](#) (Section 7.4 on page 491) to access the data that has been received.

Listeners are typically set up when the *DataReader* is created (see [Creating DataReaders](#) (Section 7.3.1 on page 463)). You can also set one up after creation by using the *DataReader*'s **get_listener()** and **set_listener()** operations. Connext DDS will invoke a *DataReader*'s *Listener* to report the status changes listed in [Table 7.4 DataReaderListener Callbacks](#) (if the *Listener* is set up to handle the particular status, see [Setting Up DataReaderListeners](#) (Section 7.3.4 on the previous page)).

Table 7.4 DataReaderListener Callbacks

This DataReaderListener callback...	...is triggered by a change in this status:
on_data_available()	DATA_AVAILABLE Status (Section 7.3.7.1 on page 471)
on_liveliness_changed()	LIVELINESS_CHANGED Status (Section 7.3.7.4 on page 475)
on_requested_deadline_missed()	REQUESTED_DEADLINE_MISSED Status (Section 7.3.7.5 on page 476)
on_requested_incompatible_qos()	REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.7.6 on page 477)
on_sample_lost()	SAMPLE_LOST Status (Section 7.3.7.7 on page 478)
on_sample_rejected()	SAMPLE_REJECTED Status (Section 7.3.7.8 on page 479)
on_subscription_matched()	SUBSCRIPTION_MATCHED Status (Section 7.3.7.9 on page 482)

Note that the same callbacks can be implemented in the *SubscriberListener* or *DomainParticipantListener* instead. There is only one *SubscriberListener* callback that takes precedence over a *DataReaderListener*'s. An **on_data_on_readers()** callback in the *SubscriberListener* (or *DomainParticipantListener*) takes precedence over the **on_data_available()** callback of a *DataReaderListener*.

If the *SubscriberListener* implements an **on_data_on_readers()** callback, it will be invoked instead of the *DataReaderListener*'s **on_data_available()** callback when new data arrives. The **on_data_on_readers()** operation can in turn cause the **on_data_available()** method of the appropriate *DataReaderListener* to be invoked by calling the *Subscriber*'s **notify_datareaders()** operation. For more information on status and *Listeners*, see [Listeners](#) (Section 4.4 on page 177).

[Simple DataReaderListener](#) (Section Figure 7.10 on the next page) shows a *DataReaderListener* that simply prints the data it receives.

Figure 7.10 Simple DataReaderListener

```

class MyReaderListener : public DDSDataReaderListener {
public:
    virtual void on_data_available(DDSDataReader* reader);
    // don't do anything for the other callbacks
};
void MyReaderListener::on_data_available(DDSDataReader* reader)
{
    FooDataReader *Foo_reader = NULL;
    FooSeq data_seq; // In C, sequences have to be initialized
    DDS_SampleInfoSeq info_seq; // before use, see The Sequence Data Structure \(Section 7.4.5 on page 502\)
    DDS_ReturnCode_t retcode;
    int i;
    // Must cast generic reader into reader of specific type
    Foo_reader = FooDataReader::narrow(reader);
    if (Foo_reader == NULL) {
        printf("DataReader narrow error\n");
        return;
    }
    retcode = Foo_reader->take(data_seq, info_seq,
        DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
        DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
    if (retcode == DDS_RETCODE_NO_DATA) {
        return;
    } else if (retcode != DDS_RETCODE_OK) {
        printf("take error %d\n", retcode);
        return;
    }
    for (i = 0; i < data_seq.length(); ++i) {
        // the data may not be valid if the DDS sample is
        // meta information about the creation or deletion
        // of an instance
        if (info_seq[i].valid_data) {
            FooTypeSupport::print_data(&data_seq[i]);
        }
    }
    // Connex DDS gave a pointer to internal memory via
    // take(), must return the memory when finished processing the data
    retcode = Foo_reader->return_loan(data_seq, info_seq);
    if (retcode != DDS_RETCODE_OK) {
        printf("return loan error %d\n", retcode);
    }
}

```

7.3.5 Checking DataReader Status and StatusConditions

You can access individual communication status for a *DataReader* with the operations shown in [Table 7.5 DataReader Status Operations](#).

Table 7.5 DataReader Status Operations

Use this operation...	...to retrieve this status:
<code>get_datareader_cache_status</code>	DATA_READER_CACHE_STATUS (Section 7.3.7.2 on page 471)
<code>get_datareader_protocol_status</code>	DATA_READER_PROTOCOL_STATUS (Section 7.3.7.3 on page 472)
<code>get_matched_publication_datareader_protocol_status</code>	
<code>get_liveliness_changed_status</code>	LIVELINESS_CHANGED Status (Section 7.3.7.4 on page 475)
<code>get_sample_lost_status</code>	SAMPLE_LOST Status (Section 7.3.7.7 on page 478)
<code>get_sample_rejected_status</code>	SAMPLE_REJECTED Status (Section 7.3.7.8 on page 479)
<code>get_requested_deadline_missed_status</code>	REQUESTED_DEADLINE_MISSED Status (Section 7.3.7.5 on page 476)
<code>get_requested_incompatible_qos_status</code>	REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.7.6 on page 477)
<code>get_subscription_match_status</code>	SUBSCRIPTION_MATCHED Status (Section 7.3.7.9 on page 482)
<code>get_status_changes</code>	All of the above
<code>get_statuscondition</code>	See StatusConditions (Section 4.6.8 on page 197)

These methods are useful in the event that no *Listener* callback is set to receive notifications of status changes. If a *Listener* is used, the callback will contain the new status information, in which case calling these methods is unlikely to be necessary.

The `get_status_changes()` operation provides a list of statuses that have changed since the last time the status changes were ‘reset.’ A status change is reset each time the application calls the corresponding `get_*_status()`, as well as each time Connex DDS returns from calling the *Listener* callback associated with that status.

For more on status, see [Setting Up DataReaderListeners](#) (Section 7.3.4 on page 466), [Statuses for DataReaders](#) (Section 7.3.7 on the next page), and [Listeners](#) (Section 4.4 on page 177).

7.3.6 Waiting for Historical Data

The `wait_for_historical_data()` operation waits (blocks) until all "historical" data is received from matched *DataWriters*. "Historical" data means DDS samples that were written before the *DataReader* joined the DDS domain.

This operation is intended only for *DataReaders* that have:

- [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) **kind** set to *TRANSIENT_LOCAL* (*not VOLATILE*)
- [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) **kind** set to RELIABLE

Calling `wait_for_historical_data()` on a non-reliable *DataReader* will always return immediately, since Connex DDS will never deliver historical data to non-reliable *DataReaders*.

As soon as an application enables a non-VOLATILE *DataReader*, it will start receiving both "historical" data as well as any new data written by matching *DataWriters*. If you want the subscribing application to wait until all "historical" data is received, use this operation:

```
DDS_ReturnCode_t wait_for_historical_data (const DDS_Duration_t & max_wait)
```

The `wait_for_historical_data()` operation blocks the calling thread until either all "historical" data is received or the duration specified by the `max_wait` parameter elapses, whichever happens first. A return value of OK indicates that all the "historical" data was received; a return value of TIMEOUT indicates that `max_wait` elapsed before all the data was received.

`wait_for_historical_data()` will return immediately if no *DataWriters* have been discovered at the time the operation is called. Therefore it is advisable to make sure at least one *DataWriter* has been discovered before calling this operation; one way to do this is to use `get_subscription_matched_status()`, like this:

```
while (1) {
    DDS_SubscriptionMatchedStatus status;
    MyType_reader->get_subscription_matched_status(status);
    if (status.current_count > 0) { break; }
    NDDSUtility::sleep(sleep_period);
}
```

7.3.7 Statuses for DataReaders

There are several types of statuses available for a *DataReader*. You can use the `get*_status()` operations ([Checking DataReader Status and StatusConditions \(Section 7.3.5 on page 468\)](#)) to access and reset them, use a *DataReaderListener* ([Setting Up DataReaderListeners \(Section 7.3.4 on page 466\)](#)) to listen for changes in their values (for those statuses that have Listeners), or use a *StatusCondition* and a *WaitSet* ([StatusConditions \(Section 4.6.8 on page 197\)](#)) to wait for changes. Each status has an associated data structure and is described in more detail in the following sections.

- [DATA_AVAILABLE Status \(Section 7.3.7.1 on the facing page\)](#)
- [DATA_READER_CACHE_STATUS \(Section 7.3.7.2 on the facing page\)](#)
- [DATA_READER_PROTOCOL_STATUS \(Section 7.3.7.3 on page 472\)](#)
- [LIVELINESS_CHANGED Status \(Section 7.3.7.4 on page 475\)](#)
- [REQUESTED_DEADLINE_MISSED Status \(Section 7.3.7.5 on page 476\)](#)

- [REQUESTED_INCOMPATIBLE_QOS Status \(Section 7.3.7.6 on page 477\)](#)
- [SAMPLE_LOST Status \(Section 7.3.7.7 on page 478\)](#)
- [SAMPLE_REJECTED Status \(Section 7.3.7.8 on page 479\)](#)
- [SUBSCRIPTION_MATCHED Status \(Section 7.3.7.9 on page 482\)](#)

7.3.7.1 DATA_AVAILABLE Status

This status indicates that new data is available for the *DataReader*. In most cases, this means that one new DDS sample has been received. However, there are situations in which more than one DDS samples for the *DataReader* may be received before the **DATA_AVAILABLE** status changes. For example, if the *DataReader* has the [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) set to be non-**VOLATILE**, then the *DataReader* may receive a batch of old DDS data samples all at once. Or if data is being received reliably from *DataWriters*, Connex DDS may present several DDS samples of data simultaneously to the *DataReader* if they have been originally received out of order.

A change to this status also means that the **DATA_ON_READERS** status is changed for the *DataReader's Subscriber*. This status is reset when you call **read()**, **take()**, or one of their variations.

Unlike most other statuses, this status (as well as **DATA_ON_READERS** for *Subscribers*) is a *read communication status*. See [Statuses for Subscribers \(Section 7.2.9 on page 458\)](#) and [Types of Communication Status \(Section 4.3.1 on page 170\)](#) for more information on read communication statuses.

The *DataReaderListener's on_data_available()* callback is invoked when this status changes, unless the *SubscriberListener (Setting Up SubscriberListeners (Section 7.2.6 on page 454))* or *DomainParticipantListener (Setting Up DomainParticipantListeners (Section 8.3.5 on page 560))* has implemented an **on_data_on_readers()** callback. In that case, **on_data_on_readers()** will be invoked instead.

7.3.7.2 DATA_READER_CACHE_STATUS

This status keeps track of the number of DDS samples in the reader's cache.

This status does not have an associated Listener. You can access this status by calling the *DataReader's get_datareader_cache_status()* operation, which will return the status structure described in [Table 7.6 DDS_DataReaderCacheStatus](#); this operation will also reset the status so it is no longer considered “changed.”

Table 7.6 DDS_DataReaderCacheStatus

Type	Field Name	Description
DDS_Long	sample_count_peak	Highest number of DDS samples in the <i>DataReader's</i> queue over the lifetime of the <i>DataReader</i> .

Table 7.6 DDS_DataReaderCacheStatus

Type	Field Name	Description
DDS_Long	sample_count	Current number of DDS samples in the <i>DataReader's</i> queue. Includes DDS samples that may not yet be available to be read or taken by the user due to DDS samples being received out of order or settings in the PRESENTATION QosPolicy (Section 6.4.6 on page 330) .

7.3.7.3 DATA_READER_PROTOCOL_STATUS

The status of a *DataReader's* internal protocol related metrics (such as the number of DDS samples received, filtered, rejected) and the status of wire protocol traffic. The structure for this status appears in [Table 7.7 DDS_DataReaderProtocolStatus](#).

This status does not have an associated Listener. You can access this status by calling the following operations on the *DataReader* (which return the status structure described in [Table 7.7 DDS_DataReaderProtocolStatus](#)):

get_datareader_protocol_status() returns the sum of the protocol status for all the matched publications for the *DataReader*.

get_matched_publication_datareader_protocol_status() returns the protocol status of a particular matched publication, identified by a **publication_handle**.

The **get_*_status()** operations also reset the related status so it is no longer considered “changed.”

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted. If you try to get the matched subscription status for a remote entity that is no longer alive, the ‘get status’ call will return an error.

Table 7.7 DDS_DataReaderProtocolStatus

Type	Field Name	Description
DDS_LongLong	received_sample_count	The number of DDS samples from a remote <i>DataWriter</i> received for the first time by a local <i>DataReader</i> .
	received_sample_count_change	The incremental change in the number of DDS samples from a remote <i>DataWriter</i> received for the first time by a local <i>DataReader</i> since the last time the status was read.
	received_sample_bytes	The number of bytes of DDS samples from a remote <i>DataWriter</i> received for the first time by a local <i>DataReader</i> .
	received_sample_bytes_change	The incremental change in the number of bytes of DDS samples from a remote <i>DataWriter</i> received for the first time by a local <i>DataReader</i> since the last time the status was read.
DDS_LongLong	duplicate_sample_count	The number of DDS samples from a remote <i>DataWriter</i> received, not for the first time, by a local <i>DataReader</i> .
	duplicate_sample_count_change	The incremental change in the number of DDS samples from a remote <i>DataWriter</i> received, not for the first time, by a local <i>DataReader</i> since the last time the status was read.
	duplicate_sample_bytes	The number of bytes of DDS samples from a remote <i>DataWriter</i> received, not for the first time, by a local <i>DataReader</i> .
	duplicate_sample_bytes_change	The incremental change in the number of bytes of DDS samples from a remote <i>DataWriter</i> received, not for the first time, by a local <i>DataReader</i> since the last time the status was read.
DDS_LongLong	filtered_sample_count	The number of DDS samples filtered by the local <i>DataReader</i> due to ContentFilteredTopics or Time-Based Filter.
	filtered_sample_count_change	The incremental change in the number of DDS samples filtered by the local <i>DataReader</i> due to Content-FilteredTopics or Time-Based Filter since the last time the status was read.
	filtered_sample_bytes	The number of bytes of DDS samples filtered by the local <i>DataReader</i> due to ContentFilteredTopics or Time-Based Filter.
	filtered_sample_bytes_change	The incremental change in the number of bytes of DDS samples filtered by the local <i>DataReader</i> due to ContentFilteredTopics or Time-Based Filter since the last time the status was read.

Table 7.7 DDS_DataReaderProtocolStatus

Type	Field Name	Description
DDS_LongLong	received_heartbeat_count	The number of Heartbeats from a remote <i>DataWriter</i> received by a local <i>DataReader</i> .
	received_heartbeat_count_change	The incremental change in the number of Heartbeats from a remote <i>DataWriter</i> received by a local <i>DataReader</i> since the last time the status was read.
	received_heartbeat_bytes	The number of bytes of Heartbeats from a remote <i>DataWriter</i> received by a local <i>DataReader</i> .
	received_heartbeat_bytes_change	The incremental change in the number of bytes of Heartbeats from a remote <i>DataWriter</i> received by a local <i>DataReader</i> since the last time the status was read.
DDS_LongLong	sent_ack_count	The number of ACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> .
	sent_ack_count_change	The incremental change in the number of ACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> since the last time the status was read.
	sent_ack_bytes	The number of bytes of ACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> .
	sent_ack_bytes_change	The incremental change in the number of bytes of ACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> since the last time the status was read.
DDS_LongLong	sent_nack_count	The number of NACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> .
	sent_nack_count_change	The incremental change in the number of NACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> since the last time the status was read.
	sent_nack_bytes	The number of bytes of NACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> .
	sent_nack_bytes_change	The incremental change in the number of bytes of NACKs sent from a local <i>DataReader</i> to a matching remote <i>DataWriter</i> since the last time the status was read.

Table 7.7 DDS_DataReaderProtocolStatus

Type	Field Name	Description
DDS_LongLong	received_gap_count	The number of GAPS received from remote <i>DataWriter</i> to this <i>DataReader</i> .
	received_gap_count_change	The incremental change in the number of GAPS received from remote <i>DataWriter</i> to this <i>DataReader</i> since the last time the status was read.
	received_gap_bytes	The number of bytes of GAPS received from remote <i>DataWriter</i> to this <i>DataReader</i> .
	received_gap_bytes_change	The incremental change in the number of bytes of GAPS received from remote <i>DataWriter</i> to this <i>DataReader</i> since the last time the status was read.
DDS_LongLong	rejected_sample_count	The number of times a DDS sample is rejected for unanticipated reasons in the receive path.
	rejected_sample_count_change	The incremental change in the number of times a DDS sample is rejected for unanticipated reasons in the receive path since the last time the status was read.
DDS_SequenceNumber_t	first_available_sample_sequence_number	Sequence number of the first available DDS sample in a matched <i>DataWriter's</i> reliability queue. Applicable only when retrieving matched <i>DataWriter</i> statuses.
	last_available_sample_sequence_number	Sequence number of the last available DDS sample in a matched <i>DataWriter's</i> reliability queue. Applicable only when retrieving matched <i>DataWriter</i> statuses.
	last_committed_sample_sequence_number	Sequence number of the last committed DDS sample (i.e. available to be read or taken) in a matched <i>DataWriter's</i> reliability queue. Applicable only when retrieving matched <i>DataWriter</i> statuses. For best-effort <i>DataReaders</i> , this is the sequence number of the latest DDS sample received. For reliable <i>DataReaders</i> , this is the sequence number of the latest DDS sample that is available to be read or taken from the <i>DataReader's</i> queue.
DDS_Long	uncommitted_sample_count	Number of received DDS samples that are not yet available to be read or taken due to being received out of order. Applicable only when retrieving matched <i>DataWriter</i> statuses.

7.3.7.4 LIVELINESS_CHANGED Status

This status indicates that the liveliness of one or more matched *DataWriters* has changed (i.e., one or more *DataWriters* has become alive or not alive). The mechanics of determining liveliness between a *DataWriter* and a *DataReader* is specified in their [LIVELINESS QosPolicy \(Section 6.5.13 on page 382\)](#).

The structure for this status appears in [Table 7.8 DDS_LivelinessChangedStatus](#).

Table 7.8 DDS_LivelinessChangedStatus

Type	Field Name	Description
DDS_Long	alive_count	Number of matched <i>DataWriters</i> that are currently alive.
	not_alive_count	Number of matched <i>DataWriters</i> that are not currently alive.
	alive_count_change	The change in the alive_count since the last time the <i>Listener</i> was called or the status was read.
	not_alive_count_change	The change in the not_alive_count since the last time the <i>Listener</i> was called or the status was read.
DDS_InstanceHandle_t	last_publication_handle	A handle to the last <i>DataWriter</i> to change its liveliness.

The *DataReaderListener*'s **on_liveliness_changed()** callback may be called for the following reasons:

- Liveliness is truly lost—a DDS sample has not been received within the time-frame specified in the [LIVELINESS QoS Policy \(Section 6.5.13 on page 382\)](#) **lease_duration**.
- Liveliness is recovered after being lost.
- A new matching entity has been discovered.
- A QoS has changed such that a pair of matching entities are no longer matching (such as a change to the [PartitionQoS Policy](#)). In this case, the middleware will no longer keep track of the entities' liveliness. Furthermore:
 - If liveliness was maintained: **alive_count** will decrease and **not_alive_count** will remain the same.
 - If liveliness had been lost: **alive_count** will remain the same and **not_alive_count** will decrease.

You can also retrieve the value by calling the *DataReader*'s **get_liveliness_changed_status()** operation; this will also reset the status so it is no longer considered “changed.”

This status is reciprocal to the [RELIABLE_READER_ACTIVITY_CHANGED Status \(DDS Extension\) \(Section 6.3.6.9 on page 281\)](#) for a *DataWriter*.

7.3.7.5 REQUESTED_DEADLINE_MISSED Status

This status indicates that the *DataReader* did not receive a new DDS sample for an data-instance within the time period set in the *DataReader*'s [DEADLINE QoS Policy \(Section 6.5.5 on page 363\)](#). For non-

keyed Topics, this simply means that the *DataReader* did not receive data within the DEADLINE period. For keyed *Topics*, this means that for one of the data-instances that the *DataReader* was receiving, it has not received a new DDS sample within the DEADLINE period. For more information about keys and instances, see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#).

The structure for this status appears in [Table 7.9 DDS_RequestedDeadlineMissedStatus](#).

Table 7.9 DDS_RequestedDeadlineMissedStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times that the deadline was violated for any instance read by the <i>DataReader</i> .
	total_count_change	The change in total_count since the last time the <i>Listener</i> was called or the status was read.
DDS_InstanceHandle_t	last_instance_handle	Handle to the last data-instance in the <i>DataReader</i> for which a requested deadline was missed.

The *DataReaderListener*'s **on_requested_deadline_missed()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataReader*'s **get_requested_deadline_missed_status()** operation; this will also reset the status so it is no longer considered “changed.”

7.3.7.6 REQUESTED_INCOMPATIBLE_QOS Status

A change to this status indicates that the *DataReader* discovered a *DataWriter* for the same *Topic*, but that *DataReader* had requested QoS settings incompatible with this *DataWriter*'s offered QoS.

The structure for this status appears in [Table 7.10 DDS_RequestedIncompatibleQosStatus](#).

Table 7.10 DDS_RequestedIncompatibleQosStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the <i>DataReader</i> discovered a <i>DataWriter</i> for the same <i>Topic</i> with an offered QoS that is incompatible with that requested by the <i>DataReader</i> .
DDS_Long	total_count_change	The change in total_count since the last time the <i>Listener</i> was called or the status was read.
DDS_QosPolicyId_t	last_policy_id	The ID of the QoSPolicy that was found to be incompatible the last time an incompatibility was detected. (Note: if there are multiple incompatible policies, only one of them is reported here.)
DDS_QosPolicyCountSeq	policies	A list containing—for each policy—the total number of times that the <i>DataReader</i> discovered a <i>DataWriter</i> for the same <i>Topic</i> with a offered QoS that is incompatible with that requested by the <i>DataReader</i> .

The *DataReaderListener*'s `on_requested_incompatible_qos()` callback is invoked when this status changes. You can also retrieve the value by calling the *DataReader*'s `get_requested_incompatible_qos_status()` operation; this will also reset the status so it is no longer considered “changed.”

7.3.7.7 SAMPLE_LOST Status

This status indicates that one or more DDS samples written by a matched *DataWriter* have failed to be received.

For a *DataReader*, when there are insufficient resources to accept incoming DDS samples of data, DDS samples may be dropped by the receiving application. Those DDS samples are considered to be REJECTED (see [SAMPLE_REJECTED Status \(Section 7.3.7.8 on the facing page\)](#)). But *DataWriters* are limited in the number of published DDS data samples that they can store, so that if a *DataWriter* continues to publish DDS data samples, new data may overwrite old data that have not yet been received by the *DataReader*. The DDS samples that are overwritten can never be resent to the *DataReader* and thus are considered to be *lost*.

This status applies to reliable *and* best-effort *DataReaders*, see the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#).

The structure for this status appears in [Table 7.11 DDS_SampleLostStatus](#).

Table 7.11 DDS_SampleLostStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative count of all the DDS samples that have been lost, across all instances of data written for the <i>Topic</i> .
	total_count_change	The incremental number of DDS samples lost since the last time the <i>Listener</i> was called or the status was read.
DDS_SampleLostStatusKind	last_reason	The reason the last DDS sample was lost. See Table 7.12 DDS_SampleLostStatusKind .

The reason the DDS sample was lost appears in the **last_reason** field. The possible values are listed in [Table 7.12 DDS_SampleLostStatusKind](#).

Table 7.12 DDS_SampleLostStatusKind

Reason Kind	Description
NOT_LOST	The DDS sample was not lost.
LOST_BY_AVAILABILITY_WAITING_TIME	AvailabilityQosPolicy's max_data_availability_waiting_time expired.

Table 7.12 DDS_SampleLostStatusKind

Reason Kind	Description
LOST_BY_INCOMPLETE_COHERENT_SET	A DDS sample is lost because it is part of an incomplete coherent set.
LOST_BY_INSTANCES_LIMIT	A resource limit on the number of instances was reached.
LOST_BY_LARGE_COHERENT_SET	A DDS sample is lost because it is part of a large coherent set.
LOST_BY_REMOTE_WRITER_SAMPLES_PER_VIRTUAL_QUEUE_LIMIT"	A resource limit on the number of DDS samples published by a remote writer on behalf of a virtual writer that a <i>DataReader</i> may store was reached.
LOST_BY_REMOTE_WRITERS_PER_INSTANCE_LIMIT	A resource limit on the number of remote writers for a single instance from which a <i>DataReader</i> may read was reached.
LOST_BY_REMOTE_WRITERS_PER_SAMPLE_LIMIT	A resource limit on the number of remote writers per DDS sample was reached.
LOST_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT	A resource limit on the number of DDS samples from a given remote writer that a <i>DataReader</i> may store was reached.
LOST_BY_VIRTUAL_WRITERS_LIMIT	A resource limit on the number of virtual writers from which a <i>DataReader</i> may read was reached.
LOST_BY_WRITER	A <i>DataWriter</i> removed the DDS sample before being received by the <i>DataReader</i> .

The *DataReaderListener*'s **on_sample_lost()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataReader*'s **get_sample_lost_status()** operation; this will also reset the status so it is no longer considered "changed."

7.3.7.8 SAMPLE_REJECTED Status

This status indicates that one or more DDS samples received from a matched *DataWriter* have been dropped by the *DataReader* because a resource limit would have been exceeded. For example, if the receive queue is full, the number of DDS samples in the queue is equal to the **max_samples** parameter of the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#).

The structure for this status appears in [Table 7.13 DDS_SampleRejectedStatus](#). The reason the DDS sample was rejected appears in the **last_reason** field. The possible values are listed in [Table 7.14 DDS_SampleRejectedStatusKind](#).

Table 7.13 DDS_SampleRejectedStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative count of all the DDS samples that have been rejected by the <i>DataReader</i> .
	total_count_change	The incremental number of DDS samples rejected since the last time the <i>Listener</i> was called or the status was read.
	current_count	The current number of writers with which the <i>DataReader</i> is matched.
	current_count_change	The change in current_count since the last time the <i>Listener</i> was called or the status was read.
DDS_SampleRejectedStatusKind	last_reason	Reason for rejecting the last DDS sample. See Table 7.14 DDS_SampleRejectedStatusKind .
DDS_InstanceHandle_t	last_instance_handle	Handle to the data-instance for which the last DDS sample was rejected.

Table 7.14 DDS_SampleRejectedStatusKind

Reason Kind	Description	Related QosPolicy
DDS_NOT_REJECTED	DDS sample was accepted.	
DDS_REJECTED_BY_INSTANCES_LIMIT	A resource limit on the number of instances that can be handled at the same time by the <i>DataReader</i> was reached.	RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405)
DDS_REJECTED_BY_REMOTE_WRITERS_LIMIT	A resource limit on the number of <i>DataWriters</i> from which a <i>DataReader</i> may read was reached.	DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2 on page 517)
DDS_REJECTED_BY_REMOTE_WRITERS_PER_INSTANCE_LIMIT	A resource limit on the number of <i>DataWriters</i> for a single instance from which a <i>DataReader</i> may read was reached.	

Table 7.14 DDS_SampleRejectedStatusKind

Reason Kind	Description	Related QosPolicy
DDS_REJECTED_BY_SAMPLES_LIMIT	A resource limit on the total number of DDS samples was reached.	RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405)
DDS_REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT	A resource limit on the number of DDS samples per instance was reached.	
DDS_REJECTED_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT	A resource limit on the number of DDS samples that a <i>DataReader</i> may store from a specific <i>DataWriter</i> was reached.	DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2 on page 517)
DDS_REJECTED_BY_VIRTUAL_WRITERS_LIMIT	A resource limit on the number of virtual writers from which a <i>DataReader</i> may read was reached.	
DDS_REJECTED_BY_REMOTE_WRITERS_PER_SAMPLE_LIMIT	A resource limit on the number of remote writers per DDS sample was reached.	
DDS_REJECTED_BY_REMOTE_WRITER_SAMPLES_PER_VIRTUAL_QUEUE_LIMIT	A resource limit on the number of DDS samples published by a remote writer on behalf of a virtual writer that a <i>DataReader</i> may store was reached.	

The *DataReaderListener*'s **on_sample_rejected()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataReader*'s **get_sample_rejected_status()** operation; this will also reset the status so it is no longer considered “changed.”

7.3.7.9 SUBSCRIPTION_MATCHED Status

A change to this status indicates that the *DataReader* discovered a matching *DataWriter*. A ‘match’ occurs only if the *DataReader* and *DataWriter* have the same *Topic*, same data type (implied by having the same *Topic*), and compatible QoS Policies. In addition, if user code has directed Connex DDS to ignore certain *DataWriters*, then those *DataWriters* will never be matched. See [Ignoring Publications and Subscriptions \(Section 16.4.2 on page 786\)](#) for more on setting up a *DomainParticipant* to ignore specific *DataWriters*.

The structure for this status appears in [Table 7.15 DDS_SubscriptionMatchedStatus](#).

Table 7.15 DDS_SubscriptionMatchedStatus

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the <i>DataReader</i> discovered a "match" with a <i>DataWriter</i> .
	total_count_change	The change in total_count since the last time the <i>Listener</i> was called or the status was read.
	current_count	The number of <i>DataWriters</i> currently matched to the concerned <i>DataReader</i> .
	current_count_change	The change in current_count since the last time the listener was called or the status was read.
	current_count_peak	The highest value that current_count has reached until now.
DDS_InstanceHandle_t	last_publication_handle	Handle to the last <i>DataWriter</i> that matched the <i>DataReader</i> causing the status to change.

The *DataReaderListener*'s **on_subscription_matched()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataReader*'s **get_subscription_match_status()** operation; this will also reset the status so it is no longer considered “changed.”

7.3.8 Setting DataReader QoS Policies

A *DataReader*'s QoS Policies control its behavior. Think of QoS Policies as the ‘properties’ for the *DataReader*. The **DDS_DataReaderQos** structure has the following format:

```
struct DDS_DataReaderQos {
    DDS_DurabilityQosPolicy      durability;
    DDS_DeadlineQosPolicy       deadline;
    DDS_LatencyBudgetQosPolicy  latency_budget;
    DDS_LivelinessQosPolicy     liveliness;
    DDS_ReliabilityQosPolicy     reliability;
```

```

DDS_DestinationOrderQosPolicy      destination_order;
DDS_HistoryQosPolicy               history;
DDS_ResourceLimitsQosPolicy        resource_limits;
DDS_UserDataQosPolicy              user_data;
DDS_TimeBasedFilterQosPolicy       time_based_filter;
DDS_ReaderDataLifecycleQosPolicy   reader_data_lifecycle;
DDS_TransportPriorityQosPolicy      transport_priority;
DDS_TypeConsistencyEnforcementQosPolicy type_consistency;
// Extensions to the DDS standard:
DDS_DataReaderResourceLimitsQosPolicy reader_resource_limits;
DDS_DataReaderProtocolQosPolicy     protocol;
DDS_TransportSelectionQosPolicy     transport_selection;
DDS_TransportUnicastQosPolicy       unicast;
DDS_TransportMulticastQosPolicy     multicast;
DDS_PropertyQosPolicy              property;
DDS_ServiceQosPolicy               service;
DDS_AvailabilityQosPolicy           availability;
DDS_EntityNameQosPolicy             subscription_name;
DDS_TypeSupportQosPolicy            type_support;
};

```

Note: `set_qos()` cannot always be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

[Table 7.16 DataReader QoS Policies](#) summarizes the meaning of each policy. (They appear alphabetically in the table.) For information on *why* you would want to change a particular QoS Policy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 7.16 DataReader QoS Policies

QoS Policy	Description
Availability	<p>This QoS policy is used in the context of two features:</p> <p>For a Collaborative DataWriter, specifies the group of <i>DataWriters</i> expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip DDS samples.</p> <p>For a Durable Subscription, configures a set of Durable Subscriptions on a <i>DataWriter</i>.</p> <p>See AVAILABILITY QoS Policy (DDS Extension) (Section 6.5.1 on page 337)</p>
DataReaderProtocol	<p>This QoS Policy configures the DDS on-the-network protocol, RTPS. See DATA_READER_PROTOCOL QoS Policy (DDS Extension) (Section 7.6.1 on page 511).</p>
DataReaderResourceLimits	<p>Various settings that configure how <i>DataReaders</i> allocate and use physical memory for internal resources. See DATA_READER_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 7.6.2 on page 517).</p>
Deadline	<p>For a <i>DataReader</i>, specifies the maximum expected elapsed time between arriving DDS data samples.</p> <p>For a <i>DataWriter</i>, specifies a commitment to publish DDS samples with no greater elapsed time between them.</p> <p>See DEADLINE QoS Policy (Section 6.5.5 on page 363).</p>

Table 7.16 DataReader QoS Policies

QoS Policy	Description
DestinationOrder	Controls how Connex DDS will deal with data sent by multiple <i>DataWriters</i> for the same topic. Can be set to "by reception timestamp" or to "by source timestamp". See DESTINATION_ORDER QoS Policy (Section 6.5.6 on page 365) .
Durability	Specifies whether or not Connex DDS will store and deliver data that were previously published to new <i>DataReaders</i> . See DURABILITY QoS Policy (Section 6.5.7 on page 368) .
EntityName	Assigns a name to a <i>DataReader</i> . See ENTITY_NAME QoS Policy (DDS Extension) (Section 6.5.9 on page 374) .
History	Specifies how much data must to stored by Connex DDS for the <i>DataWriter</i> or <i>DataReader</i> . This QoS Policy affects the RELIABILITY QoS Policy (Section 6.5.19 on page 400) as well as the DURABILITY QoS Policy (Section 6.5.7 on page 368) . See HISTORY QoS Policy (Section 6.5.10 on page 376) .
LatencyBudget	Suggestion to Connex DDS on how much time is allowed to deliver data. See LATENCYBUDGET QoS Policy (Section 6.5.11 on page 380) .
Liveliness	Specifies and configures the mechanism that allows <i>DataReaders</i> to detect when <i>DataWriters</i> become disconnected or "dead." See LIVELINESS QoS Policy (Section 6.5.13 on page 382) .
Property	Stores name/value (string) pairs that can be used to configure certain parameters of Connex DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See PROPERTY QoS Policy (DDS Extension) (Section 6.5.17 on page 394) .
ReaderDataLifeCycle	Controls how a <i>DataReader</i> manages the lifecycle of the data that it has received. See READER_DATA_LIFECYCLE QoS Policy (Section 7.6.3 on page 523) .
Reliability	Specifies whether or not Connex DDS will deliver data reliably. See RELIABILITY QoS Policy (Section 6.5.19 on page 400) .
ResourceLimits	Controls the amount of physical memory allocated for entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See RESOURCE_LIMITS QoS Policy (Section 6.5.20 on page 405) .
Service	Intended for use by RTI infrastructure services. User applications should not modify its value. See SERVICE QoS Policy (DDS Extension) (Section 6.5.21 on page 408) .
TimeBasedFilter	Set by a <i>DataReader</i> to limit the number of new data values received over a period of time. See TIME_BASED_FILTER QoS Policy (Section 7.6.4 on page 526) .
TransportMulticast	Specifies the multicast address on which a <i>DataReader</i> wants to receive its data. Can specify a port number as well as a subset of the available transports with which to receive the multicast data. See TRANSPORT_MULTICAST QoS Policy (DDS Extension) (Section 7.6.5 on page 529) .

Table 7.16 DataReader QoS Policies

QoS Policy	Description
TransportPriority	Set on a <i>DataReader</i> to tell Connex DDS that the data being sent has a different "priority" than other data. For <i>DataReaders</i> , the data being sent refers to ACKNACK messages. See TRANSPORT_PRIORITY QoS Policy (Section 6.5.22 on page 409) .
TransportSelection	Allows you to select which physical transports a <i>DataWriter</i> or <i>DataReader</i> may use to send or receive its data. See TRANSPORT_SELECTION QoS Policy (DDS Extension) (Section 6.5.23 on page 411) .
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See TRANSPORT_UNICAST QoS Policy (DDS Extension) (Section 6.5.24 on page 412) .
TypeSupport	Used to attach application-specific value(s) to a <i>DataWriter</i> or <i>DataReader</i> . These values are passed to the serialization or deserialization routine of the associated data type. See TYPESUPPORT QoS Policy (DDS Extension) (Section 6.5.25 on page 416) .
TypeConsistencyEnforcement	Defines rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it. See TYPE_CONSISTENCY_ENFORCEMENT QoS Policy (Section 7.6.6 on page 532) .
UserData	Along with Topic Data QoS Policy and Group Data QoS Policy, used to attach a buffer of bytes to Connex DDS's discovery meta-data. See USER_DATA QoS Policy (Section 6.5.26 on page 417) .

For a *DataReader* to communicate with a *DataWriter*, their corresponding QoS Policies must be compatible. For QoS Policies that apply both to the *DataWriter* and the *DataReader*, the setting in the *DataWriter* is considered what the *DataWriter* “offers” and the setting in the *DataReader* is what the *DataReader* “requests.” Compatibility means that what is offered by the *DataWriter* equals or surpasses what is requested by the *DataReader*. See [QoS Requested vs. Offered Compatibility—the RxO Property \(Section 4.2.1 on page 167\)](#).

Some of the policies may be changed after the *DataReader* has been created. This allows the application to modify the behavior of the *DataReader* while it is in use. To modify the QoS of an existing *DataReader*, use the `get_qos()` and `set_qos()` operations on the *DataReader*. This is a general pattern for all Entities, described in more detail in [Changing the QoS for an Existing Entity \(Section 4.1.7.3 on page 161\)](#).

7.3.8.1 Configuring QoS Settings when the DataReader is Created

As described in [Creating DataReaders \(Section 7.3.1 on page 463\)](#), there are different ways to create a *DataReader*, depending on how you want to specify its QoS (with or without a QoS Profile).

In [Creating a DataReader with Default QoS Policies \(Section Figure 7.9 on page 465\)](#), we saw an example of how to create a *DataReader* with default QoS Policies by using the special constant, `DDS_DATAREADER_QOS_DEFAULT`, which indicates that the default QoS values for a *DataReader* should be used. The default *DataReader* QoS values are configured in the Publisher or

DomainParticipant, you can change them with `set_default_datareader_qos()` or `set_default_datareader_qos_with_profile()`. Then any *DataReaders* created with the *Subscriber* will use the new default values. As described in [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#), this is a general pattern that applies to the construction of all *Entities*.

To create a *DataReader* with non-default QoS without using a QoS Profile, see the example code in [Figure 7.11 Creating a DataReader with Modified QoS Policies \(not from a profile\) below](#). It uses the *Publisher's* `get_default_reader_qos()` method to initialize a `DDS_DataReaderQos` structure. Then, the policies are modified from their default values before the structure is used in the `create_datareader()` method.

You can also create a *DataReader* and specify its QoS settings via a QoS Profile. To do so, you will call `create_datareader_with_profile()`, as seen in [Figure 7.12 Creating a DataReader with a QoS Profile on the facing page](#).

If you want to use a QoS profile, but then make some changes to the QoS before creating the *DataReader*, call `get_datareader_qos_from_profile()` and `create_datareader()` as seen in [Figure 7.13 Getting QoS Values from a Profile, Changing QoS Values, Creating a DataReader with Modified QoS Values on the facing page](#).

For more information, see [Creating DataReaders \(Section 7.3.1 on page 463\)](#) and [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 7.11 Creating a DataReader with Modified QoS Policies (not from a profile)

```
DDS_DataReaderQos reader_qos;1
// initialize reader_qos with default values
subscriber->get_default_datareader_qos(reader_qos);
// make QoS changes here
reader_qos.history.depth = 5;
// Create the reader with modified qos
DDSDataReader * reader = subscriber->create_datareader(
    topic, reader_qos, NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL) {
    // ... error
}
// narrow it for your specific data type
FooDataReader* foo_reader = FooDataReader::narrow(reader);
```

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

Figure 7.12 Creating a DataReader with a QoS Profile

```

// Create the datareader
DDSDataReader * reader =
    subscriber->create_datareader_with_profile(
        topic, "MyReaderLibrary", "MyReaderProfile",
        NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL) {
    // ... error
};
// narrow it for your specific data type
FooDataReader* foo_reader = FooDataReader::narrow(reader);

```

Figure 7.13 Getting QoS Values from a Profile, Changing QoS Values, Creating a DataReader with Modified QoS Values

```

DDS_DataReaderQos reader_qos;1
// Get reader QoS from profile
retcode = factory->get_datareader_qos_from_profile(reader_qos,
    "ReaderProfileLibrary", "ReaderProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
reader_qos.history.depth = 5;
DDSDataReader * reader = subscriber->create_datareader(topic, reader_qos,
    NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL) {
    // handle error
}

```

7.3.8.2 Comparing QoS Values

The `equals()` operation compares two *DataReader*'s `DDS_DataWriterQoS` structures for equality. It takes two parameters for the two *DataReader*'s QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

7.3.8.3 Changing QoS Settings After DataReader Has Been Created

There are 2 ways to change an existing *DataReader*'s QoS after it has been created—again depending on whether or not you are using a QoS Profile.

¹Note: In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

- To change QoS programmatically (that is, without using a QoS Profile), use `get_qos()` and `set_qos()`. See the example code in [Figure 7.14 Changing the QoS of an Existing DataReader \(without a QoS Profile\) below](#). It retrieves the current values by calling the DataReader's `get_qos()` operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QoS Policies cannot be changed after the DataReader has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.
- You can also change a DataReader's (and all other Entities') QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see [Figure 7.15 Changing the QoS of an Existing DataReader with a QoS Profile below](#). For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 7.14 Changing the QoS of an Existing DataReader (without a QoS Profile)

```
// Get current QoS
if (datareader->get_qos(reader_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
reader_qos.history.depth = 5;
// Set the new QoS
if (datareader->set_qos(reader_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 7.15 Changing the QoS of an Existing DataReader with a QoS Profile

```
retcode = datareader->set_qos_with_profile(
    "ReaderProfileLibrary", "ReaderProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

7.3.8.4 Using a Topic's QoS to Initialize a DataWriter's QoS

Several *DataReader* QoS Policies can also be found in the QoS Policies for *Topics* (see [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#)). The QoS Policies set in the Topic do not directly affect the *DataReaders* (or *DataWriters*) that use that *Topic*. In many ways, some QoS Policies are a *Topic*-level concept, even though the DDS standard allows you to set different values for those policies for different *DataReaders* and *DataWriters* of the same *Topic*. Thus, the policies in the **DDS_TopicQos** structure exist as a way to help centralize and annotate the intended or suggested values of those QoSs. Connex DDS does not check to see if the actual policies set for a *DataReader* is aligned with those set in the *Topic* to which it is bound.

There are many ways to use the QoS Policies' values set in the *Topic* when setting the QoS Policies' values in a *DataReader*. The most straight forward way is to get the values of policies directly from the *Topic* and use them in the policies for the *DataReader*. [Figure 6.21 Copying Selected QoS from a Topic when Creating a DataWriter on page 307](#) shows an example of how to this for a *DataWriter*; the pattern applies to *DataReaders* as well.

The *Subscriber's* `copy_from_topic_qos()` operation can be used to copy all the common policies from the *Topic* QoS to a **DataReaderQoS**, as illustrated in [Figure 6.22 Copying all QoS from a Topic when Creating a DataWriter on page 308](#) for *DataWriters*.

The special macro, **DDS_DATAREADER_QOS_USE_TOPIC_QOS**, can be used to indicate that the *DataReader* should be created with the QoS that results from modifying the default *DataReader* QoS with the values specified by the *Topic*. See [Figure 6.23 Combining Default Topic and DataWriter QoS \(Option 1\) on page 309](#) and [Figure 6.24 Combining Default Topic and DataWriter QoS \(Option 2\) on page 309](#) for examples involving *DataWriters*. The same pattern applies to *DataReaders*. For more information on the use and manipulation of QoS, see [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#).

7.3.9 Navigating Relationships Among Entities

7.3.9.1 Finding Matching Publications

The following *DataReader* operations can be used to get information about the *DataWriters* that will send data to this *DataReader*.

- `get_matched_publications()`
- `get_matched_publication_data()`

The `get_matched_publications()` operation will return a sequence of handles to matched *DataWriters*. You can use these handles in the `get_matched_publication_data()` method to get information about the *DataWriter* such as the values of its QoS Policies.

Note that *DataWriter* that have been ignored using the *DomainParticipant's* `ignore_publication()` operation are not considered to be matched even if the *DataWriter* has the same *Topic* and compatible QoS Policies. Thus, they will not be included in the list of *DataWriters* returned by `get_matched_publications()`. See [Ignoring Publications and Subscriptions \(Section 16.4.2 on page 786\)](#) for more on `ignore_publication()`.

You can also get the `DATA_READER_PROTOCOL_STATUS` for matching publications with `get_matched_publication_datareader_protocol_status()` (see [DATA_READER_PROTOCOL_STATUS \(Section 7.3.7.3 on page 472\)](#)).

Note:

- Status/data for a matched publication is only kept while the matched publication is alive. Once a matched publication is no longer alive, its status is deleted. If you try to get the status/data for a matched publication that is no longer alive, the 'get data' or 'get status' call will return an error.

See also: [Finding the Matching Publication's ParticipantBuiltinTopicData \(Section 7.3.9.2 below\)](#)

7.3.9.2 Finding the Matching Publication's ParticipantBuiltinTopicData

get_matched_publication_participant_data() allows you to get the `DDS_ParticipantBuiltinTopicData` (see [Table 16.1 Participant Built-in Topic's Data Type \(DDS_ParticipantBuiltinTopicData\)](#)) of a matched publication using a publication handle.

This operation retrieves the information on a discovered *DomainParticipant* associated with the publication that is currently matching with the *DataReader*.

The publication handle passed into this operation must correspond to a publication currently associated with the *DataReader*. Otherwise, the operation will fail with `RETCODE_BAD_PARAMETER`. The operation may also fail with `RETCODE_PRECONDITION_NOT_MET` if the publication handle corresponds to the same *DomainParticipant* to which the *DataReader* belongs.

Use **get_matched_publications()** (see [Finding Matching Publications \(Section 7.3.9.1 on the previous page\)](#)) to find the publications that are currently matched with the *DataReader*.

Note: This operation does not retrieve the `ParticipantBuiltinTopicData` property. This information is available through the **on_data_available()** callback (if a *DataReaderListener* is installed on the *PublicationBuiltinTopicDataDataReader*).

7.3.9.3 Finding a DataReader's Related Entities

These *DataReader* operations are useful for obtaining a handle to various related entities:

- **get_subscriber()**
- **get_topicdescription()**

The **get_subscriber()** operation returns the *Subscriber* that created the *DataReader*. **get_topicdescription()** returns the *Topic* with which the *DataReader* is associated.

7.3.9.4 Looking Up an Instance Handle

Some operations, such as **read_instance()** and **take_instance()**, take an **instance_handle** parameter. If you need to get such as handle, you can call the **lookup_instance()** operation, which takes an instance as a parameter and returns a handle to that instance.

7.3.9.5 Getting the Key Value for an Instance

If you have a handle to a data-instance, you can use the **FooDataReader**'s `get_key_value()` operation to retrieve the key for that instance. The value of the key is decomposed into its constituent fields and returned in a **Foo** structure. For information on keys and keyed data types, please see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#).

7.4 Using DataReaders to Access Data (Read & Take)

For user applications to access the data received for a *DataReader*, they must use the type-specific derived class or set of functions in the C API. Thus for a user data type 'Foo', you must use methods of the **FooDataReader** class. The type-specific class or functions are automatically generated if you use *RTI Code Generator*. Else, you will have to create them yourself, see [Type Codes for Built-in Types \(Section 3.8.4.1 on page 143\)](#) for more details.

7.4.1 Using a Type-Specific DataReader (FooDataReader)

This section doesn't apply to the Modern C++ API, where a *DataReader*'s data type is part of its template definition: **DataReader<Foo>**.

Using a *Subscriber* you will create a *DataReader* associating it with a specific data type, for example 'Foo'. Note that the *Subscriber*'s `create_datareader()` method returns a generic *DataReader*. When your code is ready to access *DDS* data samples received for the *DataReader*, you must use type-specific operations associated with the **FooDataReader**, such as `read()` and `take()`.

To cast the generic *DataReader* returned by `create_datareader()` into an object of type **FooDataReader**, you should use the type-safe `narrow()` method of the **FooDataReader** class. `narrow()` will make sure that the generic *DataReader* passed to it is indeed an object of the **FooDataReader** class before it makes the cast. Else, it will return NULL. [Simple SubscriberListener \(Section Figure 7.8 on page 456\)](#) shows an example:

```
Foo_reader = FooDataReader::narrow(reader);
```

[Table 7.3 DataReader Operations](#) lists type-specific operations using a **FooDataReader**. Also listed are generic, non-type specific operations that can be performed using the base class object **DDSDataReader** (or **DDS_DataReader** in C). In C, you must pass a pointer to a **DDS_DataReader** to those generic functions.

7.4.2 Loaning and Returning Data and SampleInfo Sequences

7.4.2.1 C, Traditional C++, Java and .NET

The **read()** and **take()** operations (and their variations) return information to your application in two sequences:

- Received *DDS* data samples in a sequence of the data type
- Corresponding information about each DDS sample in a **SampleInfo** sequence

These sequences are parameters that are passed by your code into the **read()** and **take()** operations. If you use empty sequences (sequences that are initialized but have a maximum length of 0), Connex DDS will fill those sequences with memory directly loaned from the receive queue itself. There is no copying of the data or of **SampleInfo** when the contents of the sequences are loaned. This is certainly the most efficient way for your code to retrieve the data.

However when you do so, your code must return the loaned sequences back to Connex DDS so that they can be reused by the receive queue. If your code does not return the loan by calling the **FooDataReader**'s **return_loan()** method, then Connex DDS will eventually run out of memory to store *DDS* data samples received from the network for that *DataReader*. See [Using Loaned Sequences in read\(\) and take\(\) \(Section Figure 7.16 below\)](#) for an example of borrowing and returning loaned sequences.

```
DDS_ReturnCode_t return_loan(
    FooSeq &received_data, DDS_SampleInfoSeq &info_seq);
```

Figure 7.16 Using Loaned Sequences in read() and take()

```
// In C++ and Java, sequences are automatically initialized
// to be empty
FooSeq data_seq;1
DDS_SampleInfoSeq info_seq;
DDS_ReturnCode_t retcode;
...
// with empty sequences, a take() or read() will return loaned
// sequence elements
retcode = Foo_reader->take(data_seq, info_seq,
    DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
```

¹For the C API, you must use the `FooSeq_initialize()` and `DDS_SampleInfoSeq_initialize()` operations or the macro `DDS_SEQUENCE_INITIALIZER` to initialize the `FooSeq` and `DDS_SampleInfoSeq` to be empty. For example, `DDS_SampleInfoSeq infoSeq; DDS_SampleInfoSeq_initialize(&infoSeq);` or `FooSeq fooSeq = DDS_SEQUENCE_INITIALIZER;`

```

    ... // process the returned data
    // must return the loaned sequences when done processing
    Foo_reader->return_loan(data_seq, info_seq);
    ...

```

If your code provides its own sequences to the read/take operations, then Connex DDS will copy the data from the receive queue. In that case, you do not have to call **return_loan()** when you are finished with the data. However, you must make sure the following is true, or the read/take operation will fail with a return code of **DDS_RETCODE_PRECONDITION_NOT_MET**:

- The **received_data** of type **FooSeq** and **info_seq** of type **DDS_SampleInfoSeq** passed in as parameters have the same maximum size (length).
- The maximum size (length) of the sequences are less than or equal to the passed in parameter, **max_samples**.

7.4.2.2 Modern C++

The **read()** and **take()** operations (and their variations) return **LoanedSamples**, an iterable collection of loaned, read-only samples each containing the actual data and meta-information about the sample. A **LoanedSamples** collection automatically returns the loan to the middleware in its destructor. You can also explicitly call **LoanedSamples::return_loan()**.

Figure 7.17 Using **LoanedSamples** to read data

```

dds::sub::LoanedSamples<Foo> samples = reader.take();
for (auto sample : samples) { // process the data
    if (sample.info().valid()) {
        std::cout << sample.data() << std::endl;
    }
}

```

7.4.3 Accessing DDS Data Samples with Read or Take

To access the *DDS* data samples that Connex DDS has received for a *DataReader*, you must invoke the **read()** or **take()** methods. These methods return a list (sequence) of *DDS* data samples and additional information about the *DDS* samples in a corresponding list (sequence) of **SampleInfo** structures. The contents of **SampleInfo** are described in [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#).

Calling **read()**, **take()**, or one of their variations resets the **DATA_AVAILABLE** status.

The way Connex DDS builds the collection of *DDS* samples depends on QoS policies set on the *DataReader* and *Subscriber*, the **source_timestamp** of the *DDS* samples, and the **sample_states**, **view_states**, and **instance_states** parameters passed to the read/take operation.

In **read()** and **take()**, you may enter parameters so that Connex DDS selectively returns *DDS* data samples currently stored in the *DataReader*'s receive queue. You may want Connex DDS to return all of the data in a single list or only a subset of the available *DDS* samples as configured using the **sample_states**, **view_states**, and **instance_states** masks. [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#) describes how these masks are used to determine which *DDS* data samples should be returned.

7.4.3.1 Read vs. Take

The difference between **read()** and **take()** is how Connex DDS treats the data that is returned. With **take()**, Connex DDS will remove the data from the *DataReader*'s receive queue. The data returned by Connex DDS is no longer stored by Connex DDS. With **read()**, Connex DDS will continue to store the data in the *DataReader*'s receive queue. The same data may be read again until it is taken in subsequent **take()** calls. Note that the data stored in the *DataReader*'s receive queue may be overwritten, even if it has not been read, depending on the setting of the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#).

The **read()** and **take()** operations are non-blocking calls, so that they may return no data (**DDS_RETCODE_NO_DATA**) if the receive queue is empty or has no data that matches the criteria specified by the **StateMasks**.

The **read_w_condition()** and **take_w_condition()** operations take a **ReadCondition** as a parameter instead of *DDS* sample, view or instance states. The only *DDS* samples returned will be those for which the **ReadCondition** is **TRUE**. These operations, in conjunction with **ReadConditions** and a **WaitSet**, allow you to perform 'waiting reads.' For more information, see [ReadConditions and QueryConditions \(Section 4.6.7 on page 195\)](#).

As you will see, **read** and **take** have the same parameters:

```
DDS_ReturnCode_t read( FooSeq &received_data_seq,
                      DDS_SampleInfoSeq &info_seq,
                      DDS_Long max_samples,
                      DDS_SampleStateMask sample_states,
                      DDS_ViewStateMask view_states,
                      DDS_InstanceStateMask instance_states);

DDS_ReturnCode_t take( FooSeq &received_data_seq,
                      DDS_SampleInfoSeq &info_seq,
                      DDS_Long max_samples,
                      DDS_SampleStateMask sample_states,
                      DDS_ViewStateMask view_states,
                      DDS_InstanceStateMask instance_states);
```

Note: These operations may loan internal Connex DDS memory, which must be returned with **return_loan()**. See [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#).

Both operations return an ordered collection of DDS data samples (in the **received_data_seq** parameter) and information about each DDS sample (in the **info_seq** parameter). Exactly how they are ordered depends on the setting of the **PRESENTATION QosPolicy** (Section 6.4.6 on page 330) and the **DESTINATION_ORDER QosPolicy** (Section 6.5.6 on page 365). For more details please see the API Reference HTML documentation for **read()** and **take()**.

In **read()** and **take()**, you can use the **sample_states**, **view_states**, and **instance_states** parameters to specify properties that are used to select the actual DDS samples that are returned by those methods. With different combinations of these three parameters, you can direct Connex DDS to return all DDS data samples, DDS data samples that you have not accessed before, the DDS data samples of instances that you have not seen before, DDS data samples of instances that have been disposed, etc. The possible values for the different states are described both in the API Reference HTML documentation and in [The SampleInfo Structure](#) (Section 7.4.6 on page 504).

[Table 7.17 Read and Take Operations](#) lists the variations of the **read()** and **take()** operations.

Table 7.17 Read and Take Operations

Read Operations	Take Operations	Modern C++ ¹	Description	Reference
read	take	reader.read() or reader.select() .state(...) .read()	Reads/takes a collection of DDS data samples from the <i>DataReader</i> . Can be used for both keyed and non-keyed data types.	Accessing DDS Data Samples with Read or Take (Section 7.4.3 on page 493)
read_instance	take_instance	reader.select() .instance(...) .read()	Identical to read() and take() , but all returned DDS samples belong to a single instance, which you specify as a parameter. Can only be used with keyed data types.	read_instance and take_instance (Section 7.4.3.4 on page 497)
read_instance_w_condition	take_instance_w_condition	reader.select() .instance() .condition(...) .read()	Identical to read_instance() and take_instance() , but all returned DDS samples belong to the single specified instance <i>and</i> satisfy the specified ReadCondition.	read_instance_w_condition and take_instance_w_condition (Section 7.4.3.7 on page 500)
read_next_instance	take_next_instance	reader.select() .next_instance(...).read()	Similar to read_instance() and take_instance() , but the actual instance is not directly specified as a parameter. Instead, the DDS samples will all belong to instance ordered after the instance that is specified by the previous_handle parameter.	read_next_instance and take_next_instance (Section 7.4.3.5 on page 498)

¹For the Modern C++, only the read() operation is shown; the take() variant is parallel.

Table 7.17 Read and Take Operations

Read Operations	Take Operations	Modern C++ ¹	Description	Reference
read_next_instance_w_condition	take_next_instance_w_condition	reader.select() .next_instance(...) .condition(...) .read()	Accesses a collection of DDS data samples of the next instance that match a specific set of ReadConditions, from the <i>DataReader</i> .	read_next_instance_w_condition and take_next_instance_w_condition (Section 7.4.3.8 on page 501)
read_next_sample	take_next_sample	reader.select() .state (DataState::not_read())	Provides a convenient way to access the next DDS DDS sample in the receive queue that has not been accessed before.	read_next_sample and take_next_sample (Section 7.4.3.3 on the facing page)
read_w_condition	take_w_condition	reader.select() .condition(...)	Accesses a <i>collection</i> of DDS data samples from the <i>DataReader</i> that match specific ReadCondition criteria.	read_w_condition and take_w_condition (Section 7.4.3.6 on page 500)

7.4.3.2 General Patterns for Accessing Data

Once the DDS data samples are available to the data readers, the DDS samples can be read or taken by the application. The basic rule is that the application may do this in any order it wishes. This approach is very flexible and allows the application ultimate control.

To access data coherently, or in order, the [PRESENTATION QosPolicy](#) (Section 6.4.6 on page 330) must be set properly.

Accessing DDS samples If No Order or Coherence Is Required

Simply access the data by calling read/take on each *DataReader* in any order you want.

You do not have to call **begin_access()** and **end_access()**. However, doing so is not an error and it will have no effect.

You can call the *Subscriber*'s **get_datareaders()** operation to see which *DataReaders* have data to be read, but you do not need to read all of them or read them in a particular order. The **get_datareaders()** operation will return a logical 'set' in the sense that the same *DataReader* will not appear twice. The order of the *DataReaders* returned is not specified.

Accessing DDS samples within a SubscriberListener

¹For the Modern C++, only the read() operation is shown; the take() variant is parallel.

This case describes how to access the data inside the listener's **on_data_on_readers()** operation (regardless of the PRESENTATION QoS policy settings).

To do so, you can call read/take on each *DataReader* in any order. You can also delegate accessing of the data to the DataReaderListeners by calling the *Subscriber's* **notify_datareaders()** operation.

Similar to the previous case, you can still call the Subscriber's **get_datareaders()** operation to determine which *DataReaders* have data to be read, but you do not have to read all of them, or read them in a particular order. **get_datareaders()** will return a logical 'set.'

You do not have to call **begin_access()** and **end_access()**. However, doing so is not an error and it will have no effect.

7.4.3.3 read_next_sample and take_next_sample

The **read_next_sample()** or **take_next_sample()** operation is used to retrieve the next DDS sample that hasn't already been accessed. It is a simple way to 'read' DDS samples and frees your application from managing sequences and specifying DDS sample, instance or view states. It behaves the same as calling **read()** or **take()** with **max_samples = 1**, **sample_states = NOT_READ**, **view_states = ANY_VIEW_STATE**, and **instance_states = ANY_INSTANCE_STATE**.

```
DDS_ReturnCode_t read_next_sample(
    Foo & received_data, DDS_SampleInfo & sample_info);
DDS_ReturnCode_t take_next_sample(
    Foo & received_data, DDS_SampleInfo & sample_info);
```

It copies the next, not-previously-accessed data value from the *DataReader*. It also copies the DDS sample's corresponding **DDS_SampleInfo** structure.

If there is no unread data in the *DataReader*, the operation will return **DDS_RETCODE_NO_DATA** and nothing is copied.

Since this operation copies both the DDS data sample and the **SampleInfo** into user-provided storage, it does not allocate nor loan memory. You do not have to call **return_loan()** after this operation.

Note: If the **received_data** parameter references a structure that contains a sequence and that sequence has not been initialized, the operation will return **DDS_RETCODE_ERROR**.

7.4.3.4 read_instance and take_instance

The **read_instance()** and **take_instance()** operations are identical to **read()** and **take()**, but they are used to access DDS samples for just a specific instance (key value). The parameters are the same, except you must also supply an instance handle. These functions can only be used when the *DataReader* is tied to a

keyed type, see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#) for more about keyed data types.

These operations may return `BAD_PARAMETER` if the instance handle does not correspond to an existing data-object known to the *DataReader*.

The handle to a particular data instance could have been cached from a previous `read()` operation (value taken from the `SampleInfo` struct) or created by using the *DataReader*'s `lookup_instance()` operation.

```
DDS_ReturnCode_t read_instance(
    FooSeq &received_data,
    DDS_SampleInfoSeq &info_seq,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t &a_handle,
    DDS_SampleStateMask sample_states,
    DDS_ViewStateMask view_states,
    DDS_InstanceStateMask instance_states);
```

Note: This operation may loan internal Connex DDS memory, which must be returned with `return_loan()`. See [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#).

7.4.3.5 read_next_instance and take_next_instance

The `read_next_instance()` and `take_next_instance()` operations are similar to `read_instance()` and `take_instance()` in that they return DDS samples for a specific data instance (key value). The difference is that instead of passing the handle of the data instance for which you want DDS data samples, instead you pass the handle to a 'previous' instance. The returned DDS samples will all belong to the 'next' instance, where the ordering of instances is explained below.

```
DDS_ReturnCode_t read_next_instance(
    FooSeq &received_data,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t &previous_handle
    DDS_SampleStateMask sample_states,
    DDS_ViewStateMask view_states,
    DDS_InstanceStateMask instance_states)
```

Connex DDS orders all instances relative to each other.¹ This ordering depends on the value of the key as defined for the data type associated with the *Topic*. For the purposes of this discussion, it is 'as if' each

^aThe ordering of the instances is specific to each implementation of the DDS standard; to maximize the portability of your code, do not assume any particular order. In the case of Connex DDS (and likely other DDS implementations as well), the order is not likely to be meaningful to you as a developer; it is simply important that some ordering exists.

instance handle is represented by a unique integer and thus different instance handles can be ordered by their value.

This operation will return values for the *next* instance handle that has DDS data samples stored in the receive queue (that meet the criteria specified by the **StateMasks**). The *next* instance handle will be ordered after the **previous_handle** that is passed in as a parameter.

The special value **DDS_HANDLE_NIL** can be passed in as the **previous_handle**. Doing so, you will receive values for the “smallest” instance handle that has DDS data samples stored in the receive queue that you have not yet accessed.

You can call the **read_next_instance()** operation with a **previous_handle** that does not correspond to an instance currently managed by the *DataReader*. For example, you could use this approach to iterate through all the instances, take all the DDS samples with a **NOT_ALIVE_NO_WRITERS** instance_state, return the loans (at which point the instance information may be removed, and thus the handle becomes invalid), and then try to read the next instance.

The example below shows how to use **take_next_instance()** iteratively to process all the data received for an instance, one instance at a time. We always pass in **DDS_HANDLE_NIL** as the value of **previous_handle**. Each time through the loop, we will receive DDS samples for a different instance, since the previous time through the loop, all of the DDS samples of the previous instance were returned (and thus accessed).

```
FooSeq received_data;1
DDS_SampleInfoSeq info_seq;
while (retcode = reader->take_next_instance(received_data, info_seq,
                                           DDS_LENGTH_UNLIMITED, DDS_HANDLE_NIL,
                                           DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE,
                                           DDS_ANY_INSTANCE_STATE)
       != DDS_RETCODE_NO_DATA) {
    // the data samples returned in received_data will all
    // be for a single instance
    // process the data
    // now return the loaned sequences
    if (reader->return_loan(received_data, info_seq)
        != DDS_RETCODE_OK) {
        // handle error
    }
}
```

¹In the C API, you must use the `FooSeq_initialize()` and `DDS_SampleInfoSeq_initialize()` operations or the macro `DDS_SEQUENCE_INITIALIZER` to initialize the `FooSeq` and `DDS_SampleInfoSeq` to be empty. For example, `DDS_SampleInfoSeq infoSeq; DDS_SampleInfoSeq_initialize(&infoSeq);` or `FooSeq fooSeq = DDS_SEQUENCE_INITIALIZER;`

Note: This operation may loan internal Connex DDS memory, which must be returned with **return_loan()**. See [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#).

7.4.3.6 read_w_condition and take_w_condition

The **read_w_condition()** and **take_w_condition()** operations are identical to **read()** and **take()**, but instead of passing in the `sample_states`, `view_states`, and `instance_states` mask parameters directly, you pass in a `ReadCondition` (which specifies these masks).

```
DDS_ReturnCode_t read_w_condition (
    FooSeq &received_data,
    DDS_SampleInfoSeq &info_seq,
    DDS_Long max_samples,
    DDSReadCondition *condition)
```

Note: This operation may loan internal Connex DDS memory, which must be returned with **return_loan()**. See [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#).

7.4.3.7 read_instance_w_condition and take_instance_w_condition

The **read_instance_w_condition()** and **take_instance_w_condition()** operations are similar to **read_instance()** and **take_instance()**, respectively, except that the returned DDS samples must also satisfy a specified `ReadCondition`.

```
DDS_ReturnCode_t read_instance_w_condition(
    FooSeq & received_data,
    DDS_SampleInfoSeq & info_seq,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t & a_handle,
    DDSReadCondition * condition);
```

The behavior of **read_instance_w_condition()** and **take_instance_w_condition()** follows the same rules as **read()** and **take()** regarding pre-conditions and post-conditions for the `received_data` and `sample_info` parameters.

These functions can only be used when the *DataReader* is tied to a keyed type, see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#) for more about keyed data types.

Similar to **read()**, these operations must be provided on the specialized class that is generated for the particular application data-type that is being accessed.

Note: These operations may loan internal Connex DDS memory, which must be returned with **return_loan()**. See [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#).

7.4.3.8 read_next_instance_w_condition and take_next_instance_w_condition

The `read_next_instance_w_condition()` and `take_next_instance_w_condition()` operations are identical to `read_next_instance()` and `take_next_instance()`, but instead of passing in the `sample_states`, `view_states`, and `instance_states` mask parameters directly, you pass in a `ReadCondition` (which specifies these masks).

```
DDS_ReturnCode_t read_next_instance_w_condition (
    FooSeq &received_data,
    DDS_SampleInfoSeq &info_seq,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t &previous_handle,
    DDSReadCondition *condition)
```

Note: This operation may loan internal Connex DDS memory, which must be returned with `return_loan()`. See [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#).

7.4.3.9 The select() API (Modern C++)

The Modern C++ API combines all the previous ways to read data into a single operation: `reader.select()`. This call is followed by one or more calls to functions that configure the query and always ends in a call to `read()` or `take()`. These are the functions that configure a `select()`:

Function	Description	Default
<code>max_samples()</code>	Specifies the maximum number of samples to read or take in this call	Up to the value specified in <code>max_samples_per_read</code> (Section on page 518)
<code>instance()</code>	Specifies an instance to read or take	All instances
<code>next_instance()</code>	Indicates that read or take should return samples for the instance that follows the one being passed (Note: both <code>next_instance()</code> and <code>instance()</code> can't be specified at the same time)	All instances
<code>state()</code>	Specifies the sample state, view state and instance state	All samples
<code>content()</code>	Specifies a query on the data values to read	All samples
<code>condition()</code>	Specifies a condition (see <code>read_w_condition()</code>). If <code>condition()</code> is specified <code>state()</code> and <code>content()</code> cannot be specified. When running a query more than once on the same <code>DataReader</code> , it is more efficient to create a <code>QueryCondition</code> and pass it to <code>condition()</code> rather than using <code>content()</code> .	All samples

To read or take using the default options, simply call `reader.read()` or `reader.take()` with no arguments.

The following example shows how to call `select()`:

```
dds::sub::LoanedSamples<Foo> samples =
    reader.select()
        .max_samples(20)
        .state(dds::sub::status::DataState::new_instance())
        .content(dds::sub::Query(reader, "x > 10"))
        .instance(my_instance_handle)
        .take();
```

7.4.4 Acknowledging DDS Samples

DDS samples can be acknowledged one at a time, or as a group.

To explicitly acknowledge a single DDS sample:

```
DDS_ReturnCode_t acknowledge_sample (
    const DDS_SampleInfo & sample_info);
DDS_ReturnCode_t acknowledge_sample (
    const DDS_SampleInfo & sample_info,
    const DDS_AckResponseData_t & response_data);
```

Or you may acknowledge all previously accessed DDS samples by calling:

```
DDS_ReturnCode_t DDSDataReader::acknowledge_all ()
DDS_ReturnCode_t DDSDataReader::acknowledge_all (
    const DDS_AckResponseData_t & response_data)
```

Where:

- sample_info** is of type `DDS_SampleInfo`, identifying the DDS sample being acknowledged
- response_data** is response data sent to the `DataWriter` upon acknowledgment

These operations can only be used when the *DataReader's* [RELIABILITY QoS Policy \(Section 6.5.19 on page 400\)](#) has an **acknowledgment_kind** set to `DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE`. You must also set **max_app_ack_response_length** (in the [DATA_READER_RESOURCE_LIMITS QoS Policy \(DDS Extension\) \(Section 7.6.2 on page 517\)](#)) to a value greater than zero.

See also: [Application Acknowledgment \(Section 6.3.12 on page 288\)](#) and [Guaranteed Delivery of Data \(Section Chapter 13 on page 695\)](#).

7.4.5 The Sequence Data Structure

(This section doesn't apply to the Modern C++ API)

The DDS specification uses sequences whenever a variable-length array of elements must be passed through the API. This includes passing QoS Policies into Connex DDS, as well as retrieving DDS data

samples from Connex DDS. A sequence is an ordered collection of elements of the same type. The type of a sequence containing elements of type “**Foo**” (whether “**Foo**” is one of your types or a built-in Connex DDS type) is typically called “**FooSeq**.”

In all APIs except Java, **FooSeq** contains deep copies of **Foo** elements; in Java, which does not provide direct support for deep copy semantics, **FooSeq** contains references to **Foo** objects. In Java, sequences implement the `java.util.List` interface, and thus support all of the collection APIs and idioms familiar to Java programmers.

A sequence is logically composed of three things: an array of elements, a *maximum* number of elements that the array may contain (i.e. its allocated size), and a logical *length* indicating how many of the allocated elements are valid. The length may vary dynamically between 0 and the maximum (inclusive); it is not permissible to access an element at an index greater than or equal to the length.

A sequence may either “own” the memory associated with it, or it may “borrow” that memory. If a sequence owns its own memory, then the sequence itself will allocate the its memory and is permitted to grow and shrink that memory (i.e. change its maximum) dynamically.

You can also loan a sequence of memory using the sequence-specific operations `loan_contiguous()` or `loan_discontiguous()`. This is useful if you want Connex DDS to copy the received DDS data samples directly into data structures allocated in user space.

Please do not confuse (a) the user loaning memory to a sequence with (b) Connex DDS loaning internal memory from the receive queue to the user code via the `read()` or `take()` operations. For sequences of user data, these are complementary operations. `read()` and `take()` loan memory to the user, passing in a sequence that has been loaned memory with `loan_contiguous()` or `loan_discontiguous()`.

A sequence with loaned of memory may not change its maximum size.

For C developers:

In C, because there is no concept of a constructor, sequences must be initialized before they are used. You can either set a sequence equal to the macro `DDS_SEQUENCE_INITIALIZER` or use a sequence-specific method, `<type>Seq_initialize()`, to initialize sequences.

For C++, C++/CLI, and C# developers:

C++ sequence classes overload the `[]` operators to allow you to access their elements as if the sequence were a simple array. However, for code portability reasons, Connex DDS’s implementation of sequences does not use the Standard Template Library (STL).

For Java developers:

In Java, sequences implement the `List` interface, and typically, a `List` must contain `Objects`; it cannot contain primitive types directly. This restriction makes `Lists` of primitive types less efficient because each type must be wrapped and unwrapped into and from an `Object` as it is added to and removed from the `List`.

Connex DDS provides a more efficient implementation for sequences of primitive types. In Connex DDS, primitive sequence types (e.g., **IntSeq**, **FloatSeq**, etc.) are implemented as wrappers around arrays of primitive types. The wrapper also provides the usual **List** APIs; however, these APIs manipulate **Objects** that store the primitive type.

More efficient APIs are also provided that manipulate the primitive types directly and thus avoid unnecessary memory allocations and type casts. These additional methods are named according to the pattern *<standard method><primitive type>*; for example, the **IntSeq** class defines methods **addInt()** and **getInt()** that correspond to the **List** APIs **add()** and **get()**. **addInt()** and **getInt()** directly manipulate **int** values while **add()** and **get()** manipulate **Objects** that contain a single **int**.

For more information on sequence APIs in all languages, please consult the API Reference HTML documentation (from the main page, select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **Sequence Support**).

7.4.6 The SampleInfo Structure

When you invoke the **read/take** operations, for every DDS data sample that is returned, a corresponding **SampleInfo** is also returned. **SampleInfo** structures provide you with additional information about the DDS data samples received by Connex DDS.

Table 7.18 [DDS_SampleInfo Structure](#) shows the format of the **SampleInfo** structure.

Table 7.18 DDS_SampleInfo Structure

Type	Field Name	Description
DDS_SampleStateKind	sample_state	See Sample States (Section 7.4.6.2 on page 506)
DDS_ViewStateKind	view_state	See View States (Section 7.4.6.3 on page 506)
DDS_InstanceStateKind	instance_state	See Instance States (Section 7.4.6.4 on page 507)
DDS_Time_t	source_timestamp	Time stored by the <i>DataWriter</i> when the DDS sample was written.
DDS_InstanceHandle_t	instance_handle	Handle to the data-instance corresponding to the DDS sample.
DDS_InstanceHandle_t	publication_handle	Local handle to the <i>DataWriter</i> that modified the instance. This is the same instance handle returned by get_matched_publications() . You can use this handle when calling get_matched_publication_data() .

Table 7.18 DDS_SampleInfo Structure

Type	Field Name	Description
DDS_Long	disposed_generation_count	See Generation Counts and Ranks (Section 7.4.6.5 on page 508) .
	no_writers_generation_count	
	sample_rank	
	generation_rank	
	absolute_generation_rank	
DDS_Boolean	valid_data	Indicates whether the DDS data sample includes valid data. See Valid Data Flag (Section 7.4.6.6 on page 510) .
DDS_Time_t	reception_timestamp	Time stored when the DDS sample was committed by the <i>DataReader</i> . See Reception Timestamp (Section 7.4.6.1 on the next page) .
DDS_SequenceNumber_t	publication_sequence_number	Publication sequence number assigned when the DDS sample was written by the <i>DataWriter</i> .
DDS_SequenceNumber_t	reception_sequence_number	Reception sequence number assigned when the DDS sample was committed by the <i>DataReader</i> . See Reception Timestamp (Section 7.4.6.1 on the next page) .
struct DDS_GUID_t	original_publication_virtual_guid	Original publication virtual GUID. If the <i>Publisher's</i> access_scope is GROUP, this field contains the <i>Publisher</i> virtual GUID that uniquely identifies the <i>DataWriter</i> group.
struct DDS_SequenceNumber_t	original_publication_virtual_sequence_number	Original publication virtual sequence number. If the <i>Publisher's</i> access_scope is GROUP, this field contains the <i>Publisher</i> virtual sequence number that uniquely identifies a DDS sample within the <i>DataWriter</i> group.
DDS_Long	flag	Flags associated with the DDS sample; set by using the flag field in <code>DDS_WriteParams_t</code> when writing a DDS sample with <code>FooDataWriter_write_w_params()</code> (see Writing Data (Section 6.3.8 on page 283)).

Table 7.18 DDS_SampleInfo Structure

Type	Field Name	Description
struct DDS_GUID_t	source_guid	The application logical data source associated with the sample.
struct DDS_GUID_t	related_source_guid	The application logical data source that is related to the sample.
struct DDS_GUID_t	related_subscription_guid	The related_reader_guid associated with the sample.

7.4.6.1 Reception Timestamp

In reliable communication, if DDS data samples are received out of order, Connex DDS will not deliver them until all the previous DDS data samples have been received. For example, if DDS sample 2 arrives before DDS sample 1, DDS sample 2 cannot be delivered until DDS sample 1 is received. The **reception_timestamp** is the time when all previous DDS samples has been received—the time at which the DDS sample is *committed*. If DDS samples are all received in order, the committed time will be same as reception time. However, if DDS samples are lost on the wire, then the committed time will be later than the initial reception time.

7.4.6.2 Sample States

For each DDS sample received, Connex DDS keeps a **sample_state** relative to each *DataReader*. The **sample_state** can be either:

- READ: The *DataReader* has already accessed that DDS sample by means of **read()**.
- NOT_READ: The *DataReader* has never accessed that DDS sample before.

The DDS samples retrieved by a **read()** or **take()** need not all have the same **sample_state**.

7.4.6.3 View States

For each instance (identified by a unique key value), Connex DDS keeps a **view_state** relative to each *DataReader*. The **view_state** can be either:

- NEW: Either this is the first time the *DataReader* has ever accessed DDS samples of the instance, or the *DataReader* has accessed previous DDS samples of the instance, but the instance has since been reborn (i.e. become not-alive and then alive again). These two cases are distinguished by examining

the **disposed_generation_count** and the **no_writers_generation_count** (see [Generation Counts and Ranks \(Section 7.4.6.5 on the next page\)](#)).

- **NOT_NEW**: The *DataReader* has already accessed DDS samples of the same instance and the instance has not been reborn since.

The **view_state** in the **SampleInfo** structure is really a per-instance concept (as opposed to the **sample_state** which is per DDS sample). Thus all DDS data samples related to the same instance that are returned by **read()** or **take()** will have the same value for **view_state**.

7.4.6.4 Instance States

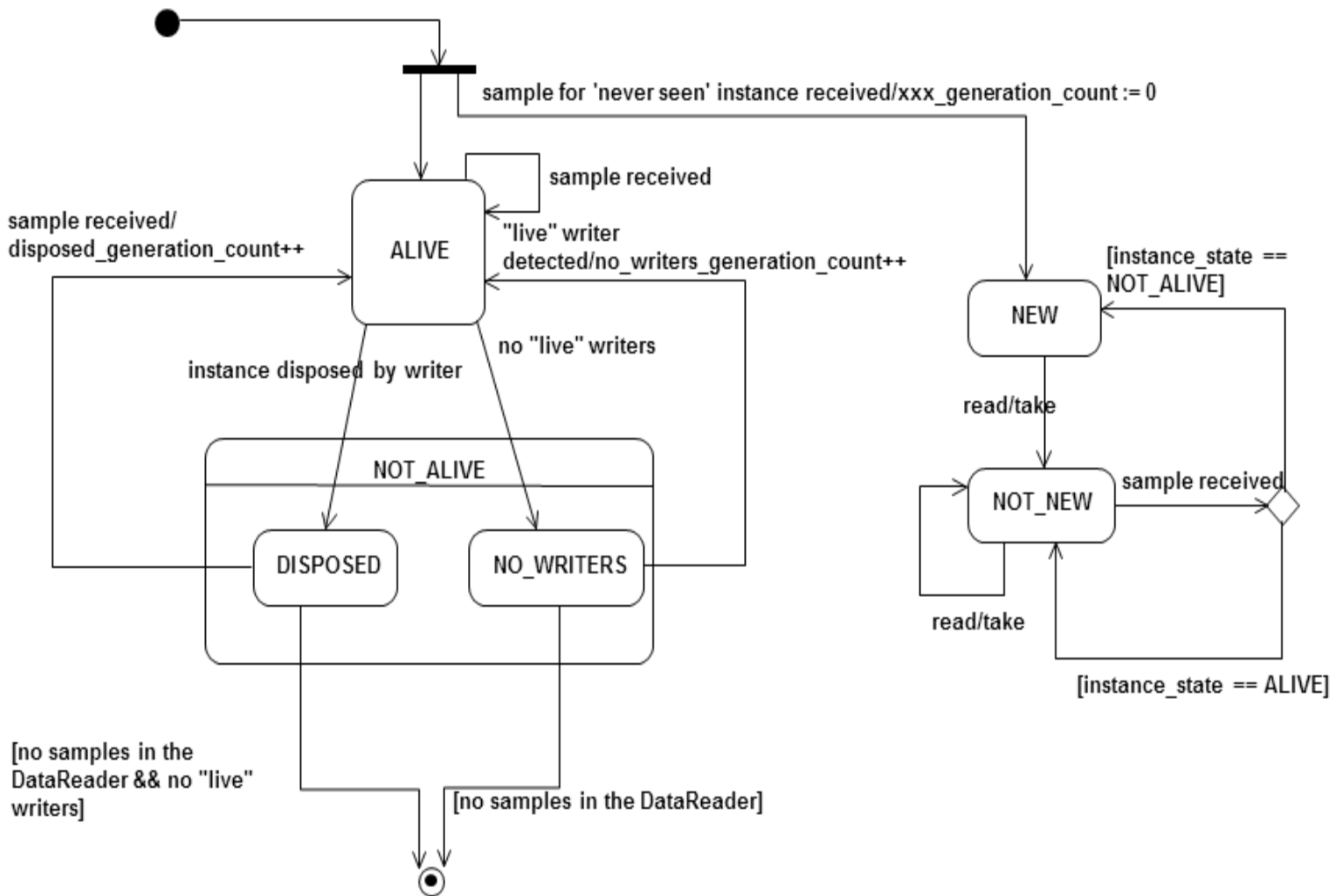
As seen in [Instance States \(Section Figure 7.18 on the next page\)](#), Connex DDS keeps an **instance_state** for each instance; it can be:

- **ALIVE**: The following are all true: (a) DDS samples have been received for the instance, (b) there are live *DataWriters* writing the instance, and (c) the instance has not been explicitly disposed (or more DDS samples have been received after it was disposed).
- **NOT_ALIVE_DISPOSED**: The instance was explicitly disposed by a *DataWriter* by means of the **dispose()** operation.
- **NOT_ALIVE_NO_WRITERS**: The instance has been declared as not-alive by the *DataReader* because it has determined that there are no live *DataWriter* entities writing that instance.

The events that cause the **instance_state** to change can depend on the setting of the **OWNERSHIP QoSPolicy (Section 6.5.15 on page 389)**:

- If **OWNERSHIP QoS** is set to **EXCLUSIVE**, the **instance_state** becomes **NOT_ALIVE_DISPOSED** only if the *DataWriter* that currently “owns” the instance explicitly disposes it. The **instance_state** will become **ALIVE** again only if the *DataWriter* that owns the instance writes it. Note that ownership of the instance is determined by a combination of the **OWNERSHIP** and **OWNERSHIP_STRENGTH** QoS Policies. Ownership of an instance can dynamically change.
- If **OWNERSHIP QoS** is set to **SHARED**, the **instance_state** becomes **NOT_ALIVE_DISPOSED** if any *DataWriter* explicitly disposes the instance. The **instance_state** becomes **ALIVE** as soon as any *DataWriter* writes the instance again.

Figure 7.18 Instance States



Since the **instance_state** in the **SampleInfo** structure is a per-instance concept, all DDS data samples related to the same instance that are returned by **read()** or **take()** will have the same value for **instance_state**.

7.4.6.5 Generation Counts and Ranks

Generation counts and ranks allow your application to distinguish DDS samples belonging to different ‘generations’ of the instance. It is possible for an instance to become alive, be disposed and become not-alive, and then to cycle again from alive to not-alive states during the operation of an application. Each time an instance becomes alive defines a new generation for the instance.

It is possible that an instance may cycle through alive and not-alive states multiple times before the application accesses the DDS data samples for the instance. This means that the DDS data samples returned by

read() and **take()** may cross generations. That is, some DDS samples were published when the instance was alive in one generation and other DDS samples were published when the instance transitioned through the non-alive state into the alive state again. It may be important to your application to distinguish the DDS data samples by the generation in which they were published.

Each *DataReader* keeps two counters for each *new* instance it detects (recall that instances are distinguished by their key values):

- **disposed_generation_count**: Counts how many times the **instance_state** of the corresponding instance changes from **NOT_ALIVE_DISPOSED** to **ALIVE**. The counter is reset when the instance resource is reclaimed.
- **no_writers_generation_count**: Counts how many times the **instance_state** of the corresponding instance changes from **NOT_ALIVE_NO_WRITERS** to **ALIVE**. The counter is reset when the instance resource is reclaimed.

The **disposed_generation_count** and **no_writers_generation_count** fields in the **SampleInfo** structure capture a snapshot of the corresponding counters at the time the corresponding DDS sample was received.

The **sample_rank** and **generation_rank** in the **SampleInfo** structure are computed relative to the sequence of DDS samples returned by **read()** or **take()**:

- **sample_rank**: Indicates how many DDS samples of the same instance follow the current one in the sequence. The DDS samples are always time-ordered, thus the newest DDS sample of an instance will have a **sample_rank** of 0. Depending on what you have configured **read()** and **take()** to return, a **sample_rank** of 0 may or may not be the newest DDS sample that was ever received. It is just the newest DDS sample in the sequence that was returned.
- **generation_rank**: Indicates the difference in ‘generations’ between the DDS sample and the newest DDS sample of the same instance as returned in the sequence. If a DDS sample belongs to the same generation as the newest DDS sample in the sequence returned by **read()** and **take()**, then **generation_rank** will be 0.
- **absolute_generation_rank**: Indicates the difference in ‘generations’ between the DDS sample and the newest DDS sample of the same instance ever received by the *DataReader*. Recall that the data sequence returned by **read()** and **take()** may not contain all of the data in the *DataReader*’s receive queue. Thus, a DDS sample that belongs to the newest generation of the instance will have an **absolute_generation_rank** of 0.

Like the ‘generation count’ values, the ‘rank’ values are also reset to 0 if the instance resource is reclaimed.

By using the **sample_rank**, **generation_rank** and **absolute_generation_rank** information in the **SampleInfo** structure, your application can determine exactly what happened to the instance and thus make appropriate decisions of what to do with the DDS data samples received for the instance. For example:

- A DDS sample with **sample_rank** = 0 is the newest DDS sample of the instance in the returned sequence.
- DDS samples that belong to the same generation will have the same **generation_rank** (as well as **absolute_generation_rank**).
- DDS samples with **absolute_generation_rank** = 0 belong to the newest generation for the instance received by the *DataReader*.

7.4.6.6 Valid Data Flag

The **SampleInfo** structure's **valid_data** flag indicates whether the DDS sample contains data or is only used to communicate a change in the **instance_state** of the instance.

Normally, each DDS sample contains both a **SampleInfo** structure and some data. However, there are situations in which the DDS sample only contains the **SampleInfo** and does not have any associated data. This occurs when Connex DDS notifies the application of a change of state for an instance that was caused by some internal mechanism (such as a timeout) for which there is no associated data. An example is when Connex DDS detects that an instance has no writers and changes the corresponding **instance_state** to **NOT_ALIVE_NO_WRITERS**.

If this flag is **TRUE**, then the DDS sample contains valid Data. If the flag is **FALSE**, the Dds Sample contains no data.

To ensure correctness and portability, your application must check the **valid_data** flag prior to accessing the data associated with the DDS sample, and only access the data if it is **TRUE**.

7.5 Subscriber QoS Policies

Subscribers have the same set of QoS Policies as *Publishers*; see [Publisher/Subscriber QoS Policies \(Section 6.4 on page 312\)](#).

- **ENTITYFACTORY** QoS Policy ([Section 6.4.2 on page 315](#))
- **EXCLUSIVE_AREA** QoS Policy (DDS Extension) ([Section 6.4.3 on page 318](#))
- **GROUP_DATA** QoS Policy ([Section 6.4.4 on page 320](#))
- **PARTITION** QoS Policy ([Section 6.4.5 on page 323](#))
- **PRESENTATION** QoS Policy ([Section 6.4.6 on page 330](#))

7.6 DataReader QoS Policies

This section describes the QoS Policies that are strictly for *DataReaders* (not for *DataWriters*). For a complete list of QoS Policies that apply to *DataReaders*, see [Table 7.16 DataReader QoS Policies](#).

- [DATA_READER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 7.6.1 below\)](#)
- [DATA_READER_RESOURCE_LIMITS QoS Policy \(DDS Extension\) \(Section 7.6.2 on page 517\)](#)
- [READER_DATA_LIFECYCLE QoS Policy \(Section 7.6.3 on page 523\)](#)
- [TIME_BASED_FILTER QoS Policy \(Section 7.6.4 on page 526\)](#)
- [TRANSPORT_MULTICAST QoS Policy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#)
- [TYPE_CONSISTENCY_ENFORCEMENT QoS Policy \(Section 7.6.6 on page 532\)](#)

7.6.1 DATA_READER_PROTOCOL QoS Policy (DDS Extension)

The `DATA_READER_PROTOCOL` QoS Policy applies only to *DataReaders* that are set up for reliable operation (see [RELIABILITY QoS Policy \(Section 6.5.19 on page 400\)](#)). This policy allows the application to fine-tune the reliability protocol separately for each *DataReader*. For details of the reliable protocol used by Connex DDS, see [Reliable Communications \(Section Chapter 10 on page 629\)](#).

Connex DDS uses a standard protocol for packet (user and meta data) exchange between applications. The `DataReaderProtocol` QoS Policy gives you control over configurable portions of the protocol, including the configuration of the reliable data delivery mechanism of the protocol on a per *DataReader* basis.

These configuration parameters control timing and timeouts, and give you the ability to trade off between speed of data loss detection and repair, versus network and CPU bandwidth used to maintain reliability.

It is important to tune the reliability protocol on a per *DataReader* basis to meet the requirements of the end-user application so that data can be sent between *DataWriters* and *DataReaders* in an efficient and optimal manner in the presence of data loss.

You can also use this QoS Policy to control how DDS responds to "slow" reliable *DataReaders* or ones that disconnect or are otherwise lost.

See the [RELIABILITY QoS Policy \(Section 6.5.19 on page 400\)](#) for more information on the per-*DataReader/DataWriter* reliability configuration. The [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QoS Policy \(Section 6.5.20 on page 405\)](#) also play an important role in the DDS reliability protocol.

This policy includes the members presented in [Table 7.19 DDS_DataReaderProtocolQoS Policy](#) and [Table 7.20 DDS_RtpsReliableReaderProtocol_t](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

When setting the fields in this policy, the following rule applies. If this is false, Connex DDS returns `DDS_RETCODE_INCONSISTENT_POLICY` when setting the QoS:

`max_heartbeat_response_delay >= min_heartbeat_response_delay`

Table 7.19 DDS_DataReaderProtocolQoSPolicy

Type	Field Name	Description
DDS_GUID_t	virtual_guid	<p>The virtual GUID (Global Unique Identifier) is used to uniquely identify the same <i>DataReader</i> across multiple incarnations. In other words, this value allows Connex DDS to remember information about a <i>DataReader</i> that may be deleted and then recreated.</p> <p>This value is used to provide durable reader state.</p> <p>For more information, see Durability and Persistence Based on Virtual GUIDs (Section 12.2 on page 680).</p> <p>By default, Connex DDS will assign a virtual GUID automatically. If you want to restore the <i>DataReader's</i> state after a restart, you can get the <i>DataReader's</i> virtual GUID using its <code>get_qos()</code> operation, then set the virtual GUID of the restarted <i>DataReader</i> to the same value.</p>
DDS_UnsignedLong	rtps_object_id	<p>Determines the <i>DataReader's</i> RTPS object ID, according to the DDS-RTPS Interoperability Wire Protocol.</p> <p>Only the last 3 bytes are used; the most significant byte is ignored.</p> <p>The <code>rtps_host_id</code>, <code>rtps_app_id</code>, <code>rtps_instance_id</code> in the WIRE_PROTOCOL QoSPolicy (DDS Extension) (Section 8.5.9 on page 610), together with the 3 least significant bytes in <code>rtps_object_id</code>, and another byte assigned by Connex DDS to identify the entity type, forms the BuiltinTopicKey in SubscriptionBuiltinTopicData.</p>
DDS_Boolean	expects_inline_qos	<p>Specifies whether this <i>DataReader</i> expects inline QoS with every DDS sample.</p> <p><i>DataReaders</i> usually rely on the discovery process to propagate QoS changes for matched <i>DataWriters</i>. Another way to get QoS information is to have it sent inline with a DDS sample.</p> <p>With Connex DDS, <i>DataWriters</i> and <i>DataReaders</i> cache discovery information, so sending inline QoS is typically unnecessary. The use of inline QoS is only needed for stateless implementations of DDS in which <i>DataReaders</i> do not cache Discovery information.</p> <p>The complete set of QoS that a <i>DataWriter</i> may send inline is specified by the Real-Time Publish-Subscribe (RTPS) Wire Interoperability Protocol.</p> <p>Note: The use of inline QoS creates an additional wire-payload, consuming extra bandwidth and serialization/deserialization time.</p>
DDS_Boolean	disable_positive_acks	<p>Determines whether the <i>DataReader</i> sends positive acknowledgements (ACKs) to matching <i>DataWriters</i>.</p> <p>When TRUE, the matching <i>DataWriter</i> will keep DDS samples in its queue for this <i>DataReader</i> for a minimum keep duration (see Disabling Positive Acknowledgements (Section 6.5.3.3 on page 354)).</p> <p>When strict-reliability is not required and NACK-based reliability is sufficient, setting this field reduces overhead network traffic.</p>

Table 7.19 DDS_DataReaderProtocolQosPolicy

Type	Field Name	Description
DDS_Boolean	propagate_dispose_of_unregistered_instances	<p>Indicates whether or not an instance can move to the DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE state without being in the DDS_ALIVE_INSTANCE_STATE state.</p> <p>When set to TRUE, the <i>DataReader</i> will receive dispose notifications even if the instance is not alive.</p> <p>This field only applies to keyed <i>DataReaders</i>.</p> <p>To make sure the key is available to the <i>FooDataReader</i>'s <code>get_key_value()</code> operation, use this option in combination with setting the <i>DataWriter</i>'s <code>serialize_key_with_dispose</code> field (in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.3 on page 347)) to TRUE.</p> <p>See Propagating Serialized Keys with Disposed-Instance Notifications (Section 6.5.3.5 on page 356).</p>
DDS_Boolean	propagate_unregister_of_disposed_instances	<p>Indicates whether or not an instance can move to the DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE state without being in the DDS_ALIVE_INSTANCE_STATE state.</p> <p>When set to TRUE, the <i>DataReader</i> will receive unregister notifications even if the instance is not alive.</p> <p>This field only applies to keyed <i>DataReaders</i>.</p>
DDS_Rtps-ReliableReader-Protocol_t	rtps_reliable_reader	See Table 7.20 DDS_RtpsReliableReaderProtocol_t

Table 7.20 DDS_RtpsReliableReaderProtocol_t

Type	Field Name	Description
DDS_ Duration_ t	heartbeat_ suppression_ duration	<p>How long additionally received heartbeats are suppressed.</p> <p>When a reliable <i>DataReader</i> receives consecutive heartbeats within a short duration, this may trigger redundant NACKs. To prevent the <i>DataReader</i> from sending redundant NACKs, the <i>DataReader</i> may ignore the latter heartbeat(s) for this amount of time.</p> <p>See How Often Heartbeats are Resent (heartbeat_period) (Section 10.3.4.1 on page 645).</p>
	min_ heartbeat_ response_ delay	Minimum delay between when the <i>DataReader</i> receives a heartbeat and when it sends an ACK/NACK.
	max_ heartbeat_ response_ delay	Maximum delay between when the <i>DataReader</i> receives a heartbeat and when it sends an ACK/NACK. Increasing this value helps prevent NACK storms, but increases latency.
	nack_period	Rate at which to send negative acknowledgements to new <i>DataWriters</i> . See Example (Section 7.6.1.3 on page 516) .
DDS_ Long	receive_ window_ size	The number of received out-of-order DDS samples a reader can keep at a time. See Receive Window Size (Section 7.6.1.1 on the facing page)
DDS_ Duration_ t	round_trip_ time	The duration from sending a NACK to receiving a repair of a DDS sample. See Round-Trip Time For Filtering Redundant NACKs (Section 7.6.1.2 on page 516)
DDS_ Duration_ t	app_ack_ period	<p>The period at which application-level acknowledgment messages are sent.</p> <p>A <i>DataReader</i> sends application-level acknowledgment messages to a <i>DataWriter</i> at this periodic rate, and will continue sending until it receives a message from the <i>DataWriter</i> that it has received and processed the acknowledgment.</p>

Table 7.20 DDS_RtpsReliableReaderProtocol_t

Type	Field Name	Description
DDS_Boolean	samples_per_app_ack	<p>The minimum number of DDS samples acknowledged by one application-level acknowledgment message.</p> <p>This setting applies only when the RELIABILITY QosPolicy (Section 6.5.19 on page 400) acknowledgment_kind is set to APPLICATION_EXPLICIT or APPLICATION_AUTO.</p> <p>A <i>DataReader</i> will immediately send an application-level acknowledgment message when it has at least this many DDS samples that have been acknowledged. It will not send an acknowledgment message until it has at least this many DDS samples pending acknowledgment.</p> <p>For example, calling the <i>DataReader</i>'s acknowledge_sample() this many times consecutively will trigger the sending of an acknowledgment message. Calling the <i>DataReader</i>'s acknowledge_all() may trigger the sending of an acknowledgment message, if at least this many DDS samples are being acknowledged at once. See Acknowledging DDS Samples (Section 7.4.4 on page 502).</p> <p>This is independent of the DDS_RtpsReliableReaderProtocol_t's app_ack_period, where a <i>DataReader</i> will send acknowledgment messages at the periodic rate regardless.</p> <p>When this is set to DDS_LENGTH_UNLIMITED, acknowledgment messages are sent only periodically, at the rate set by DDS_RtpsReliableReaderProtocol_t's app_ack_period.</p>
DDS_Duration_t	min_app_ack_response_keep_duration	<p>Minimum duration for which application-level acknowledgment response data is kept.</p> <p>The user-specified response data of an explicit application-level acknowledgment (called by <i>DataReader</i>'s acknowledge_sample() or acknowledge_all() operations) is cached by the <i>DataReader</i> for the purpose of reliably resending the data with the acknowledgment message. After this duration has passed from the time of the first acknowledgment, the response data is dropped from the cache and will not be resent with future acknowledgments for the corresponding DDS sample(s).</p>

7.6.1.1 Receive Window Size

A reliable *DataReader* presents DDS samples it receives to the user in-order. If it receives DDS samples out-of-order, it stores them internally until the other missing DDS samples are received. For example, if the *DataWriter* sends DDS samples 1 and 2, if the *DataReader* receives 2 first, it will wait until it receives 1 before passing the DDS samples to the user.

The number of out-of-order DDS samples that a *DataReader* can keep is set by the **receive_window_size**. A larger window allows more out-of-order DDS samples to be kept. When the window is full, any subsequent out-of-order DDS samples received will be dropped, and such drops would necessitate NACK repairs that would degrade throughput. So, in network environments where out-of-order samples are more probable or where NACK repairs are costly, this window likely should be increased.

By default, the window is set to 256, which is the maximum number of DDS samples a single NACK submessage can request.

7.6.1.2 Round-Trip Time For Filtering Redundant NACKs

When a *DataReader* requests for a DDS sample to be resent, there is a delay from when the NACK is sent, to when it receives the resent DDS sample. During that delay, the *DataReader* may receive HEARTBEATs that normally would trigger another NACK for the same DDS sample. Such redundant repairs waste bandwidth and degrade throughput.

The **round_trip_time** is a user-configured estimate of the delay between sending a NACK to receiving a repair. A *DataReader* keeps track of when a DDS sample has been NACK'd, and will prevent subsequent NACKs from redundantly requesting for the same DDS sample, until the round trip time has passed.

Note that the default value of 0 seconds means that the *DataReader* does not filter for redundant NACKs.

7.6.1.3 Example

For many applications, changing these values will not be necessary. However, the more nodes that your distributed application uses, and the greater the amount of network traffic it generates, the more likely it is that you will want to consider experimenting with these values.

When a reliable *DataReader* receives a heartbeat from a *DataWriter*, it will send an ACK/NACK packet back to the *DataWriter*. Instead of sending the packet out immediately, the *DataReader* can choose to send it after a delay. This policy sets the minimum and maximum time to delay; the actual delay will be a random value in between. (For more on heartbeats and ACK/NACK messages, see [Discovery \(Section Chapter 14 on page 709\)](#).)

Why is a delay useful? For *DataWriters* that have multiple reliable *DataReaders*, an efficient way of heartbeating all of the *DataReaders* is to send a single heartbeat via multicast. In that case, all of the *DataReaders* will receive the heartbeat (approximately) simultaneously. If all *DataReaders* immediately respond with a ACK/NACK packet, the network may be flooded. While the size of a ACK/NACK packet is relatively small, as the number of *DataReaders* increases, the chance of packet collision also increases. All of these conditions may lead to dropped packets which forces the *DataWriter* to send out additional heartbeats that cause more simultaneous heartbeats to be sent, ultimately resulting a network packet storm.

By forcing each *DataReader* to wait for a random amount of time, bounded by the minimum and maximum values in this policy, before sending an ACK/NACK response to a heartbeat, the use of the network is spread out over a period of time, decreasing the peak bandwidth required as well as the likelihood of dropped packets due to collisions. This can increase the overall performance of the reliable connection while avoiding a network storm.

When a reliable *DataReader* first matches a reliable *DataWriter*, the *DataReader* sends periodic NACK messages at the specified period to pull historical data from the *DataWriter*. The *DataReader* will stop sending periodic NACKs when it has received all historical data available at the time that it matched the *DataWriter*. The *DataReader* ensures that at least one NACK is sent per period; for example, if, within a NACK period, the *DataReader* responds to a HEARTBEAT message with a NACK, then the *DataReader* will not send another periodic NACK.

7.6.1.4 Properties

This QosPolicy cannot be modified after the *DataReader* is created.

It only applies to *DataReaders*, so there are no restrictions for setting it compatibly with respect to *DataWriters*.

7.6.1.5 Related QosPolicies

- [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#)
- [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#)

7.6.1.6 Applicable Dds Entities

- [DataReaders \(Section 7.3 on page 459\)](#)

7.6.1.7 System Resource Considerations

Changing the values in this policy requires making tradeoffs between minimizing latency (decreasing **min_heartbeat_response_delay**), maximizing determinism (decreasing the difference between **min_heartbeat_response_delay** and **max_heartbeat_response_delay**), and minimizing network collisions/spreading out the ACK/NACK packets across a time interval (increasing the difference between **min_heartbeat_response_delay** and **max_heartbeat_response_delay** and/or shifting their values between different *DataReaders*).

If the values are poorly chosen with respect to the characteristics and requirements of a given application, the latency and/or throughput of the application may suffer.

7.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension)

The `DATA_READER_RESOURCE_LIMITS` QosPolicy extends your control over the memory allocated by Connex DDS for *DataReaders* beyond what is offered by the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#). `RESOURCE_LIMITS` controls memory allocation with respect to the *DataReader* itself: the number of DDS samples that it can store in the receive queue and the number of instances that it can manage simultaneously. `DATA_READER_RESOURCE_LIMITS` controls memory allocation on a per matched-*DataWriter* basis. The two are orthogonal.

This policy includes the members in [Table 7.21 DDS_DataReaderResourceLimitsQosPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 7.21 DDS_DataReaderResourceLimitsQosPolicy

Type	Field Name	Description
DDS_ Long	max_remote_writers	Maximum number of <i>DataWriters</i> from which a <i>DataReader</i> may receive DDS data samples, among all instances. For unkeyed <i>Topics</i> : max_remote_writers must = max_remote_writers_per_instance
	max_remote_writers_per_instance	Maximum number of <i>DataWriters</i> from which a <i>DataReader</i> may receive DDS data samples for a single instance. For unkeyed <i>Topics</i> : max_remote_writers must = max_remote_writers_per_instance
	max_samples_per_remote_writer	Maximum number of DDS samples received out-of-order that a <i>DataReader</i> can store from a single reliable <i>DataWriter</i> . max_samples_per_remote_writer must be <= RESOURCE_LIMITS::max_samples
	max_infos	Maximum number of DDS_SampleInfo structures that a <i>DataReader</i> can allocate. max_infos must be >= RESOURCE_LIMITS::max_samples
	initial_remote_writers	Initial number of <i>DataWriters</i> from which a <i>DataReader</i> may receive DDS data samples, including all instances. For unkeyed <i>Topics</i> : initial_remote_writers must = initial_remote_writers_per_instance
	initial_remote_writers_per_instance	Initial number of <i>DataWriters</i> from which a <i>DataReader</i> may receive DDS data samples for a single instance. For unkeyed <i>Topics</i> : initial_remote_writers must = initial_remote_writers_per_instance
	initial_infos	Initial number of DDS_SampleInfo structures that a <i>DataReader</i> will allocate.
	initial_outstanding_reads	Initial number of times in which memory can be concurrently loaned via read/take calls without being returned with return_loan() .
	max_outstanding_reads	Maximum number of times in which memory can be concurrently loaned via read/take calls without being returned with return_loan() .
	max_samples_per_read	Maximum number of DDS samples that can be read/taken on a <i>DataReader</i> .
DDS_ Boolean	disable_fragmentation_support	Determines whether the <i>DataReader</i> can receive fragmented DDS samples. When fragmentation support is not needed, disabling fragmentation support will save some memory resources.

Table 7.21 DDS_DataReaderResourceLimitsQoSPolicy

Type	Field Name	Description
DDS_ Long	max_ fragmented_ samples	<p>The maximum number of DDS samples for which the <i>DataReader</i> may store fragments at a given point in time.</p> <p>At any given time, a <i>DataReader</i> may store fragments for up to max_fragmented_samples DDS samples while waiting for the remaining fragments. These DDS samples need not have consecutive sequence numbers and may have been sent by different <i>DataWriters</i>. Once all fragments of a DDS sample have been received, the DDS sample is treated as a regular DDS sample and becomes subject to standard QoS settings, such as max_samples. Connex DDS will drop fragments if the max_fragmented_samples limit has been reached.</p> <p>For best-effort communication, Connex DDS will accept a fragment for a new DDS sample, but drop the oldest fragmented DDS sample from the same remote writer.</p> <p>For reliable communication, Connex DDS will drop fragments for any new DDS samples until all fragments for at least one older DDS sample from that writer have been received.</p> <p>Only applies if disable_fragmentation_support is FALSE.</p>
	initial_ fragmented_ samples	<p>The initial number of DDS samples for which a <i>DataReader</i> may store fragments.</p> <p>Only applies if disable_fragmentation_support is FALSE.</p>
	max_ fragmented_ samples_per_ remote_ writer	<p>The maximum number of DDS samples per remote writer for which a <i>DataReader</i> may store fragments. This is a logical limit, so a single remote writer cannot consume all available resources.</p> <p>Only applies if disable_fragmentation_support is FALSE.</p>
	max_ fragments_ per_ sample	<p>Maximum number of fragments for a single DDS sample.</p> <p>Only applies if disable_fragmentation_support is FALSE.</p>
DDS_ Boolean	dynamically_ allocate_ fragmented_ samples	<p>By default, the middleware does not allocate memory upfront, but instead allocates memory from the heap upon receiving the first fragment of a new sample. The amount of memory allocated equals the amount of memory needed to store all fragments in the sample. Once all fragments of a sample have been received, the sample is deserialized and stored in the regular receive queue. At that time, the dynamically allocated memory is freed again.</p> <p>This QoS setting is useful for large, but variable-sized data types where up-front memory allocation for multiple samples based on the maximum possible sample size may be expensive. The main disadvantage of not pre-allocating memory is that one can no longer guarantee the middleware will have sufficient resources at run-time.</p> <p>If dynamically_allocate_fragmented_samples is FALSE, the middleware will allocate memory up-front for storing fragments for up to initial_fragmented_samples samples. This memory may grow up to max_fragmented_samples if needed.</p> <p>Only applies if disable_fragmentation_support is FALSE.</p>
DDS_ Long	max_total_ instances	<p>Maximum number of instances for which a <i>DataReader</i> will keep state.</p> <p>See max_total_instances and max_instances (Section 7.6.2.1 on page 522)</p>

Table 7.21 DDS_DataReaderResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	max_remote_virtual_writers	<p>The maximum number of virtual writers (identified by a virtual GUID) from which a <i>DataReader</i> may read, including all instances.</p> <p>When the <i>Subscriber</i>'s access_scope is GROUP, this value determines the maximum number of <i>DataWriter</i> groups supported by the <i>Subscriber</i>. Since the <i>Subscriber</i> may contain more than one <i>DataReader</i>, only the setting of the first applies.</p>
DDS_Long	initial_remote_virtual_writers	<p>The initial number of virtual writers from which a <i>DataReader</i> may read, including all instances.</p>
DDS_Long	max_remote_virtual_writers_per_instance	<p>Maximum number of virtual remote writers that can be associated with an instance.</p> <p>For unkeyed types, this value is ignored.</p> <p>The features of Durable Reader State and MultiChannel DataWriters, as well as Persistence Service^a, require Connex DDS to keep some internal state per virtual writer and instance that is used to filter duplicate DDS samples. These duplicate DDS samples could be coming from different <i>DataWriter</i> channels or from multiple executions of Persistence Service.</p> <p>Once an association between a remote virtual writer and an instance is established, it is permanent—it will not disappear even if the physical writer incarnating the virtual writer is destroyed.</p> <p>If max_remote_virtual_writers_per_instance is exceeded for an instance, Connex DDS will not associate this instance with new virtual writers. Duplicate DDS samples coming from these virtual writers will not be filtered on the reader.</p> <p>If you are not using Durable Reader State, MultiChannel <i>DataWriters</i> or Persistence Service, you can set this property to 1 to optimize resources.</p> <p>For additional information about the virtual writers see Mechanisms for Achieving Information Durability and Persistence (Section Chapter 12 on page 675).</p>
DDS_Long	initial_remote_virtual_writers_per_instance	<p>Initial number of virtual remote writers per instance.</p> <p>For unkeyed types, this value is ignored.</p>
DDS_Long	max_remote_writers_per_sample	<p>Maximum number of remote writers that are allowed to write the same DDS sample.</p> <p>One scenario in which two <i>DataWriters</i> may write the same DDS sample is when using Persistence Service. The <i>DataReader</i> may receive the same DDS sample from the original <i>DataWriter</i> and from an Persistence Service <i>DataWriter</i>.</p>

^aPersistence Service is included with the Connex DDS Professional, Evaluation, and Basic package types. It saves DDS data samples so they can be delivered to subscribing applications that join the system at a later time (see [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)).

Table 7.21 DDS_DataReaderResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	max_query_condition_filters	This value determines the maximum number of unique query condition content filters that a reader may create. Each query condition content filter is comprised of both its query_expression and query_parameters . Two query conditions that have the same query_expression will require unique query condition filters if their query_parameters differ. Query conditions that differ only in their state masks will share the same query condition filter.
DDS_Long	max_app_ack_response_length	Maximum length of application-level acknowledgment response data. The maximum length of response data in an application-level acknowledgment. When set to zero, no response data is sent with application-level acknowledgments.
DDS_Boolean	keep_minimum_state_for_instances	Determines whether the DataReader keeps a minimum instance state for up to max_total_instances . The minimum state is useful for filtering samples in certain scenarios. See max_total_instances and max_instances (Section 7.6.2.1 on the next page)

DataReaders must allocate internal structures to handle: the maximum number of *DataWriters* that may connect to it; whether or not a *DataReader* handles data fragmentation and how many data fragments that it may handle (for DDS data samples larger than the MTU of the underlying network transport); how many simultaneous outstanding loans of internal memory holding DDS data samples can be provided to user code; as well as others.

Most of these internal structures start at an initial size and, by default, will grow as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want to bound the amount of memory that can be used by a *DataReader*. Setting the initial size to the maximum size will prevent Connex DDS from dynamically allocating any memory after the *DataReader* is created.

This policy also controls how the allocated internal data structure may be used. For example, *DataReaders* need data structures to keep track of all of the *DataWriters* that may be sending it DDS data samples. The total number of *DataWriters* that it can keep track of is set by the **initial_remote_writers** and **max_remote_writers** values. For keyed Topics, **initial_remote_writers_per_instance** and **max_remote_writers_per_instance** control the number of *DataWriters* allowed by the *DataReader* to modify the value of a single instance.

By setting the max value to be less than **max_remote_writers**, you can prevent instances with many *DataWriters* from using up the resources and starving other instances. Once the resources for keeping track of *DataWriters* are used up, the *DataReader* will not be able to accept “connections” from new *DataWriters*. The *DataReader* will not be able to receive data from new matching *DataWriters* which would be ignored.

In the reliable protocol used by Connex DDS to support a RELIABLE setting for the [RELIABILITY QosPolicy](#) (Section 6.5.19 on page 400), the *DataReader* must temporarily store DDS data samples that have been received out-of-order from a reliable *DataWriter*. The storage of out-of-order DDS samples is

allocated from the *DataReader*'s receive queue and shared among all reliable *DataWriters*. The parameter **max_samples_per_remote_writer** controls the maximum number of out-of-order data DDS samples that the *DataReader* is allowed to store for a single *DataWriter*. This value must be less than the **max_samples** value set in the [RESOURCE_LIMITS QoSPolicy \(Section 6.5.20 on page 405\)](#).

max_samples_per_remote_writer allows Connex DDS to share the limited resources of the *DataReader* equitably so that a single *DataWriter* is unable to use up all of the storage of the *DataReader* while missing DDS data samples are being resent.

When setting the values of the members, the following rules apply:

- **max_remote_writers** >= **initial_remote_writers**
- **max_remote_writers_per_instance** >= **initial_remote_writers_per_instance**
max_remote_writers_per_instance <= **max_remote_writers**
- **max_infos** >= **initial_infos**
max_infos >= RESOURCE_LIMITS::max_samples
- **max_outstanding_reads** >= **initial_outstanding_reads**
- **max_remote_writers** >= **max_remote_writers_per_instance**
- **max_samples_per_remote_writer** <= RESOURCE_LIMITS::max_samples

If any of the above are false, Connex DDS returns the error code **DDS_RETCODE_INCONSISTENT_POLICY** when setting the *DataReader*'s QoS.

7.6.2.1 max_total_instances and max_instances

The features [Durable Reader State \(Section 12.4 on page 686\)](#), [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#), and [Persistence Service \(Part 6: RTI Persistence Service \(Section on page 932\)\)](#) require Connex DDS to keep some internal state even for instances without *DataWriters* or DDS samples in the *DataReader*'s queue or that have been purged due to a dispose. The additional state is used to filter duplicate DDS samples that could be coming from different *DataWriter* channels or from multiple executions of Persistence Service. The total maximum number of instances that will be managed by the middleware, including instances without associated *DataWriters* or DDS samples or that have been purged due to a dispose, is determined by **max_total_instances**. This additional state will only be kept for up to **max_total_instances** if **keep_minimum_state_for_instances** is TRUE, otherwise the additional state will not be kept for any instances.

7.6.2.2 Example

The **max_samples_per_remote_writer** value affects sharing and starvation. **max_samples_per_remote_writer** can be set to less than the RESOURCE_LIMITS QoSPolicy's **max_samples** to prevent a single

DataWriter from starving others. This control is especially important for *Topics* that have their **OWNERSHIP** QoSPolicy (Section 6.5.15 on page 389) set to **SHARED**.

In the case of **EXCLUSIVE** ownership, a lower-strength remote *DataWriter* can "starve" a higher-strength remote *DataWriter* by making use of more of the *DataReader*'s resources, an undesirable condition. In the case of **SHARED** ownership, a remote *DataWriter* may starve another remote *DataWriter*, making the sharing not really equal.

7.6.2.3 Properties

This QoSPolicy cannot be modified after the *DataReader* is created.

It only applies to *DataReaders*, so there are no restrictions for setting it compatibly on the *DataWriter*.

7.6.2.4 Related QoS Policies

- **RESOURCE_LIMITS** QoSPolicy (Section 6.5.20 on page 405)
- **OWNERSHIP** QoSPolicy (Section 6.5.15 on page 389)

7.6.2.5 Applicable Dds Entities

- *DataReaders* (Section 7.3 on page 459)

7.6.2.6 System Resource Considerations

Increasing any of the “initial” values in this policy will increase the amount of memory allocated by Connext DDS when a new *DataReader* is created. Increasing any of the “max” values will not affect the initial memory allocated for a new *DataReader*, but will affect how much additional memory may be allocated as needed over the *DataReader*'s lifetime.

Setting a max value greater than an initial value thus allows your application to use memory more dynamically and efficiently in the event that the size of the application is not well-known ahead of time. However, Connext DDS may dynamically allocate memory in response to network communications.

7.6.3 READER_DATA_LIFECYCLE QoS Policy

This policy controls the behavior of the *DataReader* with regards to the lifecycle of the data instances it manages, that is, the data instances that have been received and for which the *DataReader* maintains some internal resources.

When a *DataReader* receives data, it is stored in a receive queue for the *DataReader*. The user application may either take the data from the queue or leave it there. This QoS controls whether or not Connext DDS will automatically remove data from the receive queue (so that user applications cannot access it afterwards) when Connext DDS detects that there are no more *DataWriters* alive for that data.

DataWriters may also call **dispose()** on its data, informing *DataReaders* that the data no longer exists. This QoSPolicy also controls whether or not Connex DDS automatically removes disposed data from the receive queue.

For keyed Topics, the consideration of removing DDS data samples from the receive queue is done on a per instance (key) basis. Thus when Connex DDS detects that there are no longer *DataWriters* alive for a certain key value for a *Topic* (an instance of the *Topic*), it can be configured to remove all DDS data samples for a certain instance (key). *DataWriters* also can dispose its data on a per instance basis. Only the DDS data samples of disposed instances would be removed by Connex DDS if so configured.

This policy helps purge untaken DDS samples from not-alive-instances and thus may prevent a *DataReader* from reclaiming resources. With this policy, the untaken DDS samples from not-alive-instances are purged and treated as if the DDS samples were taken after the specified amount of time.

The *DataReader* internally maintains the DDS samples that have not been taken by the application, subject to the constraints imposed by other QoS policies such as [HISTORY QoSPolicy \(Section 6.5.10 on page 376\)](#) and [RESOURCE_LIMITS QoSPolicy \(Section 6.5.20 on page 405\)](#).

The *DataReader* also maintains information regarding the identity, view-state, and instance-state of data instances, even after all DDS samples have been ‘taken’ (see [Accessing DDS Data Samples with Read or Take \(Section 7.4.3 on page 493\)](#)). This is needed to properly compute the states when future DDS samples arrive.

Under normal circumstances, a *DataReader* can only reclaim all resources for instances for which there are no *DataWriters* and for which all DDS samples have been ‘taken.’ The last DDS sample taken by the *DataReader* for that instance will have an instance state of NOT_ALIVE_NO_WRITERS or NOT_ALIVE_DISPOSED_INSTANCE (depending on whether or not the instance was disposed by the last *DataWriter* that owned it.) If you are using the default (infinite) values for this QoSPolicy, this behavior can cause problems if the application does not ‘take’ those DDS samples for some reason. The ‘untaken’ DDS samples will prevent the *DataReader* from reclaiming the resources and they would remain in the *DataReader* indefinitely.

A *DataReader* can also reclaim all resources for instances that have an instance state of NOT_ALIVE_DISPOSED and for which all DDS samples have been ‘taken’. *DataReaders* will only reclaim resources in this situation when **autopurge_disposed_instances_delay** has been set to zero.

It includes the members in [Table 7.22 DDS_ReaderDataLifecycleQoSPolicy](#).

Table 7.22 DDS_ReaderDataLifecycleQoSPolicy

Type	Field Name	Description
DDS_Duration_t	autopurge_nowriter_samples_delay	How long the <i>DataReader</i> maintains information about an instance once its instance_state becomes NOT_ALIVE_NO_WRITERS.

Table 7.22 DDS_ReaderDataLifecycleQosPolicy

Type	Field Name	Description
DDS_Duration_t	autopurge_disposed_samples_delay	How long the <i>DataReader</i> maintains information about an instance once its instance_state becomes NOT_ALIVE_DISPOSED.
DDS_Duration_t	autopurge_disposed_instances_delay	How long the <i>DataReader</i> maintains information about an instance once its instance_state becomes NOT_ALIVE_DISPOSED. (Note: only values of 0 or INFINITE are currently supported).

autopurge_nowriter_samples_delay: This defines the minimum duration for which the *DataReader* will maintain information regarding an instance once its **instance_state** becomes **NOT_ALIVE_NO_WRITERS**. After this time elapses, the *DataReader* will purge all internal information regarding the instance, any untaken DDS samples will also be lost.

autopurge_disposed_samples_delay: This defines the minimum duration for which the *DataReader* will maintain DDS samples of an instance once its **instance_state** becomes **NOT_ALIVE_DISPOSED**. After this time elapses, the *DataReader* will purge all internal information regarding the instance; any untaken DDS samples will also be lost.

autopurge_disposed_instances_delay: This defines the minimum duration for which the *DataReader* will maintain DDS samples of an instance once its instance_state becomes **NOT_ALIVE_DISPOSED**. After this time elapses, the *DataReader* will purge all internal information regarding the instance.

7.6.3.1 Properties

This QoS policy *can* be modified after the *DataReader* is enabled.

It only applies to *DataReaders*, so there are no RxO restrictions for setting it compatibly on the *DataWriter*.

7.6.3.2 Related QoS Policies

- [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#)
- [LIVELINESS QoS Policy \(Section 6.5.13 on page 382\)](#)
- [OWNERSHIP QoS Policy \(Section 6.5.15 on page 389\)](#)
- [RESOURCE_LIMITS QoS Policy \(Section 6.5.20 on page 405\)](#)
- [WRITER_DATA_LIFECYCLE QoS Policy \(Section 6.5.27 on page 419\)](#)

7.6.3.3 Applicable Dds Entities

- [DataReaders \(Section 7.3 on page 459\)](#)

7.6.3.4 System Resource Considerations

None.

7.6.4 TIME_BASED_FILTER QoSPolicy

The `TIME_BASED_FILTER` QoSPolicy allows you to specify that data should not be delivered more than once per specified period for data-instances of a *DataReader*—regardless of how fast *DataWriters* are publishing new DDS samples of the data-instance.

This QoS policy allows you to optimize resource usage (CPU and possibly network bandwidth) by only delivering the required amount of data to different *DataReaders*.

DataWriters may send data faster than needed by a *DataReader*. For example, a *DataReader* of sensor data that is displayed to a human operator in a GUI application does not need to receive data updates faster than a user can reasonably perceive changes in data values. This is often measure in tenths (0.1) of a second up to several seconds. However, a *DataWriter* of sensor information may have *DataReaders* that are processing the sensor information to control parts of the system and thus need new data updates in measures of hundredths (0.01) or thousandths (0.001) of a second.

With this QoS policy, different *DataReaders* can set their own time-based filters, so that data published faster than the period set by a *DataReader* will be dropped by the middleware and not delivered to the *DataReader*. Note that all filtering takes place on the reader side.

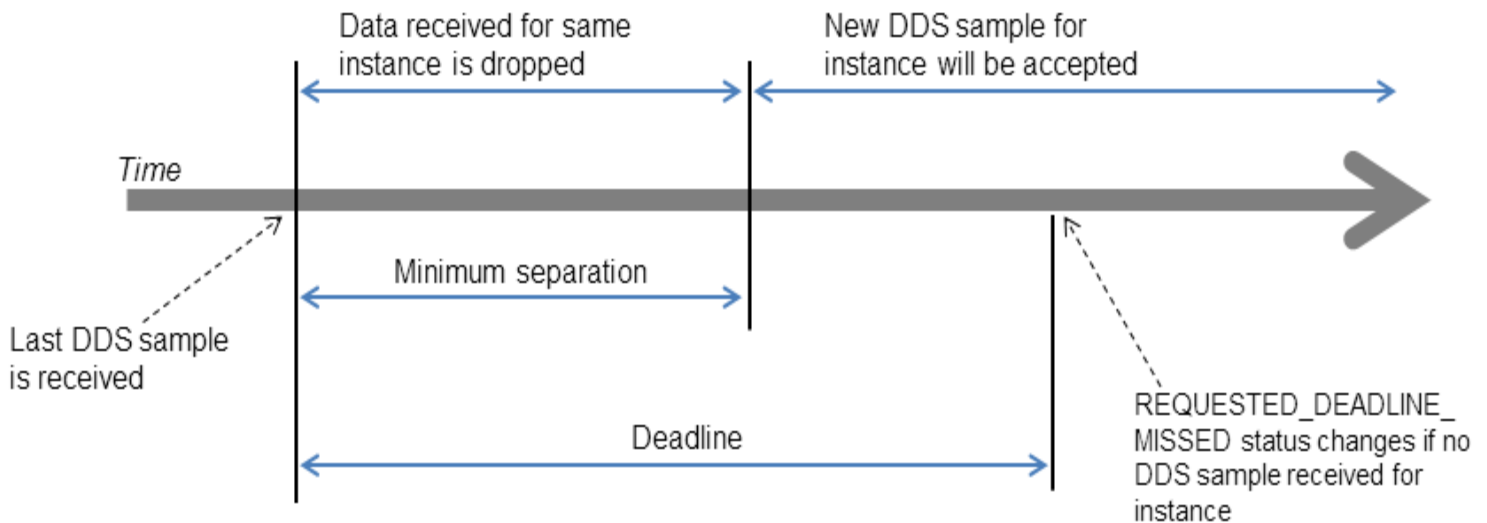
It includes the member in [Table 7.23 DDS_TimeBasedFilterQoSPolicy](#). For the default and valid range, please refer to the API Reference HTML documentation.

Table 7.23 DDS_TimeBasedFilterQoSPolicy

Type	Field Name	Description
DDS_Duration_t	minimum_separation	Minimum separation time between DDS samples of the same instance. Must be <= DEADLINE::period

As seen in [Accepting Data for DataReaders \(Section Figure 7.19 on the facing page\)](#), it is inconsistent to set a *DataReader*'s **minimum_separation** longer than its [DEADLINE QoSPolicy \(Section 6.5.5 on page 363\) period](#).

Figure 7.19 Accepting Data for DataReaders



DDS data samples for a *DataReader* can be filtered out using the `TIME_BASED_FILTER` QoS (`minimum_separation`). Once a DDS sample for an instance has been received, Connex DDS will accept but drop any new data samples for the same instance that arrives within the time specified by `minimum_separation`. After the `minimum_separation`, a new DDS sample that arrives is accepted and stored in the receive queue, and the timer starts again. If no DDS samples arrive by the `DEADLINE`, the `REQUESTED_DEADLINE_MISSED` status will be changed and Listeners called back if installed.

This QoS Policy allows a *DataReader* to subsample the data being published for a data instance by *DataWriters*. If a user application only needs new DDS samples for a data instance to be received at a specified period, then there is no need for Connex DDS to deliver data faster than that period. However, whether or not data being published by a *DataWriter* at a faster rate than set by the `TIME_BASED_FILTER` QoS is sent on the wire depends on several factors, including whether the *DataReader* is receiving the data reliably and if the data is being sent via multicast for multiple *DataReaders*.

For best effort data delivery, if the data type is unkeyed and the *DataWriter* has an infinite liveliness **lease_duration** ([LIVELINESS QoS Policy \(Section 6.5.13 on page 382\)](#)), Connex DDS will only send as many packets to a *DataReader* as required by the `TIME_BASED_FILTER`, no matter how fast the *DataWriter*'s `write()` function is called.

For multicast data delivery to multiple *DataReaders*, the *DataReader* with the lowest `TIME_BASED_FILTER` **minimum_separation** determines the *DataWriter*'s send rate. For example, if a *DataWriter* sends multicast to two *DataReaders*, one with **minimum_separation** of 2 seconds and one with **minimum_separation** of 1 second, the *DataWriter* will send every 1 second.

Other configurations (for example, when the *DataWriter* is reliable, or the data type is keyed, or the *DataWriter* has a finite liveliness **lease_duration**) must send all data published by the *DataWriter*. On reception, only the data that passes the `TIME_BASED_FILTER` will be stored in the *DataReader*'s

receive queue. Extra data will be accepted but dropped. Note that filtering is only applied on ‘alive’ DDS samples (that is, DDS samples that have *not* been disposed/unregistered).

7.6.4.1 Example

The purpose of this QosPolicy is to prevent fast *DataWriters* from overwhelming a *DataReader* that cannot process the data at the rate the data is being published. In certain configurations, the number of packets sent by Connex DDS can also be reduced thus minimizing the consumption of network bandwidth.

You may want to change the **minimum_separation** between DDS data samples for one or more of the following reasons:

- The *DataReader* is connected to the network via a low-bandwidth connection that is unable to sustain the amount of traffic generated by the matched *DataWriter(s)*.
- The rate at which the matched *DataWriter(s)* can generate DDS samples is faster than the rate at which the *DataReader* can process them. Or faster than needed by the *DataReader*. For example, a graphical user interface seldom needs to be updated faster than 30 times a second, even if new data values are available much faster.
- The resource limits of the *DataReader* are constrained relative to the number of DDS samples that could be generated by the matched *DataWriter(s)*. Too many packets coming at once will cause them to be exhausted before the *DataReader* has time to process them.

7.6.4.2 Properties

This QosPolicy can be modified at any time.

It only applies to *DataReaders*, so there are no restrictions for setting it compatibly on the *DataWriter*.

7.6.4.3 Related QosPolicies

- [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#)
- [DEADLINE QosPolicy \(Section 6.5.5 on page 363\)](#)
- [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\) \(Section 7.6.5 on the facing page\)](#)

7.6.4.4 Applicable Dds Entities

- [DataReaders \(Section 7.3 on page 459\)](#)

7.6.4.5 System Resource Considerations

Depending on the values of other QosPolicies such as RELIABILITY and TRANSPORT_MULTICAST, this policy may be able to decrease the usage of network bandwidth and CPU by preventing unneeded packets from being sent and processed.

7.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension)

This QosPolicy specifies the multicast address on which a *DataReader* wants to receive its data. It can also specify a port number as well as a subset of the available transports with which to receive the multicast data.

By default, *DataWriters* will send individually addressed packets for each *DataReader* that subscribes to the topic of the *DataWriter*—this is known as unicast delivery. Thus, as many copies of the data will be sent over the network as there are *DataReaders* for the data. The network bandwidth used by a *DataWriter* will thus increase linearly with the number of *DataReaders*.

Multicast is a concept supported by some transports, most notably UDP/IP, so that a *single* packet on the network can be addressed such that it is received by multiple nodes. This is more efficient when the same data needs to be sent to multiple nodes. By using multicast, the network bandwidth usage will be constant, independent of the number of *DataReaders*.

Coordinating the multicast address specified by *DataReaders* can help optimize network bandwidth usage in systems where there are multiple *DataReaders* for the same *Topic*.

The QosPolicy structure includes the members in [Table 7.24 DDS_TransportMulticastQosPolicy](#).

Table 7.24 DDS_TransportMulticastQosPolicy

Type	Field Name	Description
DDS_TransportMulticastSettingSeq (A sequence of the type shown in Table 7.25 DDS_TransportMulticastSetting_t)	value	A sequence of multicast locators. (See Locator Format (Section 14.2.1.1 on page 714) .)
DDS_TransportMulticastKind	kind	<p>This field can be set to one of the following two values: DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS or DDS_UNICAST_ONLY_TRANSPORT_MULTICAST_QOS.</p> <p>If it is set to DDS_AUTOMATIC_TRANSPORT_MULTICAST_QOS, the behavior depends on the content of DDS_TransportMulticastQosPolicy::value:</p> <p>If DDS_TransportMulticastQosPolicy::value does not have any elements, multicast will not be used.</p> <p>If DDS_TransportMulticastQosPolicy::value first element has an empty address, the address will be obtained from DDS_TransportMulticastMappingQosPolicy.</p> <p>If none of the elements in DDS_TransportMulticastQosPolicy::value are empty, and at least one element has a valid address, then that address will be used.</p> <p>If it is set to DDS_UNICAST_ONLY_TRANSPORT_MULTICAST_QOS, then multicast will not be used.</p>

Table 7.25 DDS_TransportMulticastSetting_t

Type	Field Name	Description
DDS_StringSeq	transports	A sequence of transport aliases that specifies which transports should be used to receive multicast messages for this <i>DataReader</i> .
char *	receive_address	A multicast group address to which the <i>DataWriter</i> should send data for this <i>DataReader</i> .
DDS_Long	receive_port	The port that should be used in the addressing of multicast messages destined for this <i>DataReader</i> . A value of 0 will cause Connex DDS to use a default port number based on domain ID. See Ports Used for Discovery (Section 14.5 on page 738) .

To take advantage of multicast, the value of this QoSPolicy must be coordinated among all of the applications on a network for *DataReaders* of the same *Topic*. For a *DataWriter* to send a single packet that will be received by all *DataReaders* simultaneously, the same multicast address must be used.

To use this QoSPolicy, you will also need to specify a port number. A port number of 0 will cause Connex DDS to automatically use a default value. As explained in [Ports Used for Discovery \(Section 14.5 on page 738\)](#), the default port number for multicast addresses is based on the domain ID. Should you choose to use a different port number, then for every unique port number used by Entities in your application, depending on the transport, Connex DDS may create a thread to process messages received for that port on that transport. See [Connex DDS Threading Model \(Section Chapter 19 on page 837\)](#) for more about threads.

Threads are created on a per-transport basis, so if this QoSPolicy specifies multiple **transports** for a **receive_port**, then a thread may be created for each transport for that unique port. Some transports may be able to share a single thread for different ports, others can not. Note that different Entities can share the same port number, and thus, the same thread will process all of the data for all of the Entities sharing the same port number for a transport.

Also note that if the port number specified by this QoS is the same as a port number specified by a TRANSPORT_UNICAST QoS, then the transport may choose to process data received both via multicast and unicast with a single thread. Whether or not a transport must use different threads to process data received via multicast or unicast for the same port number depends on the implementation of the transport.

Notes:

- The same multicast address can be used by *DataReaders* of different *Topics*.
- Even though the TRANSPORT_MULTICAST QoS allows you to specify multiple multicast addresses for a *DataReader*, Connex DDS currently only uses one multicast address (the first in the sequence) per *DataReader*.

- If a *DataWriter* is using the [MULTI_CHANNEL QoS Policy \(DDS Extension\)](#) (Section 6.5.14 on page 386), the multicast addresses specified in the `TRANSPORT_MULTICAST` QoS Policy are ignored by that *DataWriter*. The *DataWriter* will not publish DDS samples on those locators.

7.6.5.1 Example

In an airport, there may be many different monitors that display current flight information. Assuming each monitor is controlled by a networked application, network bandwidth would be greatly reduced if flight information was published using multicast.

[Figure 7.20 Setting Up a Multicast DataReader below](#) shows an example of how to set this QoS Policy.

Figure 7.20 Setting Up a Multicast DataReader

```
...
DDS_DataReaderQos  reader_qos;
reader_listener = new HelloWorldListener();
if (reader_listener == NULL) {
    // handle error
}
// Get default data reader QoS to customize
retcode = subscriber->get_default_datareader_qos(reader_qos);
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Set up multicast reader
reader_qos.multicast.value.ensure_length(1,1);
reader_qos.multicast.value[0].receive_address =
    DDS_String_dup("239.192.0.1");
reader = subscriber->create_datareader(
    topic, reader_qos,
    reader_listener, DDS_STATUS_MASK_ALL);
```

7.6.5.2 Properties

This QoS Policy cannot be modified after the *Entity* is created.

For compatibility between *DataWriters* and *DataReaders*, the *DataWriter* must be able to send to the multicast address that the *DataReader* has specified.

7.6.5.3 Related QoS Policies

- [MULTI_CHANNEL QoS Policy \(DDS Extension\)](#) (Section 6.5.14 on page 386)
- [TRANSPORT_UNICAST QoS Policy \(DDS Extension\)](#) (Section 6.5.24 on page 412)
- [TRANSPORT_BUILTIN QoS Policy \(DDS Extension\)](#) (Section 8.5.7 on page 606)

7.6.5.4 Applicable DDS Entities

- [DomainParticipants](#) (Section 8.3 on page 547)
- [DataReaders](#) (Section 7.3 on page 459)

7.6.5.5 System Resource Considerations

On Ethernet-based systems, the number of multicast addresses that can be “listened” to by the network interface card is usually limited. The exact number of multicast addresses that can be monitored simultaneously by a NIC depends on its manufacturer. Setting a multicast address for a *DataReader* will use up one of the multicast-address slots of the NIC.

What happens if the number of different multicast addresses used by different *DataReaders* across different applications on the same node exceeds the total number supported by a NIC depends on the specific operating system. Some will prevent you from configuring too many multicast addresses to be monitored.

Many operating systems will accommodate the extra multicast addresses by putting the NIC in promiscuous mode. This means that the NIC will pass every Ethernet packet to the operating system, and the operating system will pass the packets with the specified multicast addresses to the application(s). This results in extra CPU usage. We recommend that your applications do not use more multicast addresses on a single node than the NICs on that node can listen to simultaneously in hardware.

Depending on the implementation of a transport, Connex DDS may need to create threads to receive and process data on a unique-port-number basis. Some transports can share the same thread to process data received for different ports; others like UDPv4 must have different threads for different ports. In addition, if the same port is used for both unicast and multicast, the transport implementation will determine whether or not the same thread can be used to process both unicast and multicast data. For UDPv4, only one thread is needed per port— independent of whether the data was received via unicast or multicast data. See [Receive Threads](#) (Section 19.3 on page 839) for more information.

7.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy

The `TypeConsistencyEnforcementQosPolicy` defines the rules that determine whether the type used to publish a given topic is consistent with the type used to subscribe to it.

The `QosPolicy` structure includes the member in [Table 7.26 DDS_TypeConsistencyEnforcementQosPolicy](#).

Table 7.26 DDS_TypeConsistencyEnforcementQosPolicy

Type	Field Name	Description
DDS_TypeConsistencyKind	kind	<p>Can be either:</p> <ul style="list-style-type: none"> • DISALLOW_TYPE_COERCION • ALLOW_TYPE_COERCION (default) <p>See Values for TypeConsistencyKind (Section below) for details.</p>

The type-consistency enforcement rules consist of two steps:

1. If both the *DataWriter* and *DataReader* specify a *TypeObject*, it is considered first. If the *DataReader* allows type coercion, then its type must be assignable from the *DataWriter*'s type. If the *DataReader* does not allow type coercion, then its type must be structurally identical to the type of the *DataWriter*.
2. If either the *DataWriter* or the *DataReader* does not provide a *TypeObject* definition, then the registered type names are examined. The *DataReader*'s and *DataWriter*'s registered type names must match exactly.

If either Step 1 or Step 2 fails, the *Topics* associated with the *DataReader* and *DataWriter* are considered to be inconsistent and the [INCONSISTENT_TOPIC Status \(Section 5.3.1 on page 211\)](#) is updated.

The default enforcement kind is **DDS_ALLOW_TYPE_COERCION**. However, when the middleware is introspecting the built-in topic data declaration of a remote *DataReader* in order to determine whether it can match with a local *DataWriter*, if it observes that no *TypeConsistencyEnforcementQosPolicy* value is provided (as would be the case when communicating with a Service implementation not in conformance with this specification), it assumes a kind of **DDS_DISALLOW_TYPE_COERCION**.

Values for TypeConsistencyKind

- **DISALLOW_TYPE_COERCION**

With this setting, the *DataWriter* and *DataReader* must support the same data type in order for them to communicate. (This is the degree of enforcement required by the OMG DDS Specification prior to the [OMG 'Extensible and Dynamic Topic Types for DDS' Specification](#).)

When Connex DDS is introspecting the built-in topic data declaration of a remote *DataWriter* or *DataReader*, if no *TypeConsistencyEnforcementQosPolicy* value is provided (as would be the case when communicating with an implementation not in conformance with the Extensible and Dynamic Topic Types for DDS" (DDS-XTypes) specification), Connex DDS shall assume a **kind** of **DISALLOW_TYPE_COERCION**.

- **ALLOW_TYPE_COERCION (default)**

With this setting, the *DataWriter* and the *DataReader* need not support the same data type in order for them to communicate, as long as the *DataReader*'s type is assignable from the *DataWriter*'s type.

For example, the following two extensible types will be assignable to each other since *MyDerivedType* contains all the members of *MyBaseType* (**member_1**) plus an additional element (**member_2**).

```
struct MyBaseType {
    long member_1;
};
struct MyDerivedType: MyBaseType {
    long member_2;
};
```

Even if *MyDerivedType* was not explicitly inherited from *MyBaseType*, the types would still be assignable. For example:

```
struct MyBaseType {
    long member_1;
};
struct MyDerivedType {
    long member_1;
    long member_2;
};
```

For more information, see the [RTI Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types](#) and the [OMG 'Extensible and Dynamic Topic Types for DDS' Specification](#).

7.6.6.1 Properties

This *QoS*Policy cannot be modified after the *DataReader* is enabled.

It only applies to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

7.6.6.2 Related QoS Policies

- None.

7.6.6.3 Applicable Entities

- [DataReaders \(Section 7.3 on page 459\)](#)

7.6.6.4 System Resource Considerations

None.

Chapter 8 Working with DDS Domains

This section discusses how to use *DomainParticipants*. It describes the types of operations that are available for them and their QoS Policies.

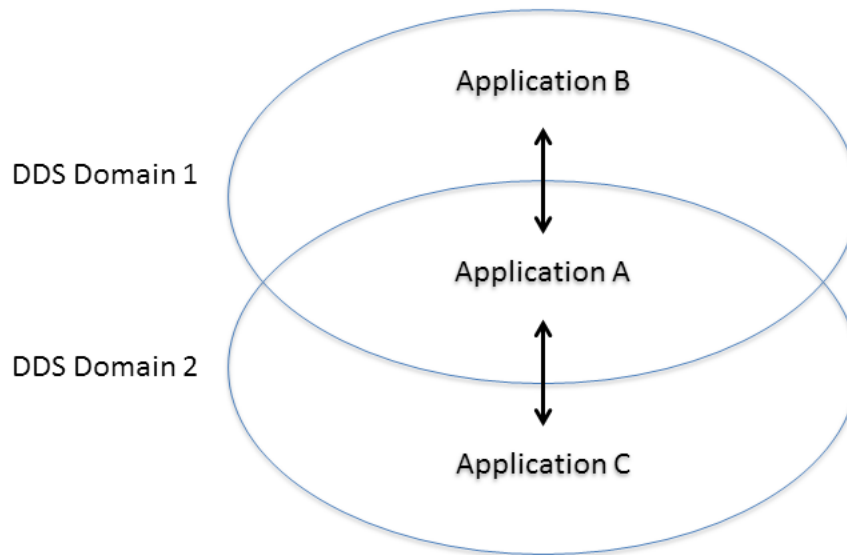
This section includes:

The goal of this section is to help you become familiar with the objects you need for setting up your Connex DDS application. For specific details on any mentioned operations, see the API Reference HTML documentation.

8.1 Fundamentals of DDS Domains and DomainParticipants

DomainParticipants are the focal point for creating, destroying, and managing other Connex DDS objects. A *DDS domain* is a logical network of applications: only applications that belong to the same DDS domain may communicate using Connex DDS. A DDS domain is identified by a unique integer value known as a domain ID. An application participates in a DDS domain by creating a *DomainParticipant* for that domain ID.

Figure 8.1 Relationship between Applications and DDS Domains



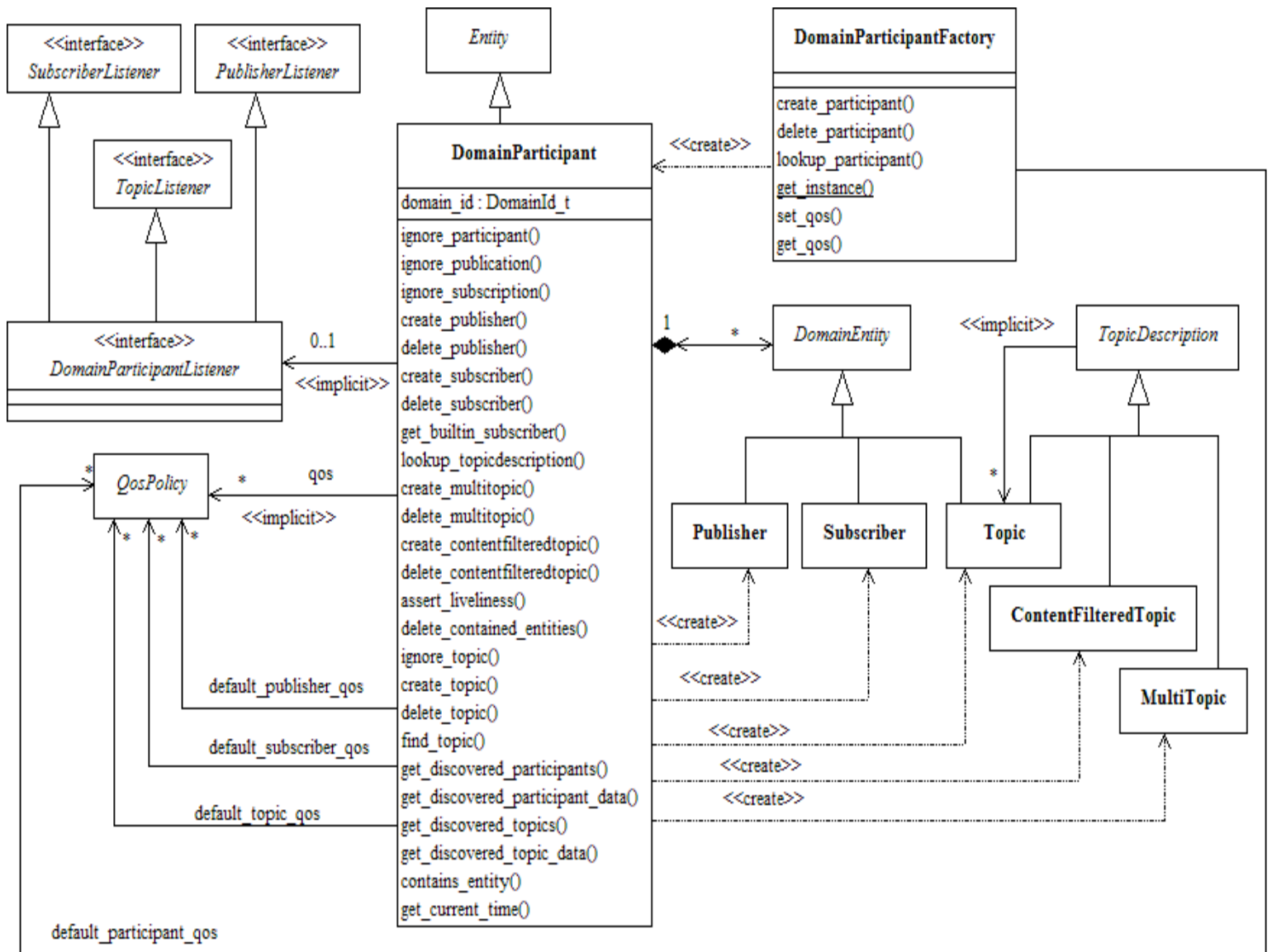
Applications can belong to multiple DDS domains—A belongs to DDS domains 1 and 2. Applications in the same DDS domain can communicate with each other, such as A and B, or A and C. Applications in different DDS domains, such as B and C, are not even aware of each other and will not exchange messages.

As seen in [Figure 8.1 Relationship between Applications and DDS Domains](#) above, a single application can participate in multiple DDS domains by creating multiple *DomainParticipants* with different domain IDs. *DomainParticipants* in the same DDS domain form a logical network; they are isolated from *DomainParticipants* of other DDS domains, even those running on the same set of physical computers sharing the same physical network. *DomainParticipants* in different DDS domains will never exchange messages with each other. Thus, a DDS domain establishes a “virtual network” linking all *DomainParticipants* that share the same domain ID.

An application that wants to participate in a certain DDS domain will need to create a *DomainParticipant*. As seen in [Figure 8.2 DDS Domain Module on the facing page](#), a *DomainParticipant* object is a container for all other *Entities* that belong to the same DDS domain. It acts as factory for the *Publisher*, *Subscriber*, and *Topic* entities. (As seen in [Sending Data \(Section Chapter 6 on page 242\)](#) and [Receiving Data \(Section Chapter 7 on page 437\)](#), in turn, *Publishers* are factories for *DataWriters* and *Subscribers* are factories for *DataReaders*.) *DomainParticipants* cannot contain other *DomainParticipants*.

Like all *Entities*, *DomainParticipants* have *QosPolicies* and *Listeners*. The *DomainParticipant* entity also allows you to set ‘default’ values for the *QosPolicies* for all the entities created from it or from the entities that it creates (*Publishers*, *Subscribers*, *Topics*, *DataWriters*, and *DataReaders*).

Figure 8.2 DDS Domain Module



Note: MultiTopics are not supported.

8.2 DomainParticipantFactory

- C, Traditional C++, Java and .NET APIs:

The main purpose of a *DomainParticipantFactory* is to create and destroy *DomainParticipants*.

In C++ terms, this is a singleton class; that is, you will only have a single *DomainParticipantFactory* in an application—no matter how many *DomainParticipants* the application may create. [Figure 8.3 Instantiating a DomainParticipantFactory below](#) shows how to instantiate a *DomainParticipantFactory*. Notice that there are no parameters to specify. Alternatively, in C++, C++/CLI, and C#, the predefined macro, **DDSTheParticipantFactory**,¹ can also be used to retrieve the singleton factory.

Unlike the other *Entities* that you create, the *DomainParticipantFactory* does not have an associated *Listener*. However, it does have associated QoS Policies, see [Setting DomainParticipantFactory QoS Policies \(Section 8.2.1 on page 543\)](#). You can change them using the factory's **get_qos()** and **set_qos()** operations. The *DomainParticipantFactory* also stores the default QoS settings that can be used when a *DomainParticipant* is created. These default settings can be changed as well, see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on page 568\)](#).

Figure 8.3 Instantiating a DomainParticipantFactory

```
DDSDomainParticipantFactory* factory = NULL;
factory = DDSDomainParticipantFactory::get_instance();
if (factory == NULL) {
    // ... error
}
```

¹In C, the macro is `DDS_TheParticipantFactory`. In Java, use the static class method `DomainParticipantFactory.TheParticipantFactory`.

- Modern C++ API:

In the Modern C++ API, there isn't an explicit `DomainParticipantFactory`. *DomainParticipants* are created using their constructors and are automatically destroyed as a reference type (See [Creating and Deleting DDS Entities \(Section 4.1.1 on page 153\)](#)).

The operations to set and get the default `DomainParticipantQos` are static functions in *DomainParticipant*: **`DomainParticipant::default_participant_qos()`**. The operations to look up participants are freestanding functions in the **`dds::domain`** and **`rti::domain`** namespaces: **`dds::domain::find()`**, **`rti::domain::find_participant_by_name()`**, and **`rti::domain::find_participants()`**. The class `QosProvider` is responsible for managing QoS profiles (see [How to Load XML-Specified QoS Settings \(Section 17.5 on page 810\)](#)).

There is a `DomainParticipantFactoryQos`, but it only contains the `ENTITY_FACTORY` to indicate if a *DomainParticipant* should be enabled in its constructor or by calling **`enable()`**, and `SYSTEM_RESOURCE_LIMITS`. The `DomainParticipantFactoryQos` getter and setter are static functions in *DomainParticipant*: **`DomainParticipant::participant_factory_qos()`**.

Another static function in *DomainParticipant* allows finalizing the implicit `DomainParticipantFactory` singleton: **`DomainParticipant::finalize_participant_factory()`**.

Once you have a *DomainParticipantFactory*, you can use it to perform the operations listed in [Table 8.1 DomainParticipantFactory Operations](#). The most important one is **`create_participant()`**, described in [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#). For more details on all operations, see the API Reference HTML documentation as well as the section of the manual listed in the Reference column.

Table 8.1 DomainParticipantFactory Operations

Working with ...	Operation	Description	Reference
Domain-Participants	create_participant	Creates a <i>DomainParticipant</i> .	Creating a <i>DomainParticipant</i> (Section 8.3.1 on page 556)
	create_participant_with_profile	Creates a <i>DomainParticipant</i> based on a QoS profile.	
	delete_participant	Deletes a <i>DomainParticipant</i> .	Deleting <i>DomainParticipants</i> (Section 8.3.2 on page 558)
	get_default_participant_qos	Gets the default QoS for <i>DomainParticipants</i> .	Getting and Setting Default QoS for <i>DomainParticipants</i> (Section 8.2.2 on page 545)
	get_participants	Returns a sequence of pointers to all the <i>DomainParticipants</i> within the <i>DomainParticipantFactory</i> .	Looking Up <i>DomainParticipants</i> (Section 8.2.4 on page 546)
	lookup_participant	Finds a specific <i>DomainParticipant</i> , based on a domain ID.	
	lookup_participant_by_name	Finds a specific <i>DomainParticipant</i> , based on a domain name.	
	set_default_participant_qos	Sets the default QoS for <i>DomainParticipants</i> .	Getting and Setting Default QoS for <i>DomainParticipants</i> (Section 8.2.2 on page 545)
	set_default_participant_qos_with_profile	Sets the default QoS for <i>DomainParticipants</i> based on a QoS profile.	
The Factory's Instance	get_instance	Gets the singleton instance of this class.	Freeing Resources Used by the <i>DomainParticipantFactory</i> (Section 8.2.3 on page 546)
	finalize_instance	Destroys the singleton instance of this class.	
The Factory's Own QoS	get_qos	Gets/sets the <i>DomainParticipantFactory</i> 's QoS.	Getting, Setting, and Comparing QoS Policies (Section 4.1.7 on page 158)
	set_qos		
	equals	Compares two <i>DomainParticipantFactory</i> 's QoS structures for equality.	

Table 8.1 DomainParticipantFactory Operations

Working with ...	Operation	Description	Reference
Threads	set_thread_factory	Specifies a ThreadFactory implementation that DomainParticipants will use to create and delete all threads.	User-Managed Threads (Section 19.7 on page 844)
	unregister_thread	Frees all resources related to a thread. This function is intended to be used at the end of any user-created threads that invoke Connex DDS APIs (not all users will have this situation). The best approach is to call it immediately before exiting such a thread, after all Connex DDS APIs have been called.	

Table 8.1 DomainParticipantFactory Operations

Working with ...	Operation	Description	Reference
Profiles & Libraries	get_default_library	Gets the default library for a DomainParticipantFactory.	Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library (Section 8.2.1.1 on the facing page)
	get_default_profile	Gets the default QoS profile for a DomainParticipantFactory.	
	get_default_profile_library	Gets the library that contains the default QoS profile for a <i>DomainParticipantFactory</i> .	
	get_<entity>_qos_from_profile	Gets the <entity> QoS values associated with a specified QoS profile. <entity> may be <i>topic</i> , <i>datareader</i> , <i>datawriter</i> , <i>subscriber</i> , <i>publisher</i> , or <i>participant</i> .	Getting QoS Values from a QoS Profile (Section 8.2.5 on page 547)
	get_<entity>_qos_from_profile_w_topic_name	Like get_<entity>_qos_from_profile(), but this operation allows you to specify a topic name associated with the entity. The topic filter expressions in the profile will be evaluated on the topic name. <entity> may be <i>topic</i> , <i>datareader</i> , or <i>datawriter</i> .	
	get_qos_profiles	Gets the names of all XML QoS profiles associated with a specified XML QoS profile library.	Configuring QoS with XML (Section 17.4 on page 803)
	get_qos_profile_libraries	Gets the names of all XML QoS profile libraries associated with the DomainParticipantFactory.	Retrieving a List of Available Libraries (Section 17.10.1 on page 823)
	load_profiles	Explicitly loads or reloads the QoS profiles.	Loading, Reloading and Unloading Profiles (Section 17.5.1 on page 811)
	reload_profiles		
	set_default_profile	Sets the default QoS profile for a DomainParticipantFactory.	Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library (Section 8.2.1.1 on the facing page)
	set_default_library	Sets the default library for a DomainParticipantFactory.	
	unload_profiles	Frees the resources associated with loading QoS profiles.	Loading, Reloading and Unloading Profiles (Section 17.5.1 on page 811)

8.2.1 Setting DomainParticipantFactory QoS Policies

The DDS_DomainParticipantFactoryQoS structure has the following format:

```

struct DDS_DomainParticipantFactoryQos {
    DDS_EntityFactoryQosPolicy          entity_factory;
    DDS_SystemResourceLimitsQosPolicy   resource_limits;
    DDS_ProfileQosPolicy                profile;
    DDS_LoggingQosPolicy                logging;
};

```

For information on *why* you would want to change a particular QoS Policy, see the section referenced in [Table 8.2 DomainParticipantFactory QoS](#).

Table 8.2 DomainParticipantFactory QoS

QoS Policy	Description
EntityFactory	Controls whether or not child entities are created in the enabled state. See ENTITYFACTORY QoS Policy (Section 6.4.2 on page 315) .
Logging	Configures the properties associated with Connex DDS logging. See LOGGING QoS Policy (DDS Extension) (Section 8.4.1 on page 572) .
Profile	Configures the way that XML documents containing QoS profiles are loaded by RTI. See PROFILE QoS Policy (DDS Extension) (Section 8.4.2 on page 573) .
SystemResourceLimits	Configures DomainParticipant-independent resources used by Connex DDS. Mainly used to change the maximum number of DomainParticipants that can be created within a single process (address space). See SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 8.4.3 on page 575) .

8.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library

You can retrieve the default QoS profile for the DomainParticipantFactory with the `get_default_profile()` operation. You can also get the default library for the DomainParticipantFactory, as well as the library that contains the DomainParticipantFactory's default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

```

virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()

```

There are also operations for setting the DomainParticipantFactory's default library and profile:

```

DDS_ReturnCode_t set_default_library (const char * library_name)
DDS_ReturnCode_t set_default_profile (const char * library_name,
                                     const char * profile_name)

```

set_default_profile() specifies the profile that will be used as the default the next time a default DomainParticipantFactory profile is needed during a call to a DomainParticipantFactory operation.

When calling a DomainParticipantFactory operation that requires a **profile_name** parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

set_default_profile() does not set the default QoS for the *DomainParticipant* that can be created by the DomainParticipantFactory. To set the default QoS using a profile, use the DomainParticipantFactory's **set_default_participant_qos_with_profile()** operation (see [Getting and Setting Default QoS for DomainParticipants \(Section 8.2.2 below\)](#)).

8.2.2 Getting and Setting Default QoS for DomainParticipants

To *get* the default QoS that will be used for creating *DomainParticipants* if **create_participant()** is called with DDS_PARTICIPANT_QOS_DEFAULT as the **qos** parameter, use this DomainParticipantFactory operation:

```
DDS_ReturnCode_t get_default_participant_qos (DDS_DomainParticipantQos & qos)
```

This operation gets the QoS settings that were specified on the last successful call to **set_default_participant_qos()** or **set_default_participant_qos_with_profile()**, or if the call was never made, the default values listed in DDS_DomainParticipantQos.

To *set* the default QoS that will be used for new *DomainParticipants*, use the following operations. Then these default QoS will be used if **create_participant()** is called with DDS_PARTICIPANT_QOS_DEFAULT as the 'qos' parameter.

```
DDS_ReturnCode_t set_default_participant_qos (  
    const DDS_DomainParticipantQos &qos)
```

or

```
DDS_ReturnCode_t set_default_participant_qos_with_profile (  
    const char *library_name, const char *profile_name)
```

Notes:

- These operations may potentially allocate memory, depending on the sequences contained in some QoS policies.
- It is not safe to set the default *DomainParticipant* QoS values while another thread may be simultaneously calling **get_default_participant_qos()**, **set_default_participant_qos()**, or **create_participant()** with DDS_PARTICIPANT_QOS_DEFAULT as the **qos** parameter. It is also not safe to

get the default *DomainParticipant* QoS values while another thread may be simultaneously calling `set_default_participant_qos()`.

8.2.3 Freeing Resources Used by the DomainParticipantFactory

The `finalize_instance()` operation explicitly reclaims resources used by the participant factory singleton (including resources use for QoS profiles).

On many operating systems, these resources are automatically reclaimed by the OS when the program terminates. However, some memory-check tools will flag those resources as unreclaimed. This method provides a way to clean up all the memory used by the participant factory.

Before calling `finalize_instance()` on a *DomainParticipantFactory*, all of the participants created by the factory must have been deleted. For a *DomainParticipant* to be successfully deleted, all *Entities* created by the participant or by the *Entities* that the participant created must have been deleted. In essence, the *DomainParticipantFactory* cannot be deleted until all other *Entities* have been deleted in an application.

Except for Linux systems: `get_instance()` and `finalize_instance()` are UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to get or finalize the factory instance. Subsequent calls are thread safe.

8.2.4 Looking Up DomainParticipants

The *DomainParticipantFactory* has these useful operations for retrieving its *DomainParticipants*:

- `get_participants()` returns a sequence of pointers to all the *DomainParticipants* within the *DomainParticipantFactory*.

```
DDS_ReturnCode_t
get_participants (DDSDomainParticipantSeq & participants)
```

- `lookup_participant()` locates an existing *DomainParticipant* based on its domain ID.

```
DDSDomainParticipant *
lookup_participant (DDS_DomainId_t domainId)
```

- `lookup_participant_by_name ()` locates an existing *DomainParticipant* based on its name.

```
DDSDomainParticipant *
lookup_participant_by_name(const char * participant_name)
```

Note: in the Modern C++ API these operations are freestanding functions `rti::domain::find_participants()`, `dds::domain::find()`, and `rti::domain::find_participant_by_name()` respectively.

8.2.5 Getting QoS Values from a QoS Profile

A QoS Profile may include configuration settings for all types of Entities. If you just want the settings for a specific type of Entity, call `get_<entity>_qos_from_profile()` (where `<entity>` may be **participant**, **publisher**, **subscriber**, **datawriter**, **datareader**, or **topic**). This is useful if you want to get the QoS values from the profile in a structure, make some changes, and then use that structure to create an entity.

```
DDS_ReturnCode_t get_<entity>_qos_from_profile (
    DDS_<Entity>Qos &qos,
    const char *library_name,
    const char *profile_name)
```

For an example, see [Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values](#) (Section Figure 6.5 on page 254).

The `get_<entity>_qos_from_profile()` operations do not take into account the **topic_filter** attributes that may be set for *DataWriter*, *DataReader*, or *Topic* QoSs in profiles (see [Topic Filters](#) (Section 17.3.4 on page 799)). If there is a topic name associated with an entity, you can call `get_<entity>_qos_from_profile_w_topic_name()` (where `<entity>` can be *datawriter*, *datareader*, or *topic*) and the topic filter expressions in the profile will be evaluated on the topic name.

```
DDS_ReturnCode_t get_<entity>_qos_from_profile_w_topic_name (
    DDS_<entity>Qos &qos,
    const char *library_name,
    const char *profile_name,
    const char *topic_name)
```

`get_<entity>_qos_from_profile()` and `get_<entity>_qos_from_profile_w_topic_name()` may allocate memory, depending on the sequences contained in some QoS policies.

Note: in the Modern C++ API, the class `QosProvider` provides the functionality described in this section. Please see the API Reference HTML documentation: [Modules](#), [RTI Connex DDS API Reference](#), [Configuring QoS Profiles with XML](#), `QosProvider`.

8.3 DomainParticipants

A *DomainParticipant* is a container for *Entity* objects that all belong to the same DDS domain. Each *DomainParticipant* has its own set of internal threads and internal data structures that maintain information about the *Entities* created by itself and other *DomainParticipants* in the same DDS domain. A *DomainParticipant* is used to create and destroy *Publishers*, *Subscribers* and *Topics*.

Once you have a *DomainParticipant*, you can use it to perform the operations listed in [Table 8.3 DomainParticipant Operations](#). For more details on all operations, see the API Reference HTML documentation. Some of the first operations you'll be interested in are `create_topic()`, `create_subscriber()`, and `create_publisher()`.

Note: Some operations cannot be used within a listener callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Builtin Subscriber	get_builtin_subscriber	Returns the builtin Subscriber.	Built-in DataReaders (Section 16.2 on page 773)

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Domain-Participants	add_peer	Adds an entry to the peer list.	Adding and Removing Peers List Entries (Section 8.5.2.3 on page 581)
	enable	Enables the <i>DomainParticipant</i> .	Enabling DDS Entities (Section 4.1.2 on page 154)
	equals	Compares two <i>DomainParticipant</i> 's QoS structures for equality.	Comparing QoS Values (Section 8.3.6.2 on page 565)
	get_discovered_participant_data	Provides the ParticipantBuiltinTopicData for a discovered <i>DomainParticipant</i> .	Learning about Discovered DomainParticipants (Section 8.3.11 on page 571)
	get_discovered_participants	Provides a list of <i>DomainParticipants</i> that have been discovered.	
	get_domain_id	Gets the domain ID of the <i>DomainParticipant</i> .	Choosing a Domain ID and Creating Multiple DDS Domains (Section 8.3.4 on page 559)
	get_listener	Gets the currently installed <i>DomainParticipantListener</i> .	Setting Up DomainParticipantListeners (Section 8.3.5 on page 560)
	get_qos	Gets the <i>DomainParticipant</i> QoS.	Setting DomainParticipant QoS Policies (Section 8.3.6 on page 562)
	ignore_participant	Rejects the connection to a remote <i>DomainParticipant</i> .	Restricting Communication—Ignoring Entities (Section 16.4 on page 784)
	remove_peer	Removes an entry from the peer list.	Adding and Removing Peers List Entries (Section 8.5.2.3 on page 581)
	set_listener	Replaces the <i>DomainParticipantListener</i> .	Setting Up DomainParticipantListeners (Section 8.3.5 on page 560)
	set_qos	Sets the <i>DomainParticipant</i> QoS.	Setting DomainParticipant QoS Policies (Section 8.3.6 on page 562)
	set_qos_with_profile	Sets the <i>DomainParticipant</i> QoS based on a QoS profile.	

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Content-Filtered-Topics	create_contentfilteredtopic	Creates a ContentFilteredTopic that can be used to process content-based subscriptions.	Creating ContentFilteredTopics (Section 5.4.3 on page 214)
	create_contentfilteredtopic_with_filter		
	delete_contentfilteredtopic	Deletes a ContentFilteredTopic.	Deleting ContentFilteredTopics (Section 5.4.4 on page 218)
	register_contentfilter	Registers a new content filter.	Registering a Custom Filter (Section 5.4.8.2 on page 234)
	unregister_contentfilter	Unregisters a new content filter.	Unregistering a Custom Filter (Section 5.4.8.3 on page 236)
	lookup_contentfilter	Gets a previously registered content filter.	Retrieving a ContentFilter (Section 5.4.8.4 on page 237)
DataReaders	create_datareader	Creates a <i>DataReader</i> with a given <i>DataReaderListener</i> , and an implicit <i>Subscriber</i> .	Creating DataReaders (Section 7.3.1 on page 463)
	create_datareader_with_profile	Creates a <i>DataReader</i> based on a QoS profile, with a given <i>DataReaderListener</i> , and an implicit <i>Subscriber</i> .	
	delete_datareader	Deletes a <i>DataReader</i> that belongs to the ‘implicit <i>Subscriber</i> .’	Deleting DataReaders (Section 7.3.3 on page 466)
	get_default_datareader_qos	Copies the default <i>DataReaderQoS</i> values into the provided structure.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	ignore_subscription	Rejects the connection to a <i>DataReader</i>	
	set_default_datareader_qos	Sets the default <i>DataReaderQoS</i> values.	
	set_default_datareader_qos_with_profile	Sets the default <i>DataReaderQoS</i> using values from a QoS profile.	

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
DataWriters	create_datawriter	Creates a <i>DataWriter</i> with a given <i>DataWriterListener</i> , and an implicit <i>Publisher</i> .	Creating Publishers (Section 6.2.2 on page 249)
	create_datawriter_with_profile	Creates a <i>DataWriter</i> based on a QoS profile, with a given <i>DataWriterListener</i> , and an implicit <i>Publisher</i> .	
	delete_datawriter	Deletes a <i>DataWriter</i> that belongs to the ‘implicit <i>Publisher</i> .’	Deleting Publishers (Section 6.2.3 on page 250)
	ignore_publication	Rejects the connection to a <i>DataWriter</i> .	Restricting Communication—Ignoring Entities (Section 16.4 on page 784)
	get_default_datawriter_qos	Copies the default <i>DataWriterQos</i> values into the provided <i>DataWriterQos</i> structure.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	set_default_datawriter_qos	Sets the default <i>DataWriterQoS</i> values.	
	set_default_datawriter_qos_with_profile	Sets the default <i>DataWriterQos</i> using values from a profile.	

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Publishers	create_publisher	Creates a <i>Publisher</i> and a <i>PublisherListener</i> .	Creating Publishers (Section 6.2.2 on page 249)
	create_publisher_with_profile	Creates a <i>Publisher</i> based on a QoS profile, and a <i>PublisherListener</i> .	
	delete_publisher	Deletes a <i>Publisher</i> .	Deleting Publishers (Section 6.2.3 on page 250)
	get_default_publisher_qos	Copies the default <i>PublisherQos</i> values into the provided <i>PublisherQos</i> structure.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	get_implicit_publisher	Gets the <i>Publisher</i> that is implicitly created by the <i>DomainParticipant</i> .	Getting the Implicit Publisher or Subscriber (Section 8.3.9 on page 569)
	get_publishers	Provides a list of all <i>Publishers</i> owned by the <i>DomainParticipant</i> .	Getting All Publishers and Subscribers (Section 8.3.13.3 on page 572)
	set_default_publisher_qos	Sets the default <i>PublisherQos</i> values.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	set_default_publisher_qos_with_profile	Sets the default <i>PublisherQos</i> using values from a QoS profile.	

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Subscribers	create_subscriber	Creates a <i>Subscriber</i> and a <i>SubscriberListener</i> .	Creating Subscribers (Section 7.2.2 on page 445)
	create_subscriber_with_profile	Creates a <i>Subscriber</i> based on a QoS profile, and a <i>SubscriberListener</i> .	
	delete_subscriber	Deletes a <i>Subscriber</i> .	Deleting Subscribers (Section 7.2.3 on page 446)
	get_default_subscriber_qos	Copies the default <i>SubscriberQos</i> values into the provided <i>SubscriberQos</i> structure.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	get_implicit_subscriber	Gets the <i>Subscriber</i> that is implicitly created by the <i>DomainParticipant</i> .	Getting the Implicit Publisher or Subscriber (Section 8.3.9 on page 569)
	get_subscribers	Provides a list of all <i>Subscribers</i> owned by the <i>DomainParticipant</i> .	Getting All Publishers and Subscribers (Section 8.3.13.3 on page 572)
	set_default_subscriber_qos	Sets the default <i>SubscriberQos</i> values.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	set_default_subscriber_qos_with_profile	Sets the default <i>SubscriberQos</i> values using values from a QoS profile.	
Durable Subscriptions	delete_durable_subscription	Deletes an existing Durable Subscription. The quorum of the existing DDS samples will be considered satisfied.	Configuring Durable Subscriptions in Persistence Service (Section 27.9 on page 955)
	register_durable_subscription	<p>Creates a Durable Subscription that will receive all DDS samples published on a <i>Topic</i>, including those published while a <i>DataReader</i> is inactive or before it may be created.</p> <p><i>RTI Persistence Service</i> will ensure that all the DDS samples on that <i>Topic</i> are retained until they are acknowledged by at least <i>N</i> <i>DataReaders</i> belonging to the Durable Subscription, where <i>N</i> is the quorum count.</p> <p>If the same Durable Subscription is created on a different <i>Topic</i>, <i>RTI Persistence Service</i> will implicitly delete the previous Durable Subscription and create a new one on the new <i>Topic</i>.</p>	

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Topics	create_topic	Creates a <i>Topic</i> and a TopicListener.	Creating Topics (Section 5.1.1 on page 202)
	create_topic_with_profile	Creates a Topic based on a QoS profile, and a TopicListener.	
	delete_topic	Deletes a <i>Topic</i> .	
	get_default_topic_qos	Copies the default TopicQos values into the provided TopicQos structure.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	get_discovered_topic_data	Retrieves the BuiltinTopicData for a discovered <i>Topic</i> .	Learning about Discovered Topics (Section 8.3.12 on page 571)
	get_discovered_topics	Returns a list of all (non-ignored) discovered <i>Topics</i> .	
	ignore_topic	Rejects a remote topic.	Restricting Communication—Ignoring Entities (Section 16.4 on page 784)
	lookup_topicdescription	Gets an existing locally-created TopicDescription (Topic).	Looking up Topic Descriptions (Section 8.3.7 on page 568)
	set_default_topic_qos	Sets the default TopicQos values.	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	set_default_topic_qos_with_profile	Sets the default TopicQos values using values from a profile.	
	find_topic	Finds an existing Topic, based on its name.	Finding a Topic (Section 8.3.8 on page 569)

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Flow-Controllers	create_flowcontroller	Creates a custom FlowController object.	Creating and Deleting FlowControllers (Section 6.6.6 on page 433)
	delete_flowcontroller	Deletes a custom FlowController object.	
	get_default_flowcontroller_property	Gets the default properties used when a new FlowController is created.	Getting/Setting Default FlowController Properties (Section 6.6.7 on page 434)
	set_default_flowcontroller_property	Sets the default properties used when a new FlowController is created.	
	lookup_flowcontroller	Finds a FlowController, based on its name.	Other FlowController Operations (Section 6.6.10 on page 435)
Libraries and Profiles	get_default_library	Gets the default library.	Getting and Setting DomainParticipant's Default QoS Profile and Library (Section 8.3.6.4 on page 567)
	get_default_profile	Gets the default profile.	
	get_default_profile_library	Gets the library that contains the default profile.	
	set_default_profile	Sets the default QoS profile.	
	set_default_library	Sets the default library.	
MultiTopics	create_multitopic	Creates a <i>MultiTopic</i> that can be used to subscribe to multiple topics and combine/filter the received data into a resulting type.	Currently not supported.
	delete_multitopic	Deletes a <i>MultiTopic</i> .	

Table 8.3 DomainParticipant Operations

Working with ...	Operation	Description	Reference
Other	assert_liveliness	Manually asserts the liveliness of this <i>DomainParticipant</i> .	Getting the Implicit Publisher or Subscriber (Section 8.3.9 on page 569)
	delete_contained_entities	Recursively deletes all the entities that were created using the "create" operations on the <i>DomainParticipant</i> and its children.	Deleting Contained Entities (Section 8.3.3 on page 559)
	contains_entity	Confirms if an entity belongs to the <i>DomainParticipant</i> or not.	Verifying Entity Containment (Section 8.3.13.1 on page 571)
	get_current_time	Gets the current time used by Connexx DDS.	Getting the Current Time (Section 8.3.13.2 on page 571)
	get_status_changes	Gets a list of statuses that have changed since the last time the application read the status or the <i>Listeners</i> were called.	Getting Status and Status Changes (Section 4.1.4 on page 157)

8.3.1 Creating a DomainParticipant

Typically, you will only need to create one *DomainParticipant* per DDS domain per application. (Although unusual, you can create multiple *DomainParticipants* for the same DDS domain in an application.)

To create a *DomainParticipant*, use the *DomainParticipantFactory*'s `create_participant()` or `create_participant_with_profile()` operation:

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Note: In the Modern C++ API, you will use the *DomainParticipant* constructors.

```

DDSDomainParticipant * create_participant (
    DDS_DomainId_t domainId,
    const DDS_DomainParticipantQos &qos,
    DDSDomainParticipantListener *listener,
    DDS_StatusMask mask)

DDSDomainParticipant * create_participant_with_profile (
    DDS_DomainId_t domainId,
    const char * library_name,
    const char *profile_name,
    DDSDomainParticipantListener *listener,
    DDS_StatusMask mask)

```

Where:

- domainId** The domain ID uniquely identifies the DDS domain that the *DomainParticipant* is in. It controls with which other *DomainParticipants* it will communicate. See [Choosing a Domain ID and Creating Multiple DDS Domains \(Section 8.3.4 on page 559\)](#) for more information on domain IDs.
- qos** If you want the default QoS settings (described in the API Reference HTML documentation), use `DDS_PARTICIPANT_QOS_DEFAULT` for this parameter (see [Creating a DomainParticipant with Default QoS Policies \(Section Figure 8.4 on the facing page\)](#)). If you want to customize any of the QoS Policies, supply a `DomainParticipantQos` structure that is described in [Setting DomainParticipant QoS Policies \(Section 8.3.6 on page 562\)](#).
- Note:** If you use `DDS_PARTICIPANT_QOS_DEFAULT`, it is not safe to create the *DomainParticipant* while another thread may simultaneously be calling the `DomainParticipantFactory`'s `set_default_participant_qos()` operation.
- listener** Listeners are callback routines. Connex DDS uses them to notify your application of specific events (status changes) that may occur. The listener parameter may be set to `NULL` if you do not want to install a *Listener*. The *DomainParticipant's Listener* is a catchall for all of the events of all of its *Entities*. If an event is not handled by an *Entity's Listener*, then the *DomainParticipantListener* may be called in response to the event. For more information, see [Setting Up DomainParticipantListeners \(Section 8.3.5 on page 560\)](#).
- mask** This bit mask indicates which status changes will cause the *Listener* to be invoked. The bits set in the mask must have corresponding callbacks implemented in the *Listener*. If you use `NULL` for the *Listener*, use `DDS_STATUS_MASK_NONE` for this parameter. If the *Listener* implements all callbacks, use `DDS_STATUS_MASK_ALL`. For information on statuses, see [Listeners \(Section 4.4 on page 177\)](#).
- library_name** A QoS Library is a named set of QoS profiles. See [URL Groups \(Section 17.8 on page 814\)](#).
- profile_name** A QoS profile groups a set of related QoS, usually one per entity. See [URL Groups \(Section 17.8 on page 814\)](#).

After you create a *DomainParticipant*, the next step is to register the data types that will be used by the application, see [Using RTI Code Generator \(rtiddsgen\) \(Section 3.6 on page 138\)](#). Then you will need to create the *Topics* that the application will publish and/or subscribe, see [Creating Topics \(Section 5.1.1 on page 202\)](#). Finally, you will use the *DomainParticipant* to create *Publishers* and/or *Subscribers*, see [Creating Publishers \(Section 6.2.2 on page 249\)](#) and [Creating Subscribers \(Section 7.2.2 on page 445\)](#).

Note: It is not safe to create one *DomainParticipant* while another thread may simultaneously be looking up ([Looking Up DomainParticipants \(Section 8.2.4 on page 546\)](#)) or deleting ([Deleting DomainParticipants \(Section 8.3.2 on the facing page\)](#)) the same *DomainParticipant*.

For more examples, see [Configuring QoS Settings when DomainParticipant is Created \(Section 8.3.6.1 on page 564\)](#).

Figure 8.4 Creating a DomainParticipant with Default QosPolicies

```

DDS_DomainId_t domain_id = 10;
// MyDomainParticipantListener is user defined and
// extends DDSDomainParticipantListener
MyDomainParticipantListener* participant_listener =
    new MyDomainParticipantListener(); // or = NULL
// Create the participant
DDSDomainParticipant* participant = factory->create_participant(
    domain_id, DDS_PARTICIPANT_QOS_DEFAULT,
    participant_listener, DDS_STATUS_MASK_ALL);
if (participant == NULL) {
    // ... error
};

```

8.3.2 Deleting DomainParticipants

If the application is no longer interested in communicating in a certain DDS domain, the *DomainParticipant* can be deleted. A *DomainParticipant* can be deleted only after all the *Entities* that were created by the *DomainParticipant* have been deleted (see [Deleting Contained Entities \(Section 8.3.3 on the next page\)](#)).

To delete a *DomainParticipant*:

You must first delete all *Entities* (*Publishers*, *Subscribers*, *ContentFilteredTopics*, and *Topics*) that were created with the *DomainParticipant*. Use the *DomainParticipant*'s `delete_<entity>()` operations to delete them one at a time, or use the `delete_contained_entities()` operation ([Deleting Contained Entities \(Section 8.3.3 on the next page\)](#)) to delete them all at the same time.

```

DDS_ReturnCode_t delete_publisher (DDSPublisher *p)
DDS_ReturnCode_t delete_subscriber (DDSSubscriber *s)
DDS_ReturnCode_t delete_contentfilteredtopic
    (DDSContentFilteredTopic *a_contentfilteredtopic)
DDS_ReturnCode_t delete_topic (DDSTopic *topic)

```

Delete the *DomainParticipant* by using the *DomainParticipantFactory*'s `delete_participant()` operation.

```

DDS_ReturnCode_t delete_participant
    (DDSDomainParticipant *a_participant)

```

Note: A *DomainParticipant* cannot be deleted within its *Listener* callback, see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

After a *DomainParticipant* has been deleted, all of the participant's internal Connex DDS threads and allocated memory will have been deleted. You should delete the *DomainParticipantListener* only after the *DomainParticipant* itself has been deleted.

Note: In the Modern C++ API, *Entities* are automatically destroyed.

8.3.3 Deleting Contained Entities

The *DomainParticipant*'s `delete_contained_entities()` operation deletes all the *Publishers* (including an implicitly created one, if it exists), *Subscribers* (including an implicitly created one, if it exists), *ContentFilteredTopics*, *MultiTopics*, and *Topics* that have been created by the *DomainParticipant*.

```
DDS_ReturnCode_t delete_contained_entities( )
```

Prior to deleting each contained entity, this operation recursively calls the corresponding `delete_contained_entities()` operation on each contained entity (if applicable). This pattern is applied recursively. Therefore, `delete_contained_entities()` on the *DomainParticipant* will end up deleting all the entities recursively contained in the *DomainParticipant*, that is also the *DataWriter*, *DataReader*, as well as the *QueryCondition* and *ReadCondition* objects belonging to the contained *DataReader*.

If `delete_contained_entities()` returns successfully, the application may delete the **DomainParticipant** knowing that it has no contained entities (see [Deleting DomainParticipants \(Section 8.3.2 on the previous page\)](#)).

8.3.4 Choosing a Domain ID and Creating Multiple DDS Domains

A domain ID identifies the DDS domain in which the *DomainParticipant* is communicating. *DomainParticipants* with the same domain ID are on the same communication “channel”. *DomainParticipants* with different domain IDs are completely isolated from each other.

The domain ID is a purely arbitrary value; you can use any integer 0 or higher, provided it does not violate the guidelines for the `DDS_RtpsWellKnownPorts_t` structure ([Ports Used for Discovery \(Section 8.5.9.3 on page 613\)](#)). Domain IDs are typically between 0 and 232. Please see the API Reference HTML documentation for the `DDS_RtpsWellKnownPorts_t` structure and in particular, `DDS_INTEROPERABLE_RTSPS_WELL_KNOWN_PORTS`.

Most distributed systems can use a single DDS domain for all of its applications. Thus a single domain ID is sufficient. Some systems may need to logically partition nodes to prevent them from communicating with each other directly, and thus will need to use multiple DDS domains. However, even in systems that only use a single DDS domain, during the testing and development phases, one may want to assign different users/testers different domain IDs for running their applications so that their tests do not interfere with each other.

To run multiple applications on the same node with the same domain ID, Connex DDS uses a participant ID to distinguish between the different *DomainParticipants* in the different applications. The participant ID is simply an integer value that must be unique across all *DomainParticipants* created on the same node that use the same domain ID. The `participant_id` is part of the [WIRE_PROTOCOL QosPolicy \(DDS Extension\) \(Section 8.5.9 on page 610\)](#).

Although usually those *DomainParticipants* have been created in different applications, the same application can also create multiple *DomainParticipants* with the same domain ID. For optimal results, the par-

participant_id should be assigned sequentially to the different *DomainParticipants*, starting from the default value of 0.

Once you have a *DomainParticipant*, you can retrieve its domain ID with the `get_domain_id()` operation.

The domain ID and participant ID are mapped to port numbers that are used by transports for discovery traffic. For information on how port numbers are calculated, see [Ports Used for Discovery \(Section 14.5 on page 738\)](#). How *DomainParticipants* discover each other is discussed in [Discovery \(Section Chapter 14 on page 709\)](#).

8.3.5 Setting Up DomainParticipantListeners

DomainParticipants may optionally have *Listeners*. *Listeners* are essentially callback routines and are how Connex DDS will notify your application of specific events (changes in status) for entities *Topics*, *Publishers*, *Subscribers*, *DataWriters*, and *DataReaders*. Each *Entity* may have a *Listener* installed and enabled to process the events for itself and all of the sub-*Entities* created from it. If an *Entity* does not have a *Listener* installed or is not enabled to listen for a particular event, then Connex DDS will propagate the event to the *Entity*'s parent. If the parent *Entity* does not process the event, Connex DDS will continue to propagate the event up the object hierarchy until either a *Listener* is invoked or the event is dropped.

The *DomainParticipantListener* is the last chance that an event can be processed for the *Entities* descended from a *DomainParticipant*. The *DomainParticipantListener* is used only if an event is not handled by any of the *Entities* contained by the participant.

A *Listener* is typically set up when the *DomainParticipant* is created (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#)). You can also set one up after creation time by using the `set_listener()` operation, as illustrated in [Setting up DomainParticipantListener \(Section Figure 8.5 below\)](#). The `get_listener()` operation can be used to retrieve the current *DomainParticipantListener*.

Figure 8.5 Setting up DomainParticipantListener

```
// MyDomainParticipantListener only handles PUBLICATION_MATCHED and
// SUBSCRIPTION_MATCHED status for DomainParticipant Entities
class MyDomainParticipantListener :
    public DDSDomainParticipantListener {
public:
    virtual void on_publication_matched(DDSDataWriter *writer,
        const DDS_PublicationMatchedStatus &status);
    virtual void on_subscription_matched(DDSDataReader *reader,
        const DDS_SubscriptionMatchedStatus &status);
};
void MyDomainParticipantListener::on_publication_matched(
    DDSDataWriter *writer,
    const DDS_PublicationMatchedStatus &status)
{
    const char *name = writer->get_topic()->get_name();
    printf("Number of matching DataReaders for Topic %s is %d\n",
        name, status.current_count);
}
```

```

};
void MyDomainParticipantListener::on_subscription_matched(
    DDSDataReader *reader,
    const DDS_SubscriptionMatchedStatus &status)
{
    const char *name =
        reader->get_topicdescription()->get_name();
    printf("Number of matching DataWriters for Topic %s is %d\n",
        name, status.current_count);
};

// Set up participant listener
MyDomainParticipantListener* participant_listener =
    new MyDomainParticipantListener();
if (participant_listener == NULL) {
    // ... handle error
}
// Create the participant with a listener
DDSDomainParticipant* participant = factory->create_participant(
    domain_id, participant_qos, participant_listener,
    DDS_PUBLICATION_MATCHED_STATUS |
    DDS_SUBSCRIPTION_MATCHED_STATUS );
if (participant == NULL) {
    // ... handle error
}

```

If a *Listener* is set for a *DomainParticipant*, the *Listener* needs to exist as long as the *DomainParticipant* exists. It is unsafe to destroy the *Listener* while it is attached to a participant. However, you may remove the *DomainParticipantListener* from a *DomainParticipant* by calling **set_listener()** with a NULL value. Once the *Listener* has been removed from the participant, you may safely destroy it (see [Types of Listeners \(Section 4.4.1 on page 177\)](#)).

Notes:

- Due to a thread-safety issue, the destruction of a *DomainParticipantListener* from an enabled *DomainParticipant* should be avoided—even if the *DomainParticipantListener* has been removed from the *DomainParticipant*. (This limitation does not affect the Java API.)
- It is possible for multiple internal Connext DDS threads to call the same method of a *DomainParticipantListener* simultaneously. You must write the methods of a *DomainParticipantListener* to be multithread safe and reentrant. The methods of the *Listener* of other Entities do not have this constraint and are guaranteed to have single threaded access.

See also:

- [Setting Up TopicListeners \(Section 5.1.5 on page 208\)](#)
- [Setting Up PublisherListeners \(Section 6.2.5 on page 257\)](#)

- [Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#)
- [Setting Up SubscriberListeners \(Section 7.2.6 on page 454\)](#)
- [Setting Up DataReaderListeners \(Section 7.3.4 on page 466\)](#)

8.3.6 Setting DomainParticipant QosPolicies

A *DomainParticipant's* QosPolicies are used to configure discovery, database sizing, threads, information sent to other *DomainParticipants*, and the behavior of the *DomainParticipant* when acting as a factory for other *Entities*.

Note: `set_qos()` cannot always be used in a listener callback; see [Restricted Operations in Listener Callbacks \(Section 4.5.1 on page 185\)](#).

The `DDS_DomainParticipantQos` structure has the following format:

```
struct DDS_DomainParticipantQos {
    DDS_UserDataQosPolicy          user_data;
    DDS_EntityFactoryQosPolicy     entity_factory;
    DDS_WireProtocolQosPolicy      wire_protocol;
    DDS_TransportBuiltinQosPolicy  transport_builtin;
    DDS_TransportUnicastQosPolicy  default_unicast;
    DDS_DiscoveryQosPolicy         discovery;
    DDS_DomainParticipantResourceLimitsQosPolicy resource_limits;
    DDS_EventQosPolicy            event;
    DDS_ReceiverPoolQosPolicy     receiver_pool;
    DDS_DatabaseQosPolicy         database;
    DDS_DiscoveryConfigQosPolicy  discovery_config;
    DDS_PropertyQosPolicy         property;
    DDS_EntityNameQosPolicy       participant_name;
    DDS_TransportMulticastMappingQosPolicy multicast_mapping;
    DDS_TypeSupportQosPolicy      type_support;
};
```

[Table 8.4 DomainParticipant QosPolicies](#) summarizes the meaning of each policy (listed alphabetically). For information on *why* you would want to change a particular QosPolicy, see the section referenced in the table.

Table 8.4 DomainParticipant QosPolicies

QosPolicy	Description
Database	Various settings and resource limits used by Connexx DDS to control its internal database. See DATABASE QosPolicy (DDS Extension) (Section 8.5.1 on page 577) .
Discovery	Configures the mechanism used by Connexx DDS to automatically discover and connect with new remote applications. See DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2 on page 580) .

Table 8.4 DomainParticipant QoS Policies

QoS Policy	Description
DiscoveryConfig	Controls the amount of delay in discovering entities in the system and the amount of discovery traffic in the network. See DISCOVERY_CONFIG QoS Policy (DDS Extension) (Section 8.5.3 on page 585) .
DomainParticipantResourceLimits	Various settings that configure how <i>DomainParticipants</i> allocate and use physical memory for internal resources, including the maximum sizes of various properties. See DOMAIN_PARTICIPANT_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 8.5.4 on page 593) .
EntityFactory	Controls whether or not child entities are created in the enabled state. See ENTITYFACTORY QoS Policy (Section 6.4.2 on page 315) .
EntityName	Assigns a name to a <i>DomainParticipant</i> . See ENTITY_NAME QoS Policy (DDS Extension) (Section 6.5.9 on page 374) .
Event	Configures the <i>DomainParticipant's</i> internal thread that handles timed events. See EVENT QoS Policy (DDS Extension) (Section 8.5.5 on page 602) .
Property	Stores name/value(string) pairs that can be used to configure certain parameters of Connex DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See PROPERTY QoS Policy (DDS Extension) (Section 6.5.17 on page 394) .
ReceiverPool	Configures threads used by Connex DDS to receive and process data from transports (for example, UDP sockets). See RECEIVER_POOL QoS Policy (DDS Extension) (Section 8.5.6 on page 604) .
TransportBuiltin	Specifies which built-in transport plugins are used. See TRANSPORT_BUILTIN QoS Policy (DDS Extension) (Section 8.5.7 on page 606) .
TransportMulticastMapping	Specifies the automatic mapping between a list of topic expressions and multicast address that can be used by a <i>DataReader</i> to receive data for a specific topic. See TRANSPORT_MULTICAST_MAPPING QoS Policy (DDS Extension) (Section 8.5.8 on page 608) .
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See TRANSPORT_UNICAST QoS Policy (DDS Extension) (Section 6.5.24 on page 412) .
TypeSupport	Used to attach application-specific value(s) to a <i>DataWriter</i> or <i>DataReader</i> . These values are passed to the serialization or deserialization routine of the associated data type. See TYPESUPPORT QoS Policy (DDS Extension) (Section 6.5.25 on page 416) .
UserData	Along with Topic Data QoS Policy and Group Data QoS Policy, used to attach a buffer of bytes to Connex DDS's discovery meta-data. See USER_DATA QoS Policy (Section 6.5.26 on page 417) .
WireProtocol	Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See WIRE_PROTOCOL QoS Policy (DDS Extension) (Section 8.5.9 on page 610) .

8.3.6.1 Configuring QoS Settings when DomainParticipant is Created

As described in [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#), there are different ways to create a DomainParticipant, depending on how you want to specify its QoS (with or without a QoS Profile).

- [Figure 8.4 Creating a DomainParticipant with Default QoS Policies on page 558](#) has an example of how to create a DomainParticipant with default QoS Policies by using the special constant, `DDS_PARTICIPANT_QOS_DEFAULT`, which indicates that the default QoS values for a DomainParticipant should be used. The default DomainParticipant QoS values are configured in the `DomainParticipantFactory`; you can change them with `set_default_participant_qos()` or `set_default_participant_qos_with_profile()` (see [Getting and Setting Default QoS for DomainParticipants \(Section 8.2.2 on page 545\)](#)). Then any DomainParticipants created with the `DomainParticipantFactory` will use the new default values. As described in [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#), this is a general pattern that applies to the construction of all Entities.
- To create a DomainParticipant with non-default QoS without using a QoS Profile, see the example code in [Figure 8.6 Creating DomainParticipant with Modified QoS Policies \(not from profile\) below](#). It uses the `DomainParticipantFactory`'s `get_default_participant_qos()` method to initialize a `DDS_ParticipantQos` structure. Then, the policies are modified from their default values before the structure is used in the `create_participant()` method.
- You can also create a DomainParticipant and specify its QoS settings via a QoS Profile. To do so, you will call `create_participant_with_profile()`, as seen in [Figure 8.7 Creating DomainParticipant with QoS Profile on the next page](#).
- If you want to use a QoS profile, but then make some changes to the QoS before creating the DomainParticipant, call `get_participant_qos_from_profile()` and `create_participant()` as seen in [Figure 8.8 Getting QoS from Profile, Creating DomainParticipant with Modified QoS Values on the next page](#).

For more information, see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#) and [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 8.6 Creating DomainParticipant with Modified QoS Policies (not from profile)

```
DDS_DomainId_t domain_id = 10;
DDS_DomainParticipantQos participant_qos;1
// initialize participant_qos with default values
factory->get_default_participant_qos(participant_qos);
// make QoS changes here
participant_qos.wire_protocol.participant_id = 2;
```

¹In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

```
// Create the participant with modified qos
DDSDomainParticipant* participant = factory->create_participant(
    domain_id, participant_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // ... error
}
```

Figure 8.7 Creating DomainParticipant with QoS Profile

```
DDS_DomainId_t domain_id = 10;
// MyDomainParticipantListener is user defined and
// extends DDSDomainParticipantListener
MyDomainParticipantListener* participant_listener
    = new MyDomainParticipantListener(); // or = NULL
// Create the participant
DDSDomainParticipant* participant =
    factory->create_participant_with_profile(domain_id,
        "MyDomainLibrary", "MyDomainProfile",
        participant_listener, DDS_STATUS_MASK_ALL);
if (participant == NULL) {
    // ... error
};
```

Figure 8.8 Getting QoS from Profile, Creating DomainParticipant with Modified QoS Values

```
DDS_DomainParticipantQos participant_qos;1
// Get DomainParticipant QoS from profile
retcode = factory->get_participant_qos_from_profile( participant_qos,
    "DomainParticipantProfileLibrary", "DomainParticipantProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
participant_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
// create participant with modified QoS
DDSDomainParticipant* participant = factory->create_participant(domain_id,
    participant_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // handle error
}
```

8.3.6.2 Comparing QoS Values

The `equals()` operation compares two *DomainParticipant*'s `DDS_DomainParticipantQoS` structures for equality. It takes two parameters for the two *DomainParticipant*'s QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

¹In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

8.3.6.3 Changing QoS Settings After DomainParticipant Has Been Created

There are two ways to change an existing DomainParticipant's QoS after it has been created—again depending on whether or not you are using a QoS Profile.

- To change QoS programmatically (that is, without using a QoS Profile), use `get_qos()` and `set_qos()`. See the example code in [Figure 8.9 Changing QoS of Existing Participant \(without QoS Profile\) below](#). It retrieves the current values by calling the DomainParticipant's `get_qos()` operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QoS Policies cannot be changed after the DomainParticipant has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.
- You can also change a *DomainParticipant's* (and all other Entities') QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see [Figure 8.10 Changing QoS of Existing Participant with QoS Profile below](#). For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

Figure 8.9 Changing QoS of Existing Participant (without QoS Profile)

```
DDS_DomainParticipantQos participant_qos;1
// Get current QoS
//participant points to an existing DDSDomainParticipant
if (participant->get_qos(participant_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Make QoS changes
participant_qos.entity_factory.autoenable_created_entities =
    DDS_BOOLEAN_FALSE;
// Set the new QoS
if (participant->set_qos(participant_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 8.10 Changing QoS of Existing Participant with QoS Profile

```
DDS_DomainParticipantQos participant_qos;2
// Get current QoS
//participant points to an existing DDSDomainParticipant
if (participant->get_qos(participant_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

¹In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

²In C, you must initialize the QoS structures before they are used, see [Special QoS Policy Handling Considerations for C \(Section 4.2.2 on page 168\)](#).

```
// Make QoS changes
participant_qos.entity_factory.autoenable_created_entities =
    DDS_BOOLEAN_FALSE;
// Set the new QoS
if (participant->set_qos(participant_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

8.3.6.4 Getting and Setting DomainParticipant's Default QoS Profile and Library

You can get the default QoS profile for the *DomainParticipant* with the **get_default_profile()** operation. You can also get the default library for the *DomainParticipant*, as well as the library that contains the *DomainParticipant's* default profile (these are not necessarily the same library); these operations are called **get_default_library()** and **get_default_library_profile()**, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

```
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()
```

There are also operations for *setting* the *DomainParticipant's* default library and profile:

```
DDS_ReturnCode_t set_default_library (
    const char * library_name)
DDS_ReturnCode_t set_default_profile (
    const char * library_name,
    const char * profile_name)
```

If the default profile/library is not set, the *DomainParticipant* inherits the default from the *DomainParticipantFactory*.

- **set_default_profile()** specifies the profile that will be used as the default the next time a default *DomainParticipant* profile is needed during a call to one of this *DomainParticipant's* operations. When calling a *DomainParticipant* operation that requires a **profile_name** parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)
- **set_default_profile()** does not set the default QoS for entities created by the *DomainParticipant*; for this functionality, use the *DomainParticipant's* **set_default_<entity>_qos_with_profile()** operation (you may pass in NULL after having called **set_default_profile()**, see [Getting and Setting Default QoS for Child Entities \(Section 8.3.6.5 on the facing page\)](#)).
- **set_default_profile()** does not set the default QoS for newly created *DomainParticipants*; for this functionality, use the *DomainParticipantFactory's* **set_default_participant_qos_with_profile()**, see [Getting and Setting Default QoS for DomainParticipants \(Section 8.2.2 on page 545\)](#)).

8.3.6.5 Getting and Setting Default QoS for Child Entities

The `set_default_<entity>_qos()` and `set_default_<entity>_qos_with_profile()` operations set the default QoS that will be used for newly created entities (where `<entity>` may be **publisher**, **subscriber**, **datawriter**, **datareader**, or **topic**). The new QoS settings will only be used if `DDS_<entity>_QOS_DEFAULT` is specified as the `qos` parameter when `create_<entity>()` is called. For example, for a *Publisher*, you can use either:

```
DDS_ReturnCode_t set_default_publisher_qos (
    const DDS_PublisherQos &qos)
DDS_ReturnCode_t set_default_publisher_qos_with_profile (
    const char *library_name,
    const char *profile_name)
```

The following operation gets the default QoS that will be used for creating *Publishers* if `DDS_PUBLISHER_QOS_DEFAULT` is specified as the ‘qos’ parameter when `create_publisher()` is called:

```
DDS_ReturnCode_t get_default_publisher_qos (
    DDS_PublisherQos & qos)
```

There are similar operations for *Subscribers*, *DataWriters*, *DataReaders* and *Topics*. These operations, `get_default_<entity>_qos()`, get the QoS settings that were specified on the last successful call to `set_default_<entity>_qos()` or `set_default_<entity>_qos_with_profile()`, or if the call was never made, the default values listed in `DDS_<entity>Qos`. They may potentially allocate memory depending on the sequences contained in some QoS policies.

Note: It is not safe to set default QoS values for an entity while another thread may be simultaneously getting or setting them, or using the `QOS_DEFAULT` constant to create the entity.

8.3.7 Looking up Topic Descriptions

The `lookup_topicdescription()` operation allows you to access a locally created `DDSTopicDescription` based on the *Topic*’s name.

```
DDSTopicDescription* lookup_topicdescription(const char *topic_name)
```

DDSTopicDescription is the base class for *Topics*, *MultiTopics*¹ and *ContentFilteredTopics*. You can narrow the `DDSTopicDescription` returned from `lookup_topicdescription()` to a *Topic* or *ContentFilteredTopic* as appropriate.

Unlike `find_topic()` (see [Finding a Topic \(Section 8.3.8 on the next page\)](#)), which logically returns a new *Topic* that must be independently deleted, *this* operation returns a reference to the original local object.

¹*Multitopics* are not supported.

If no *TopicDescription* has been created yet with the given *Topic* name, this method will return a NULL value.

The *DomainParticipant* does not have to be enabled when you call **lookup_topicdescription()**.

Note: It is not safe to create or delete a topic while another thread is calling **lookup_topicdescription()** for that same topic.

8.3.8 Finding a Topic

The **find_topic()** operation finds an existing (or ready to exist) *Topic*, based on its name. This call can be used to block for a specified duration to wait for the *Topic* to be created.

```
DDSTopic* DDSDomainParticipant::find_topic (const char * topic_name,
                                           const DDS_Duration_t & timeout)
```

If the requested *Topic* already exists, it is returned. Otherwise, **find_topic()** waits until either another thread creates it, or returns when the specified timeout occurs.

find_topic() is useful when multiple threads are concurrently creating and looking up topics. In that case, one thread can call **find_topic()** and, if another thread has not yet created the topic being looked up, it can wait for some period of time for it to do so. In almost all other cases, it is more straightforward to call **lookup_topicdescription()** (see [Looking up Topic Descriptions \(Section 8.3.7 on the previous page\)](#)).

The *DomainParticipant* must be enabled when you call **find_topic()**.

Note: Each **DDSTopic** obtained by **find_topic()** must also be deleted by calling the *DomainParticipant*'s **delete_topic()** operation (see [Deleting Topics \(Section 5.1.2 on page 204\)](#)).

8.3.9 Getting the Implicit Publisher or Subscriber

The **get_implicit_publisher()** operation allows you to access the *DomainParticipant*'s implicit *Publisher*. If one does not already exist, this operation creates an implicit *Publisher*.

There is a similar operation for implicit *Subscribers*:

```
DDSPublisher * get_implicit_publisher ()

DDSSubscriber * get_implicit_subscriber()
```

There can only be one implicit *Publisher* and one implicit *Subscriber* per *DomainParticipant*. They are created with default QoS values (`DDS_PUBLISHER_QOS_DEFAULT`) and no Listener. For more information, see [Creating Publishers Explicitly vs. Implicitly \(Section 6.2.1 on page 248\)](#). You can use an implicit *Publisher* or implicit *Subscriber* just like an explicitly created one.

An implicit *Publisher/Subscriber* is deleted automatically when **delete_contained_entities()** is called. It can also be deleted by calling **delete_publisher/subscriber()** with the implicit *Publisher/Subscriber* as a parameter.

When a *DomainParticipant* is deleted, if there are no attached *DataReaders* that belong to the implicit *Subscriber* or no attached *DataWriters* that belong to the implicit *Publisher*, any implicit *Publisher/Subscriber* will be deleted by the middleware implicitly.

Note: It is not safe to create an implicit *Publisher/Subscriber* while another thread may be simultaneously calling `set_default_[publisher/subscriber]_qos()`.

How to get the implicit *Publisher/Subscriber*. (For simplicity, error handling is not shown.)

```
using namespace DDS;
...
Publisher * publisher = NULL;
Subscriber * subscriber = NULL;
PublisherQos publisher_qos;
SubscriberQos subscriber_qos;
...
publisher = participant->get_implicit_publisher();
/* Change implicit publisher QoS */
publisher->get_qos(publisher_qos);
publisher_qos.partition.name.maximum(3);
publisher_qos.partition.name.length(3);
publisher_qos.partition.name[0] = DDS_String_dup("partition_A");
publisher_qos.partition.name[1] = DDS_String_dup("partition_B");
publisher_qos.partition.name[2] = DDS_String_dup("partition_C");
publisher->set_qos(publisher_qos);
/* Get implicit subscriber */
subscriber = participant->get_implicit_subscriber();
/* Change implicit subscriber QoS */
subscriber_qos.partition.name.maximum(3);
subscriber_qos.partition.name.length(3);
subscriber_qos.partition.name[0] = DDS_String_dup("partition_A");
subscriber_qos.partition.name[1] = DDS_String_dup("partition_B");
subscriber_qos.partition.name[2] = DDS_String_dup("partition_C");
subscriber->set_qos(subscriber_qos);
```

8.3.10 Asserting Liveliness

The `assert_liveliness()` operation manually asserts the liveliness of all the *DataWriters* created by this *DomainParticipant* that has [LIVELINESS QoS Policy \(Section 6.5.13 on page 382\)](#) kind set to `MANUAL_BY_PARTICIPANT`. When `assert_liveliness()` is called, then for those *DataWriters* who have their `LIVELINESS` set to `MANUAL_BY_PARTICIPANT`, Connex DDS will send a packet to all matched *DataReaders* that indicates that the *DataWriter* is still alive.

However, the `LIVELINESS` contract of periodically sending liveliness packets to *DataReaders* is also fulfilled when the `write()`, `assert_liveliness()`, `unregister_instance()` and `dispose()` operations on a *DataWriter* itself is called. Those calls will also cause Connex DDS to send packets that indicate the liveliness of the *DataWriter*. Therefore, it is necessary for the application to call `assert_liveliness()` on the *DomainParticipant* only if those operations on a *DataWriter* are not being invoked within the period specified by the [LIVELINESS QoS Policy \(Section 6.5.13 on page 382\)](#)

8.3.11 Learning about Discovered DomainParticipants

The `get_discovered_participants()` operation provides you with a list of *DomainParticipants* that have been discovered in the DDS domain (except any that you have said to ignore via the `ignore_participant()` operation (see [Restricting Communication—Ignoring Entities \(Section 16.4 on page 784\)](#))).

Once you have a list of discovered *DomainParticipants*, you can get more information about them by calling the `get_discovered_participant_data()` operation. This operation can only be used on *DomainParticipants* that are in the same DDS domain and have not been marked as ‘ignored.’ Otherwise, the operation will fail and return `DDS_RETCODE_PRECONDITION_NOT_MET`. The returned information is of type `DDS_ParticipantBuiltinTopicData`, described in [Table 16.1 Participant Built-in Topic’s Data Type \(DDS_ParticipantBuiltinTopicData\)](#).

8.3.12 Learning about Discovered Topics

The `get_discovered_topics()` operation provides you with a list of *Topics* that have been discovered in the DDS domain (except any that you have said to ignore via the `ignore_topic()` operation (see [Restricting Communication—Ignoring Entities \(Section 16.4 on page 784\)](#))).

Once you have a list of discovered *Topics*, you can get more information about them by calling the `get_discovered_topic_data()` operation. This operation can only be used on *Topics* that have been created by a *DomainParticipant* in the same DDS domain as the participant on which this operation is invoked and must not have been "ignored" by means of the *DomainParticipant* `ignore_topic()` operation. Otherwise, the operation will fail and return `DDS_RETCODE_PRECONDITION_NOT_MET`. The returned information is of type `DDS_TopicBuiltinTopicData`, described in [Table 16.4 Topic Built-in Topic’s Data Type \(DDS_TopicBuiltinTopicData\)](#).

8.3.13 Other DomainParticipant Operations

8.3.13.1 Verifying Entity Containment

If you have a handle to an *Entity*, and want to see if that *Entity* was created from your *DomainParticipant* (or any of its *Publishers* or *Subscribers*), use the `contains_entity()` operation, which returns a boolean.

An *Entity*’s instance handle may be obtained from built-in topic data (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)), various statuses, or from the `get_instance_handle()` operation (see [Getting an Entity’s Instance Handle \(Section 4.1.3 on page 157\)](#)).

8.3.13.2 Getting the Current Time

The `get_current_time()` operation returns the current time value from the same time-source (clock) that Connex DDS uses to timestamp the data published by *DataWriters* (`source_timestamp` of the `SampleInfo` structure, see [The SampleInfo Structure \(Section 7.4.6 on page 504\)](#)). The time-sources used by Connex DDS do not have to be synchronized nor are they synchronized by Connex DDS.

See also: [Clock Selection \(Section 8.6 on page 619\)](#).

8.3.13.3 Getting All Publishers and Subscribers

The `get_publishers()` and `get_subscribers()` operations will provide you with a list of the *DomainParticipant's Publishers* and *Subscribers*, respectively.

8.4 DomainParticipantFactory QoS Policies

This section describes QoS Policies that are strictly for the *DomainParticipantFactory* (not the *DomainParticipant*). For a complete list of QoS Policies that apply to *DomainParticipantFactory*, see [Table 8.2 DomainParticipantFactory QoS](#).

- [LOGGING QoS Policy \(DDS Extension\) \(Section 8.4.1 below\)](#)
- [PROFILE QoS Policy \(DDS Extension\) \(Section 8.4.2 on the next page\)](#)
- [SYSTEM_RESOURCE_LIMITS QoS Policy \(DDS Extension\) \(Section 8.4.3 on page 575\)](#)

8.4.1 LOGGING QoS Policy (DDS Extension)

This QoS Policy configures the properties associated with the Connex DDS logging facility.

This QoS Policy includes the members in [Table 8.5 DDS_LoggingQoS Policy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

See also: [Controlling Messages from Connex DDS \(Section 21.2 on page 865\)](#) and [Configuring Logging via XML \(Section 21.2.2 on page 871\)](#).

Table 8.5 DDS_LoggingQoS Policy

Type	Field Name	Description
NDDS_ConfigLogVerbosity	verbosity	Specifies the verbosity at which Connex DDS diagnostic information will be logged.
NDDS_Config_LogCategory	category	Specifies the category for which logging needs to be enabled.
NDDS_Config_LogPrintFormat	print_format	Specifies the format to be used to output the Connex DDS diagnostic information.
char *	output_file	Specifies the file to which the logged output is redirected.

8.4.1.1 Example

```
DSDomainParticipantFactory *factory =
    DDSDomainParticipantFactory::get_instance();
DDS_DomainParticipantFactoryQos factoryQos;
DDS_ReturnCode_t retcode = factory->get_qos(factoryQos);
```

```

if (retcode != DDS_RETCODE_OK) {
    // error
}
factoryQos.logging.output_file = DDS_String_dup("myOutput.txt");
factoryQos.logging.verbosity = NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL;
factory->set_qos(factoryQos);

```

8.4.1.2 Properties

This QoSPolicy can be changed at any time.

Since it is only configuring logging, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

8.4.1.3 Related QoS Policies

- None

8.4.1.4 Applicable DDS Entities

- [DomainParticipantFactory](#) (Section 8.2 on page 539)

8.4.1.5 System Resource Considerations

Because the **output_file** will be freed by Connex DDS, you should use **DDS_String_dup()** to allocate the string when providing an **output_file**.

8.4.2 PROFILE QoS Policy (DDS Extension)

This QoSPolicy determines the way that XML documents containing QoS profiles are loaded.

All QoS values for *Entities* can be configured with QoS profiles defined in XML documents. XML documents can be passed to Connex DDS in string form, or more likely, through files found on a file system. This QoS configures how a *DomainParticipantFactory* loads the QoS profiles defined in XML. QoS profiles may be stored in this QoS as XML documents as a string. The location of XML files defining QoS profiles may be configured via this QoS. There are also default locations where the *DomainParticipantFactory* will look for files to load QoS profiles. You may disable any or all of these default locations using the Profile QoS. For more information about QoS profiles and libraries, please see [Configuring QoS with XML](#) (Section Chapter 17 on page 791).

This QoSPolicy includes the members in [Table 8.6 DDS_ProfileQoS Policy](#). For the defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.6 DDS_ProfileQosPolicy

Type	Field Name	Description
DDS_StringSeq	string_profile	Sequence of strings (empty by default) containing a XML document to load. The concatenation of the strings in this sequence must be a valid XML document according to the XML QoS profile schema.
	url_profile	A sequence of URL groups (empty by default) containing a set of XML documents to load. See URL Groups (Section 17.8 on page 814) .
DDS_Boolean	ignore_user_profile	When TRUE, the QoS profiles contained in the file USER_QOS_PROFILES.xml in the current working directory will be ignored.
	ignore_environment_profile	When TRUE, the value of the environment variable NDDS_QOS_PROFILES will be ignored.
	ignore_resource_profile	When TRUE, the QoS profiles in the file \$NDDSHOME/resource/xml/NDDS_QOS_PROFILES.xml will be ignored. NDDS_QOS_PROFILES.xml does not exist by default. However, NDDS_QOS_PROFILES.example.xml is shipped with the host bundle of the product; you can copy it to NDDS_QOS_PROFILES.xml and modify it for your own use.

In the Modern C++ API, there is not a PROFILE QoS Policy, because the class that manages QoS profiles is `dds::core::QosProvider`—not the `DomainParticipantFactory`. A `QosProvider` can receive a `QosProviderParams` instance, which encapsulates the fields described before.

8.4.2.1 Example

Traditional C++:

```

DDSDomainParticipantFactory *factory =
    DDSDomainParticipantFactory::get_instance();
DDS_DomainParticipantFactoryQos factoryQos;

DDS_ReturnCode_t retcode = factory->get_qos(factoryQos);
if (retcode != DDS_RETCODE_OK) {
    // error
}

const char *url_profiles[2] = {
    "file://usr/local/default_dds.xml",
    "file://usr/local/alternative_default_dds.xml" };
factoryQos.profile.url_profile.from_array(url_profiles, 2);

factoryQos.profile.ignore_resource_profile = DDS_BOOLEAN_TRUE;
factory->set_qos(factoryQos);

rti::core::QosProviderParams params =

```

```

dds::core::QosProvider::Default() -> default_provider_params();

std::vector<std::string> url_profiles = {
    "file://usr/local/default_dds.xml",
    "file://usr/local/alternative_default_dds.xml" };

params.url_profile(url_profiles);
params.ignore_resource_profile(true);

dds::core::QosProvider::Default() -> default_provider_params(params);

```

8.4.2.2 Properties

This QoS Policy can be changed at any time.

Since it is only for the DomainParticipantFactory, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

8.4.2.3 Related QoS Policies

- None

8.4.2.4 Applicable Entities

- [DomainParticipantFactory](#) (Section 8.2 on page 539)

8.4.2.5 System Resource Considerations

Once the QoS profiles are loaded, the DomainParticipantFactory will keep one copy of each QoS in the QoS profiles in memory.

You can free the memory associated with the XML QoS profiles by calling the DomainParticipantFactory's **unload_profiles()** operation.

8.4.3 SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension)

The SYSTEM_RESOURCE_LIMITS QoS Policy configures *DomainParticipant*-independent resources used by Connex DDS. Its main use is to change the maximum number of *DomainParticipants* that can be created within a single process (address space).

It contains the single member as shown in [Table 8.7 DDS_SystemResourceLimitsQoS Policy](#). For the default and valid range, please refer to the API Reference HTML documentation.

Table 8.7 DDS_SystemResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	max_objects_per_thread	Sizes the thread storage that is allocated on a per-thread basis when the thread calls Connex DDS APIs.

The only parameter that you can set, `max_objects_per_thread`, controls the size of thread-specific storage that is allocated by Connex DDS for every thread that invokes a Connex DDS API. This storage is used to cache objects that have to be created on a per-thread basis when a thread traverses different portions of Connex DDS internal code.

Thus instead of dynamically creating and destroying the objects as a thread enters and leaves different parts of the code, Connex DDS caches the objects by storing them in thread-specific storage. We assume that a thread will repeatedly call Connex DDS APIs so that the objects cached will be needed again and again.

The number of objects that will be stored in the cache depends the number of APIs (sections of Connex DDS code) that a thread invokes. It also depends on the number of different *DomainParticipants* with which the thread interacts. For a single *DomainParticipant*, the maximum number of objects that could be stored is a constant— independent of the number of *Entities* created in or by the participant. A safe number to use is 200 objects per *DomainParticipant*.

A user thread that only interacts with a single *DomainParticipant* or the *Entities* thereof, would never have more than 200 objects stored in its cache. However, if the same thread invokes Connex DDS APIs on other *Entities* of other *DomainParticipants*, the maximum number of objects that may be stored will increase with the number of participants involved.

The default setting of this resource should work for most user applications. However, if your application uses more than 4 *DomainParticipants*, you may need to increase the value of `max_objects_per_thread`.

8.4.3.1 Example

Say an application uses 10 *DomainParticipants*. If a single thread was used to create all 10 *DomainParticipants*, or a single thread is used to call `write()` on *DataWriters* belonging to all 10 participants, it is possible to run out of thread-specific storage. Either the creation of the participant or the `write()` will fail.

In that case, you will need to increase the value of `max_objects_per_thread`.

8.4.3.2 Properties

This QoS policy cannot be modified after the *DomainParticipantFactory* is used to create the first *DomainParticipant* or *WaitSet* in an application.

This QoS can be set differently in different applications.

8.4.3.3 Related QoS Policies

There are no interactions with other QoS Policies.

8.4.3.4 Applicable Dds Entities

- [DomainParticipantFactory](#) (Section 8.2 on page 539)

8.4.3.5 System Resource Considerations

Increasing the value of `max_objects_per_thread` will increase the amount of memory allocated by Connex DDS for every thread that access Connex DDS code. This includes internal Connex DDS threads as well as user threads. Each object uses about 32 bytes of memory.

8.5 DomainParticipant QoS Policies

This section describes the QoS Policies that are strictly for *DomainParticipants* (and no other types of Entities). For a complete list of QoS Policies that apply to *DomainParticipant*, see [Table 8.4 DomainParticipant QoS Policies](#).

- [DATABASE QoS Policy \(DDS Extension\)](#) (Section 8.5.1 below)
- [DISCOVERY QoS Policy \(DDS Extension\)](#) (Section 8.5.2 on page 580)
- [DISCOVERY_CONFIG QoS Policy \(DDS Extension\)](#) (Section 8.5.3 on page 585)
- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QoS Policy \(DDS Extension\)](#) (Section 8.5.4 on page 593)
- [EVENT QoS Policy \(DDS Extension\)](#) (Section 8.5.5 on page 602)
- [RECEIVER_POOL QoS Policy \(DDS Extension\)](#) (Section 8.5.6 on page 604)
- [TRANSPORT_BUILTIN QoS Policy \(DDS Extension\)](#) (Section 8.5.7 on page 606)
- [TRANSPORT_MULTICAST_MAPPING QoS Policy \(DDS Extension\)](#) (Section 8.5.8 on page 608)
- [WIRE_PROTOCOL QoS Policy \(DDS Extension\)](#) (Section 8.5.9 on page 610)

8.5.1 DATABASE QoS Policy (DDS Extension)

The Database QoS Policy configures how Connex DDS manages its internal database, including how often it cleans up, the priority of the database thread, and limits on resources that may be allocated by the database. RTI uses an internal in-memory database to store information about entities created locally as well as remote entities found during the discovery process. This database uses a background thread to garbage-collect records related to deleted entities. When the *DomainParticipant* that maintains this database is deleted, it shuts down this thread.

It includes the members in [Table 8.8 DDS_DatabaseQoSPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.8 DDS_DatabaseQoSPolicy

Type	Field Name	Description
DDS_ThreadSettings_t	thread.mask thread.priority thread.stack_size	Thread settings for the database thread used by Connex DDS to periodically remove deleted records from the database. The values used for these settings are OS-dependent; see the RTI Connex DDS Core Libraries Platform Notes for details. Note: thread.cpu_list and thread.cpu_rotation are not relevant in this QoS policy.
DDS_Duration_t	shutdown_timeout	The maximum time that the <i>DomainParticipant</i> will wait for the database thread to terminate when the participant is destroyed.
DDS_Duration_t	cleanup_period	The period at which the database thread wakes up to removed deleted records.
DDS_Duration_t	shutdown_cleanup_period	The period at which the database thread wakes up to removed deleted records when the <i>DomainParticipant</i> is being destroyed.
DDS_Long	initial_records	The number of records that is initially created for the database. These records hold information for both local and remote entities that are dynamically created or discovered.
DDS_Long	max_skiplist_level	This is a performance tuning parameter that optimizes the time it takes to search the database for a record. A ‘Skip List’ is an algorithm for maintaining a list that is faster to search than a binary tree. This value should be set to $\log_2(N)$, where N is the maximum number of elements that will be stored in a single list. The list that stores the records for remote <i>DataReaders</i> or the one for remote <i>DataWriters</i> tend to have the most entries. So, the number of <i>DataWriters</i> or <i>DataReaders</i> in a system across all <i>DomainParticipants</i> in a single DDS domain, which ever is greater, can be used to set this parameter.
DDS_Long	max_weak_references	This parameter sets the maximum number of entries in the weak reference table. Weak references are used as a technique for ensuring that unreferenced objects are deleted. The actual number of weak references is permitted to grow from the value set by initial_weak_references to this maximum. To prevent Connex DDS from allocating memory for weak references after initialization, you should set the initial and maximum weak references to the same value. However, it is difficult to calculate how many weak references an application will use. To allow Connex DDS to grow the weak reference table as needed, and thus dynamically allocate memory, you should set the value of this field to DDS_LENGTH_UNLIMITED, the default setting.
DDS_Long	initial_weak_references	The initial number of entries in the weak reference table. See max_weak_references. Connex DDS may decide to use a larger initial value if initial_weak_references is set too small. If you access this parameter after a <i>DomainParticipant</i> has been created, you will see the actual value used.

You may be interested in modifying the **shutdown_timeout** and **shutdown_cleanup_period** parameters to decrease the time it takes to delete a *DomainParticipant* when your application is shutting down.

The [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\)](#) (Section 8.5.4 on page 593) controls the memory allocation for elements stored in the database.

Real-time programmers will probably want to adjust the priorities of all of the threads created by Connex DDS relative to each other as well as relative to non-Connex DDS threads in their applications. [Connex DDS Threading Model \(Section Chapter 19 on page 837\)](#), [EVENT QosPolicy \(DDS Extension\)](#) (Section 8.5.5 on page 602), and [RECEIVER_POOL QosPolicy \(DDS Extension\)](#) (Section 8.5.6 on page 604) discuss the other threads that are created by Connex DDS.

A record in the database can be deleted only when no threads are using it. Connex DDS uses a thread that periodically checks the database if records that have been marked for deletion can be removed. This period is set by `cleanup_period`. When a *DomainParticipant* is being destroyed, the thread will wake up faster at the `shutdown_cleanup_period` as other threads delete and release records in preparation for shutting down.

On Windows and VxWorks systems, the thread that is destroying the *DomainParticipant* may block up to `shutdown_timeout` seconds while waiting for the database thread to finish removing all records and terminating. On other operating systems, the thread destroying the *DomainParticipant* will block as long as required for the database thread to terminate.

The default values for those and the rest of the parameters in this QosPolicy should be sufficient for most applications.

8.5.1.1 Example

The priority of the database thread should be set to the lowest priority among all threads in a real-time system. Although, the database thread should not be permitted to starve, the work that it performs is non-time-critical.

8.5.1.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.1.3 Related QosPolicies

- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\)](#) (Section 8.5.4 on page 593)
- [EVENT QosPolicy \(DDS Extension\)](#) (Section 8.5.5 on page 602)
- [RECEIVER_POOL QosPolicy \(DDS Extension\)](#) (Section 8.5.6 on page 604)

8.5.1.4 Applicable Dds Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.1.5 System Resource Considerations

Setting the thread parameters correctly on a real-time operating system is usually critical to the proper overall functionality of the applications on that system. Larger values for the `thread.stack_size` parameter will use up more memory.

Smaller values for the `cleanup_period` and `shutdown_cleanup_period` will cause the database thread to wake up more frequently using more CPU.

Connex DDS is permitted to use up more memory for larger values of **max_skiplist_level** and **max_weak_references**. Whether or not more memory is actually used depends on actual operating conditions.

8.5.2 DISCOVERY QoS Policy (DDS Extension)

The DISCOVERY QoS configures how *DomainParticipants* discover each other on the network. It identifies where on the network this application can potentially discover other applications with which to communicate. The middleware will periodically send network packets to these locations, announcing itself to any remote applications that may be present, and will listen for announcements from those applications. The discovery process is described in detail in [Discovery \(Section Chapter 14 on page 709\)](#).

This QoS Policy includes the members in [Table 8.9 DDS_DiscoveryQoSPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.9 DDS_DiscoveryQoSPolicy

Type	Field Name	Description
DDS_StringSeq	enabled_transports	Transports available for use by the discovery process. See Transports Used for Discovery (Section 8.5.2.1 on the next page) .
DDS_StringSeq	initial_peers	Unicast locators (address/indices) of potential participants with which this <i>DomainParticipant</i> will attempt to establish communications. See Setting the ‘Initial Peers’ List (Section 8.5.2.2 on the next page) .
DDS_StringSeq	multicast_receive_addresses	List of multicast addresses on which Discovery-related messages can be received by the <i>DomainParticipant</i> . See Configuring Multicast Receive Addresses (Section 8.5.2.4 on page 582) .
DDS_Long	metatraffic_transport_priority	Transport priority to be used for sending Discovery messages. See Meta-Traffic Transport Priority (Section 8.5.2.5 on page 583) .
DDS_Boolean	accept_unknown_peers	Whether to accept a participant discovered via unicast that is not in the initial_peers list. See Controlling Acceptance of Unknown Peers (Section 8.5.2.6 on page 583) .

Table 8.9 DDS_DiscoveryQosPolicy

Type	Field Name	Description
DDS_Boolean	enable_endpoint_discovery	Whether endpoint discovery will automatically occur with discovered <i>DomainParticipants</i> . See Supervising Endpoint Discovery (Section 16.4.5 on page 788) .

8.5.2.1 Transports Used for Discovery

The `enabled_transports` field allows you to specify the set of installed and enabled transports that can be used to discover other *DomainParticipants*. This field is a sequence of strings where each string specifies an alias of a registered (and thus installed and enabled) transport. Please see the API Reference HTML documentation (select **Modules, RTI Connex DDS API Reference, Pluggable Transports**) for more information.

8.5.2.2 Setting the ‘Initial Peers’ List

When a *DomainParticipant* is created, it needs to find other participants in the same DDS domain—this is known as the ‘discovery process’ which is discussed in [Discovery \(Section Chapter 14 on page 709\)](#). One way to do so is to use this QosPolicy to specify a list of potential participants. This is the role of the parameter `initial_peers`. The strings containing peer descriptors are stored in the `initial_peers` string sequence. The format of a string discussed in [Peer Descriptor Format \(Section 14.2.1 on page 713\)](#).

The peers stored in `initial_peers` are merely *potential* peers—there is no requirement that the peer *DomainParticipants* are actually up and running or even will eventually exist. The Connex DDS discovery process will try to contact all potential peer participants in the list periodically using unicast transports (as configured by the [DISCOVERY_CONFIG QosPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#)).

The `initial_peers` parameter can be modified in source code or it can be initialized from an environment variable, `NDDS_DISCOVERY_PEERS` or from a text file, see [Configuring the Peers List Used in Discovery \(Section 14.2 on page 711\)](#).

8.5.2.3 Adding and Removing Peers List Entries

The *DomainParticipant*’s `add_peer()` operation adds a peer description to the internal peer list that was initialized by the `initial_peer` field of the `DISCOVERY` QosPolicy.

```
DDS_ReturnCode_t DDSDomainParticipant::add_peer (
    const char* peer_desc)
```

The `peer_desc` string must be formatted as specified in [Peer Descriptor Format \(Section 14.2.1 on page 713\)](#).

You can call this operation any time after the *DomainParticipant* has been enabled. An attempt will be made to contact the new peer immediately.

Adding peers with this operation has no effect on the `initial_peers` list. After a *DomainParticipant* has been created, the contents of the `initial_peers` field merely shows what the internal peer list was initialized to be. Therefore, `initial_peers` may not reflect the actual potential peer list used by a *DomainParticipant*. Furthermore, if you call `get_qos()`, the returned list of peers will not include the added peer—`get_qos()` will only show you what is set in the **initial_peers** list.

A peer added with `add_peer()` is *not* considered to be “unknown.” (That is, you may have `accept_unknown_peers` ([Controlling Acceptance of Unknown Peers \(Section 8.5.2.6 on the next page\)](#)) set to `FALSE` and still use `add_peer()`.)

You can remove an entry from the list with `remove_peer()`.

You can ignore data from a participant by using the `ignore_participant()` operation described in [Restricting Communication—Ignoring Entities \(Section 16.4 on page 784\)](#).

8.5.2.4 Configuring Multicast Receive Addresses

The `multicast_receive_addresses` field in the `DISCOVERY` `QosPolicy` is a sequence of strings that specifies a set of multicast group addresses on which the *DomainParticipant* will listen for discovery meta-traffic. Each string must have a valid multicast address in either IPv4 dot notation or IPv6 presentation format. Please look at publicly available documentation of the IPv4 and IPv6 standards for the definition and valid address ranges for multicast.

The `multicast_receive_addresses` field can be initialized from multicast addresses that appear in the `NDDS_DISCOVERY_PEERS` environment variable or text file, see [Configuring the Peers List Used in Discovery \(Section 14.2 on page 711\)](#). A multicast address found in the environment variable or text file will be added both to the `initial_peers` and `multicast_receive_addresses` fields. Note that the addresses in `initial_peers` are ones in which the *DomainParticipant* will *send* discovery meta-traffic, and the ones in `multicast_receive_addresses` are used for *receiving* discovery meta-traffic.

If `NDDS_DISCOVERY_PEERS` does *not* contain a multicast address, then **multicast_receive_addresses** is cleared and the RTI discovery process will not listen for discovery messages via multicast.

If `NDDS_DISCOVERY_PEERS` contains one or more multicast addresses, the addresses are stored in **multicast_receive_addresses**, starting at element 0. They will be stored in the order in which they appear in `NDDS_DISCOVERY_PEERS`.

Note: Currently, Connex DDS will only listen for discovery traffic on the first multicast address (element 0) in **multicast_receive_addresses**.

If you want to send discovery meta-traffic on a different set of multicast addresses than you want to receive discovery meta-traffic, set `initial_peers` and `multicast_receive_addresses` via the `QosPolicy` API.

8.5.2.5 Meta-Traffic Transport Priority

The `metatraffic_transport_priority` field is used to specify the transport priority to be used for sending all discovery meta-traffic. See the [TRANSPORT_PRIORITY QosPolicy \(Section 6.5.22 on page 409\)](#) for details on how transport priorities may be used.

8.5.2.6 Controlling Acceptance of Unknown Peers

The `accept_unknown_peers` field controls whether or not a *DomainParticipant* is allowed to communicate with other *DomainParticipants* found via unicast transport that are not in its peers list (which is the combination of the `initial_peers` list and any peers added with the `add_peer()` operation described in [Adding and Removing Peers List Entries \(Section 8.5.2.3 on page 581\)](#)).

Suppose Participant A is included in Participant B's initial peers list, but Participant B is not in Participant A's list. When Participant B contacts Participant A by sending it a unicast discovery packet, then Participant A has a choice:

- If `accept_unknown_peers` is `DDS_BOOLEAN_TRUE`, then Participant A will reply to Participant B, and communications will be established.
- If `accept_unknown_peers` is `DDS_BOOLEAN_FALSE`, then Participant A will ignore Participant B, and A and B will never talk.

Note that Participants do not exchange peer lists. So if Participant A knows about Participant B, and Participant B knows about Participant C, Participant A will not discover Participant C.

Note: If `accept_unknown_peers` is false and shared memory is disabled, applications on the same node will not communicate if only 'localhost' is specified in the peer list. If shared memory is disabled or 'shmem://' is not specified in the peer list, if you want to communicate with other applications on the same node through the loopback interface, you must put the actual node address or hostname in `NDDS_DISCOVERY_PEERS`.

8.5.2.7 Example

You will always use this policy to set the `participant_id` when you want to run more than one *DomainParticipant* in the same DDS domain on the same host.

The easiest way to set the initial peers list is to use the `NDDS_DISCOVERY_PEERS` environment variable. However, should you want asymmetric multicast addresses for sending or receiving meta-traffic, you will need to use this QosPolicy directly.

A reason to use asymmetric multicast addresses is to take advantage of the efficiency provided by using multicast, while at the same time preventing all participants from discovering each other. For example, suppose you have a system in which you have a single server node and a hundred client nodes. The client

nodes do not publish or subscribe to each other's data and thus never need to know about each others existence.

If we did not use multicast, we would have to populate the server application's peer list with 100 peer descriptors for each of the client nodes. Each client application would only need to have the server application in its peer list. The maintenance of the list is unwieldy, especially if nodes are constantly reconfigured and addresses changed. In addition, the server will send out discovery packets on a per client basis since the peer list essentially holds 100 unicast addresses.

Instead, if we used a single multicast address in the `NDDS_DISCOVERY_PEERS` environment variable, the server and all of the clients would discover each other. Certainly, the list is easier to maintain, but the total amount of traffic has actually increased since the clients are now exchanging packets with each other uselessly.

To keep the list maintainable, as well as to minimize discovery traffic, we can have the server send out packets on a multicast address by modifying its `initial_peer` field. The clients would have their `multicast_receive_addresses` field set to the same address used by the server. The `initial_peers` of the clients would only need the single unicast peer descriptor of the server as before.

Now, the server can send a single packet that will be received by all of the clients, but the clients will not discover each other because they never send out a multicast packet themselves.

8.5.2.8 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.2.9 Related QosPolicies

- [DISCOVERY_CONFIG QosPolicy \(DDS Extension\)](#) (Section 8.5.3 on the next page)
- [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\)](#) (Section 8.5.7 on page 606)

8.5.2.10 Applicable Entities

- [DomainParticipants](#) (Section 8.3 on page 547)

8.5.2.11 System Resource Considerations

For every entry in the `initial_peers` list, Connex DDS will periodically send a discovery packet to see if that participant exists. If the list has many potential participants that are never started, then CPU and network bandwidth may be wasted in sending out packets that will never be received.

8.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension)

The DISCOVERY_CONFIG QosPolicy is used to tune the discovery process. It controls how often to send discovery packets, how to determine when participants are alive or dead, and resources used by the discovery mechanism.

The amount of network traffic required by the discovery process can vary widely based on how your application has chosen to configure the middleware's network addressing (e.g. unicast vs. multicast, multicast TTL, etc.), the size of the system, whether all applications are started at the same time or whether start times are staggered, and other factors. Your application can use this policy to make trade-offs between discovery completion time and network bandwidth utilization. In addition, you can introduce random back-off periods into the discovery process to decrease the probability of network contention when many applications start simultaneously.

This QosPolicy includes the members in [Table 8.10 DDS_DiscoveryConfigQosPolicy](#). Many of these members are described in [Discovery \(Section Chapter 14 on page 709\)](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.10 DDS_DiscoveryConfigQosPolicy

Type	Field Name	Description
DDS_Duration_t	participant_liveliness_lease_duration	The time period after which other DomainParticipants can consider this one dead if they do not receive a liveliness packet from this DomainParticipant.
DDS_Duration_t	participant_liveliness_assert_period	The period of time at which this DomainParticipant will send out packets asserting that it is alive.
DDS_RemoteParticipantPurgeKind	remote_participant_purge_kind	Controls the DomainParticipant's behavior for purging records of remote participants (and their contained entities) with which discovery communication has been lost. See Controlling Purging of Remote Participants (Section 8.5.3.2 on page 591) .
DDS_Duration_t	max_liveliness_loss_detection_period	The maximum amount of time between when a remote entity stops maintaining its liveliness and when the matched local entity realizes that fact.
DDS_Long	initial_participant_announcements	Sets how many initial liveliness announcements the DomainParticipant will send when it is first enabled, or after discovering a new remote participant.

Table 8.10 DDS_DiscoveryConfigQoSPolicy

Type	Field Name	Description
DDS_Duration_t	min_initial_participant_announcement_period	Sets the minimum and maximum times between liveliness announcements.
DDS_Duration_t	max_initial_participant_announcement_period	When a participant is first enabled, or after discovering a new remote participant, Connex DDS sends initial_participant_announcements number of discovery messages. These messages are sent with a sleep period between them that is a random duration between min_initial_participant_announcement_period and max_initial_participant_announcement_period.
DDS_BuiltinTopicReaderResourceLimits_t (Section Table 8.11 on page 589)	participant_reader_resource_limits	Configures the resource for the built-in DataReaders used to access discovery information; see Resource Limits for Builtin-Topic DataReaders (Section 8.5.3.1 on page 589) and Built-In Topics (Section Chapter 16 on page 772) .
DDS_RtpsReliableReaderProtocol_t (Section Table 7.20 on page 514)	publication_reader	Configures the RTPS reliable protocol parameters for a built-in publication reader.
DDS_BuiltinTopicReaderResourceLimits_t (Section Table 8.11 on page 589)	publication_reader_resource_limits	Configures the resource for the built-in DataReaders used to access discovery information; see Resource Limits for Builtin-Topic DataReaders (Section 8.5.3.1 on page 589) and Built-In Topics (Section Chapter 16 on page 772) .
DDS_RtpsReliableReaderProtocol_t (Section Table 7.20 on page 514)	subscription_reader	Configures the RTPS reliable protocol parameters for a built-in subscription reader. Built-in subscription readers receive discovery information reliably from DomainParticipants that were dynamically discovered (see Discovery (Section Chapter 14 on page 709)).
DDS_BuiltinTopicReaderResourceLimits_t (Section Table 8.11 on page 589)	subscription_reader_resource_limits	Configures the resource for the built-in DataReaders used to access discovery information; see Resource Limits for Builtin-Topic DataReaders (Section 8.5.3.1 on page 589) and Built-In Topics (Section Chapter 16 on page 772) .
DDS_RtpsReliableWriterProtocol_t (Section Table 6.37 on page 350)	publication_writer	Configures the RTPS reliable protocol parameters for the writer side of a reliable connection. Built-in DataWriters send reliable discovery information to DomainParticipants that were dynamically discovered (see Discovery (Section Chapter 14 on page 709)).
WRITER_DATA_LIFECYCLE QoS Policy (Section 6.5.27 on page 419)	publication_writer_data_lifecycle	Configures writer data-lifecycle settings for a built-in publication writer. (DDS_WriterDataLifecycleQoSPolicy::autodispose_unregistered_instances will always be TRUE.)
DDS_RtpsReliableWriterProtocol_t (Section Table 6.37 on page 350)	subscription_writer	Configures the RTPS reliable protocol parameters for the writer side of a reliable connection. Built-in DataWriters send reliable discovery information to DomainParticipants that were dynamically discovered (see Discovery (Section Chapter 14 on page 709)).

Table 8.10 DDS_DiscoveryConfigQoSPolicy

Type	Field Name	Description
WRITER_DATA_LIFECYCLE QoS Policy (Section 6.5.27 on page 419)	subscription_writer_data_lifecycle	Configures writer data-lifecycle settings for a built-in subscription writer. (DDS_WriterDataLifecycleQoSPolicy::autodispose_unregistered_instances will always be TRUE.)
DDS_DiscoveryConfigBuiltinPluginKindMask	builtin_discovery_plugins	The kind mask for selecting built-in discovery plugins: <ul style="list-style-type: none"> Simple Discovery Protocol: DDS_DISCOVERYCONFIG_BUILTIN_SDP Enterprise Discovery Service: DDS_DISCOVERYCONFIG_BUILTIN_EDS (Requires a separate component, RTI Enterprise Discovery Service.)
DDS_Duration_t	default_domain_announcement_period	The period at which a participant will announce itself to the default DDS domain 0 using the default UDPv4 multicast group address for discovery traffic on that DDS domain. For DDS domain 0, the default discovery multicast address is 239.255.0.1:7400. To disable announcement to the default DDS domain, set this to DURATION_INFINITE. When this period is set to a value other than DURATION_INFINITE and ignore_default_domain_announcements (see below) is FALSE, you can get information about participants running in different DDS domains by creating a participant in DDS domain 0 and implementing the on_data_available callback (see DATA_AVAILABLE Status (Section 7.3.7.1 on page 471)) in the ParticipantBuiltinTopicData built-in DataReader's listener (see Built-in DataReaders (Section 16.2 on page 773)). You can learn the domain ID associated with a participant by looking at the domain_id (Section on page 774) in the ParticipantBuiltinTopicData.
DDS_Boolean	ignore_default_domain_announcements	When TRUE, ignores the announcements received by a participant on the default DDS domain 0 corresponding to participants running on domains IDs other than 0. This setting only applies to participants running on the default DDS domain 0 and using the default port mapping. When TRUE, a participant running on the default DDS domain 0 will ignore announcements from participants running on different DDS domain IDs. When FALSE, a participant running on the default DDS domain 0 will provide announcements from participants running on different DDS domain IDs to the application via the ParticipantBuiltinTopicData built-in DataReader (see Built-in DataReaders (Section 16.2 on page 773)).

Table 8.10 DDS_DiscoveryConfigQosPolicy

Type	Field Name	Description
DDS_RtpsReliableReaderProtocol_t (Section Table 7.20 on page 514)	participant_message_reader	RTPS protocol-related configuration settings for a built-in participant message reader.
DDS_ReliabilityQosPolicyKind See Table 6.59 DDS_ReliabilityQosPolicy	participant_message_reader_reliability_kind	Reliability kind configuration setting for a built-in participant message reader (default: best-effort).
DDS_RtpsReliableWriterProtocol_t (Section Table 6.37 on page 350)	participant_message_writer	RTPS protocol-related configuration settings for a built-in participant message writer.
PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.18 on page 397)	publication_writer_publish_mode	Determines whether the Discovery built-in publication DataWriter publishes data synchronously or asynchronously and how.
PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.18 on page 397)	subscription_writer_publish_mode	Determines whether the Discovery built-in subscription DataWriter publishes data synchronously or asynchronously and how.
ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1 on page 313)	asynchronous_publisher	Asynchronous publishing settings for the Discovery Publisher and all entities that are created by it.

A *DomainParticipant* needs to send a message periodically to other *DomainParticipants* to let the other participants know that it is still alive. These liveliness messages are sent to all peers in the peer list that was initialized by the `initial_peers` parameter of the [DISCOVERY QosPolicy \(DDS Extension\)](#) (Section [8.5.2](#) on page 580). Peer participants on the peer list may or may not be alive themselves. The peer *DomainParticipants* that already know about this *DomainParticipant* will use the `participant_liveliness_lease_duration` provided by *this* participant to declare the participant dead, if they have not received a liveliness message for the specified time.

The `participant_liveliness_assert_period` is the periodic rate at which this *DomainParticipant* will be sending liveliness messages. Since these liveliness messages are not sent reliably and can get dropped by the transport, it is important to set:

$$\text{participant_liveliness_assert_period} < \text{participant_liveliness_lease_duration}/N$$

where *N* is the number of liveliness messages that other *DomainParticipants* must miss before they decide that this *DomainParticipant* is dead.

DomainParticipants that receive a liveliness message from a participant that they did not know about previously will have “discovered” the participant. When one *DomainParticipant* discovers another, the discoverer will immediately send its own liveliness packets back. `initial_participant_announcements` controls

how many of these initial liveliness messages are sent, and `max_initial_participant_announcement_period` controls the time period in between each message.

After the initial set of liveliness messages are sent, the *DomainParticipant* will return to sending liveliness packets to all peers in its peer list at the rate governed by `participant_liveliness_assert_period`.

For more information on the discovery process, see [Discovery \(Section Chapter 14 on page 709\)](#).

8.5.3.1 Resource Limits for Builtin-Topic DataReaders

The `DDS_BuiltinTopicReaderResourceLimits_t` structure is shown in [Table 8.11 DDS_BuiltinTopicReaderResourceLimits_t](#). This structure contains several fields that are used to configure the resource limits of the builtin-topic *DataReaders* used to receive discovery meta-traffic from other *DomainParticipants*.

Table 8.11 DDS_BuiltinTopicReaderResourceLimits_t

Type	Field Name	Description
DDS_ Long	<code>initial_samples</code>	Initial number of meta-traffic DDS data samples that can be stored by a builtin-topic <i>DataReader</i> .
	<code>max_samples</code>	Maximum number of meta-traffic DDS data samples that can be stored by a builtin-topic <i>DataReader</i> .
	<code>initial_infos</code>	Initial number of <code>DDS_SampleInfo</code> structures allocated for the builtin-topic <i>DataReader</i> .
	<code>max_infos</code>	Maximum number of <code>DDS_SampleInfo</code> structures that can be allocated for the built-in topic <i>DataReader</i> . max_infos must be \geq max_samples
	<code>initial_outstanding_reads</code>	Initial number of times in which memory can be concurrently loaned via read/take calls on the builtin-topic <i>DataReader</i> without being returned with <code>return_loan()</code> .
	<code>max_outstanding_reads</code>	Maximum number of times in which memory can be concurrently loaned via read/take calls on the builtin-topic <i>DataReader</i> without being returned with <code>return_loan()</code> .
	<code>max_samples_per_read</code>	Maximum number of DDS samples that can be read/taken on a same built-in topic <i>DataReader</i> .
DDS_ Boolean	<code>disable_fragmentation_support</code>	Determines whether the builtin-topic <i>DataReader</i> can receive fragmented DDS samples. When fragmentation support is not needed, disabling fragmentation support will save some memory resources.

Table 8.11 DDS_BuiltinTopicReaderResourceLimits_t

Type	Field Name	Description
DDS_Long	max_fragmented_samples	<p>The maximum number of DDS samples for which the <i>DataReader</i> may store fragments at a given point in time.</p> <p>At any given time, a <i>DataReader</i> may store fragments for up to max_fragmented_samples DDS samples while waiting for the remaining fragments. These DDS samples need not have consecutive sequence numbers and may have been sent by different <i>DataWriters</i>. Once all fragments of a DDS sample have been received, the DDS sample is treated as a regular DDS sample and becomes subject to standard QoS settings, such as max_samples. Connex DDS will drop fragments if the max_fragmented_samples limit has been reached.</p> <p>For best-effort communication, Connex DDS will accept a fragment for a new DDS sample, but drop the oldest fragmented DDS sample from the same remote writer.</p> <p>For reliable communication, Connex DDS will drop fragments for any new DDS samples until all fragments for at least one older DDS sample from that writer have been received.</p> <p>Only applies if disable_fragmentation_support is FALSE.</p>
DDS_Long	initial_fragmented_samples	<p>The initial number of DDS samples for which a builtin-topic <i>DataReader</i> may store fragments.</p> <p>Only applies if disable_fragmentation_support (Section on the previous page) is FALSE.</p>
DDS_Long	max_fragmented_samples_per_remote_writer	<p>The maximum number of DDS samples per remote writer for which a builtin-topic <i>DataReader</i> may store fragments.</p> <p>Logical limit so a single remote writer cannot consume all available resources.</p> <p>Only applies if disable_fragmentation_support (Section on the previous page) is FALSE.</p>
DDS_Long	max_fragments_per_sample	<p>Maximum number of fragments for a single DDS sample.</p> <p>Only applies if disable_fragmentation_support (Section on the previous page) is FALSE.</p>
DDS_Boolean	dynamically_allocate_fragmented_samples	<p>By default, the middleware does not allocate memory upfront, but instead allocates memory from the heap upon receiving the first fragment of a new sample. The amount of memory allocated equals the amount of memory needed to store all fragments in the sample. Once all fragments of a sample have been received, the sample is deserialized and stored in the regular receive queue. At that time, the dynamically allocated memory is freed again.</p> <p>This QoS setting is useful for large, but variable-sized data types where up-front memory allocation for multiple samples based on the maximum possible sample size may be expensive. The main disadvantage of not pre-allocating memory is that one can no longer guarantee the middleware will have sufficient resources at run-time.</p> <p>If dynamically_allocate_fragmented_samples is FALSE, the middleware will allocate memory up-front for storing fragments for up to initial_fragmented_samples samples. This memory may grow up to max_fragmented_samples if needed.</p> <p>Only applies if disable_fragmentation_support (Section on the previous page) is FALSE.</p>

There are builtin-topics for exchanging data about *DomainParticipants*, for publications (*Publisher/DataWriter* combination) and for subscriptions (*Subscriber/DataReader* combination). The *DataReaders* for the publication and subscription builtin-topics are reliable. The *DataReader* for the participant builtin-topic is best effort.

You can set listeners on these *DataReaders* that are created automatically when a *DomainParticipant* is created. With these listeners, your code can be notified when remote *DomainParticipants*, *Publishers/DataWriters*, and *Subscriber/DataReaders* are discovered. You can always check the receive queues of those *DataReaders* for the same information about discovered entities at any time. Please see [Built-In Topics \(Section Chapter 16 on page 772\)](#) for more details.

The `initial_samples` and `max_samples`, and related `initial_infos` and `max_infos`, fields size the amount of declaration messages can be stored in each builtin-topic *DataReader*.

8.5.3.2 Controlling Purging of Remote Participants

When discovery communication with a remote participant has been lost, the local participant must make a decision about whether to continue attempting to communicate with that participant and its contained entities. The `remote_participant_purge_kind` is used to select the desired behavior.

This does not pertain to the situation in which a remote participant has been gracefully deleted and notification of that deletion has been successfully received by its peers. In that case, the local participant will immediately stop attempting to communicate with those entities and will remove the associated remote entity records from its internal database.

The `remote_participant_purge_kind` can be set to the following values:

DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE

This value causes Connex DDS to keep the state of a remote participant and its contained entities for as long as the participant maintains its liveliness contract (as specified by its `participant_liveliness_lease_duration` in the [DISCOVERY_CONFIG QosPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#)).

A participant will maintain its own liveliness to any remote participant via inter-participant liveliness traffic (see [LIVELINESS QosPolicy \(Section 6.5.13 on page 382\)](#)).

The default Simple Discovery Protocol described in [Discovery \(Section Chapter 14 on page 709\)](#) automatically maintains this liveliness, whereas other discovery mechanisms may or may not.

DDS_NO_REMOTE_PARTICIPANT_PURGE

With this value, Connex DDS will never purge the records of a remote participant with which discovery communication has been lost.

- If the remote participant is later rediscovered, the records that remain in the database will be re-used.
- If the remote participant is not rediscovered, the records will continue to take up space in the database for as long as the local participant remains in existence.

In most cases, you will *not* need to change this value from its default, `DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE`.

However, `DDS_NO_REMOTE_PARTICIPANT_PURGE` may be a good choice if the following conditions apply:

Discovery communication with a remote participant may be lost while data communication remains intact. This will not be the typical case if discovery takes place over the Simple Discovery Protocol, but may occur if you are using *RTI Enterprise Discovery Service*.¹

Extensive and prolonged lack of discovery communication between participants is not expected to be common, either because loss of the participant will be rare, or because participants may be lost sporadically but will typically return again.

Maintaining inter-participant liveness is problematic, perhaps because a participant has no writers with the appropriate `LIVELINESS QosPolicy` (Section 6.5.13 on page 382) `kind`.

8.5.3.3 Controlling the Reliable Protocol Used by Builtin-Topic DataWriters/DataReaders

The connection between the *DataWriters* and *DataReaders* for the publication and subscription builtin-topics are reliable. The `publication_writer`, `subscription_writer`, `publication_reader`, and `subscription_reader` parameters of the `DISCOVERY_CONFIG QosPolicy (DDS Extension)` (Section 8.5.3 on page 585) configure the reliable messaging protocol used by Connex DDS for those topics. Connex DDS's reliable messaging protocol is discussed in *Reliable Communications* (Section Chapter 10 on page 629).

See also:

- `DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)` (Section 6.5.3 on page 347)
- `DATA_READER_PROTOCOL QosPolicy (DDS Extension)` (Section 7.6.1 on page 511).

8.5.3.4 Example

Users will be most interested in setting the `participant_liveness_lease_duration` and `participant_liveness_assert_period` values for their *DomainParticipants*. Basically, the lease duration governs how fast an application realizes another application dies unexpectedly. The shorter the periods, the quicker a *DomainParticipant* can determine that a remote participant is dead and act accordingly by declaring all of the remote *DataWriters* and *DataReaders* of that participant dead as well.

However, you should realize that the shorter the period the more liveness packets will be sent by the *DomainParticipant*. How many packets is also determined by the number of peers in the peer list of the participant—whether or not the peers on the list are actually alive.

¹*RTI Enterprise Discovery Service* is an optional package that provides participant-matching services for Connex DDS applications.

8.5.3.5 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.3.6 Related QosPolicies

- [DISCOVERY QosPolicy \(DDS Extension\)](#) (Section 8.5.2 on page 580)
- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy \(DDS Extension\)](#) (Section 8.5.4 below)
- [WIRE_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 8.5.9 on page 610)
- [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347)
- [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 7.6.1 on page 511)
- [DATA_READER_RESOURCE_LIMITS QosPolicy \(DDS Extension\)](#) (Section 7.6.2 on page 517)

8.5.3.7 Applicable Dds Entities

- [DomainParticipants](#) (Section 8.3 on page 547)

8.5.3.8 System Resource Considerations

Setting smaller values for time periods can increase the CPU and network bandwidth usage. Setting larger values for maximum limits can increase the maximum memory that Connex DDS may allocate for a *DomainParticipant* while increasing the initial values will increase the initial memory allocated for a *DomainParticipant*.

8.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)

The `DOMAIN_PARTICIPANT_RESOURCE_LIMITS` QosPolicy includes various settings that configure how *DomainParticipants* allocate and use physical memory for internal resources, including the maximum sizes of various properties.

This QosPolicy sets maximum size limits on variable-length parameters used by the participant and its contained *Entities*. It also controls the initial and maximum sizes of data structures used by the participant to store information about locally-created and remotely-discovered entities (such as *DataWriters/DataReaders*), as well as parameters used by the internal database to size the hash tables used by the data structures.

By default, a *DomainParticipant* is allowed to dynamically allocate memory as needed as users create local *Entities* such as *DataWriters* and *DataReaders* or as the participant discovers new applications to store their information. By setting fixed values for the maximum parameters in this QosPolicy, you can

bound the memory that can be allocated by a *DomainParticipant*. In addition, by setting the initial values to the maximum values, you can prevent *DomainParticipants* from allocating memory after the initialization period.

The maximum sizes of several variable-length parameters—such as the number of partitions that can be stored in the [PARTITION QoSPolicy](#) (Section 6.4.5 on page 323), the maximum length of data store in the [USER_DATA QoSPolicy](#) (Section 6.5.26 on page 417) and [GROUP_DATA QoSPolicy](#) (Section 6.4.4 on page 320), and many others—can be changed from their defaults using this QoS. However, it is important that all *DomainParticipants* that need to communicate with each other use the same set of maximum values. Otherwise, when these parameters are propagated from one *DomainParticipant* to another, a *DomainParticipant* with a smaller maximum length may reject the parameter resulting in an error.

This QoSPolicy includes the members in [Table 8.12 DDS_DomainParticipantResourceLimitsQoSPolicy](#) . For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.12 DDS_DomainParticipantResourceLimitsQoSPolicy

Type	Field Name	Description
DDS_AllocationSettings_t (see description column)	local_writer_allocation	Each allocation structure configures how many objects of each type, <object>_allocation, will be allocated by the <i>DomainParticipant</i> . See Configuring Resource Limits for Asynchronous DataWriters (Section 8.5.4.1 on page 600). <pre> DDS_AllocationSettings_t { DDS_Long initial_count; DDS_Long max_count; DDS_Long incremental_count; }; </pre>
See above row	local_reader_allocation	See above row
See above row	local_publisher_allocation	See above row
See above row	local_subscriber_allocation	See above row

Table 8.12 DDS_DomainParticipantResourceLimitsQosPolicy

Type	Field Name	Description
See above row	local_topic_allocation	See above row
See above row	remote_writer_allocation	See above row
See above row	remote_reader_allocation	See above row
See above row	remote_participant_allocation	See above row
See above row	matching_writer_reader_pair_allocation	See above row
See above row	matching_reader_writer_pair_allocation	See above row
See above row	ignored_entity_allocation	See above row
See above row	content_filtered_topic_allocation	See above row
See above row	content_filter_allocation	See above row
See above row	read_condition_allocation	See above row
See above row	query_condition_allocation	See above row
See above row	outstanding_asynchronous_sample_allocation	See above row

Table 8.12 DDS_DomainParticipantResourceLimitsQosPolicy

Type	Field Name	Description
See above row	flow_controller_allocation	See above row
DDS_DomainParticipantResourceLimitsIgnoredEntityReplacementKind	ignored_entity_replacement_kind	Sets the kinds of entities allowed to be replaced when a <i>DomainParticipant</i> reaches ignored_entity_allocation.max_count . See Resource Limits Considerations for Ignored Entities (Section 16.4.4 on page 788) .
DDS_Long	local_writer_hash_buckets	Used to configure the hash tables used for database searches. If these numbers are too large then memory is wasted. If these number are too small, searching for an object will be less efficient.
DDS_Long	local_reader_hash_buckets	See above row
DDS_Long	local_publisher_hash_buckets	See above row
DDS_Long	local_subscriber_hash_buckets	See above row
DDS_Long	local_topic_hash_buckets	See above row
DDS_Long	remote_writer_hash_buckets	See above row
DDS_Long	remote_reader_hash_buckets	See above row
DDS_Long	remote_participant_hash_buckets	See above row
DDS_Long	matching_writer_reader_pair_hash_buckets	See above row
DDS_Long	matching_reader_writer_pair_hash_buckets	See above row

Table 8.12 DDS_DomainParticipantResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	ignored_entity_hash_buckets	See above row
DDS_Long	content_filtered_topic_hash_buckets	See above row
DDS_Long	content_filter_hash_buckets	See above row
DDS_Long	flow_controller_hash_buckets	See above row
DDS_Long	max_gather_destinations	Configures the maximum number of destinations that a message can be addressed in a single network send operation. Can improve efficiency if the underlying transport support can send to multiple destinations.
DDS_Long	participant_user_data_max_length	Controls the maximum lengths of USER_DATA QosPolicy (Section 6.5.26 on page 417) , TOPIC_DATA QosPolicy (Section 5.2.1 on page 209) and GROUP_DATA QosPolicy (Section 6.4.4 on page 320) for different entities. Must be configured to be the same values on all <i>DomainParticipants</i> in the same DDS domain.
DDS_Long	topic_data_max_length	See above row
DDS_Long	publisher_group_data_max_length	See above row
DDS_Long	subscriber_group_data_max_length	See above row
DDS_Long	writer_user_data_max_length	See above row
DDS_Long	reader_user_data_max_length	See above row
DDS_Long	max_partitions	Controls the maximum number of partitions that can be assigned to a Publisher or Subscriber with the PARTITION QosPolicy (Section 6.4.5 on page 323) . Must be configured to be the same value on all <i>DomainParticipants</i> in the same DDS domain.

Table 8.12 DDS_DomainParticipantResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	max_partition_cumulative_characters	Controls the maximum number of combined characters among all partition names in the PARTITION QosPolicy (Section 6.4.5 on page 323) . Must be configured to be the same value on all <i>DomainParticipants</i> in the same DDS domain.
DDS_Long	type_code_max_serialized_length	Maximum size of serialized string for type code. If your data type has an especially complex type code, you may need to increase this value. See Using Generated Types without Connex DDS (Standalone) (Section 3.7 on page 139) .
DDS_Long	type_object_max_serialized_length	Maximum length, in bytes, that the buffer to serialize TypeObject can consume. This parameter limits the size of the TypeObject that a <i>DomainParticipant</i> is able to propagate. Since TypeObjects contain all of the information of a data structure, including the strings that define the names of the members of a structure, complex data-structures can result in TypeObjects larger than the default maximum. This field allows you to specify a larger value. Cannot be unlimited.
DDS_Long	type_object_max_deserialized_length	Maximum number of bytes that a deserialized TypeObject can consume. This parameter limits the size of the TypeObject that a <i>DomainParticipant</i> is able to store.
DDS_Long	deserialized_type_object_dynamic_allocation_threshold	Threshold, in bytes, for dynamic memory allocation for the deserialized TypeObject. Above it, the memory for a TypeObject is allocated dynamically. Below it, the memory is obtained from a pool of fixed-size buffers. The size of the buffers is equal to this threshold.
DDS_Long	contentfilter_property_max_length	Maximum length of all data related to ContentFilteredTopics (Section 5.4 on page 212) .
DDS_Long	channel_seq_max_length	Maximum number of channels that can be specified in a <i>DataWriter's</i> MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.14 on page 386) .
DDS_Long	channel_filter_expression_max_length	Maximum length of a channel filter_expression in a <i>DataWriter's</i> MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.14 on page 386) .
DDS_Long	participant_property_list_max_length	Maximum number of properties ((name, value) pairs) that can be stored in the <i>DomainParticipant's</i> PROPERTY QosPolicy (DDS Extension) (Section 6.5.17 on page 394) .
DDS_Long	participant_property_string_max_length	Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a <i>DomainParticipant's</i> Property QosPolicy.

Table 8.12 DDS_DomainParticipantResourceLimitsQosPolicy

Type	Field Name	Description
DDS_Long	writer_property_list_max_length	Maximum number of properties ((name, value) pairs) that can be stored in a <i>DataWriter</i> 's Property QosPolicy.
DDS_Long	writer_property_string_max_length	Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a <i>DataWriter</i> 's Property QosPolicy.
DDS_Long	reader_property_list_max_length	Maximum number of properties ((name, value) pairs) that can be stored in a <i>DataReader</i> 's Property QosPolicy.
DDS_Long	reader_property_string_max_length	Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a <i>DataReader</i> 's Property QosPolicy.
DDS_Long	max_endpoint_groups	Maximum number of endpoint groups allowed in an DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1 on page 511) .
	max_endpoint_group_cumulative_characters	Maximum number of combined role_name characters allowed in all endpoint groups in an AvailabilityQosPolicy. The maximum number of combined characters should account for a terminating NULL (") character for each role_name string.
DDS_Long	transport_info_list_max_length	<p>When sending <i>DomainParticipant</i> discovery information, this value defines the maximum number of transports whose properties will be announced to other <i>DomainParticipants</i>.</p> <p>If a <i>DomainParticipant</i> has three transports installed and this value is two, the <i>DomainParticipant</i> will only announce information about the first two transports.</p> <p>When receiving <i>DomainParticipant</i> information, this value defines the maximum size of the list containing information about the transports installed in a remote <i>DomainParticipant</i>.</p> <p>The information about the transports installed in a <i>DomainParticipant</i> is made available to remote <i>DomainParticipants</i> through the sequence field <code>transport_info</code> in the Participant Built-in Topic's Data (see Table 16.1 Participant Built-in Topic's Data Type (DDS_ParticipantBuiltinTopicData))</p> <p>Setting this value to 0 disables the capability of Connex DDS to detect and report transport misconfigurations. However, it does not affect the capability of reaching a given <i>DomainParticipant</i> in all transports available on that <i>DomainParticipant</i>.</p>

Most of the parameters for this QosPolicy are described in the Description column of the table. However, you may need to refer to the sections listed in the column to fully understand the context in which the parameter is used.

An important parameter in this QosPolicy that is often changed by users is the **type_code_max_serialized_length**. This parameter limits the size of the type code that a *DomainParticipant* is able to store and propagate for user data types. Type codes can be used by external applications to understand user data types without having the data type predefined in compiled form. However, since type codes contain all of the information of a data structure including the strings that define the names of the members of a structure, complex data structures can result in type codes larger than the default maximum of 2048 bytes. Thus it is common for users to set this parameter to a larger value. However, as with all parameters in this QosPolicy defining maximum sizes for variable-length elements, all *DomainParticipants* should set the same value for **type_code_max_serialized_length**.

The <object type> **hash_buckets** configure the hash-table data structure that is used to efficiently search the database. The optimal number of buckets depend on the actual number of objects that will be stored in the hash table. So if you know how many *DataWriters* will be created in a *DomainParticipant*, you may change the value of **local_writer_hash_buckets** to balance memory usage against search efficiency. A smaller value will use up less memory, but a larger value will make database lookups for the object more efficient.

If you modify any of the <entity type> **_data_max_length**, **max_partitions**, or **max_partition_cummulative_characters** parameters, then you must make sure that they are modified to be the same value for all *DomainParticipants* in the same DDS domain for all applications. If they are different and an application sends data that is larger than another application is configure to hold, then the two *Entities*, whether a matching *DataWriter/DataReader* pair or even two *DomainParticipants* will fail to connect.

8.5.4.1 Configuring Resource Limits for Asynchronous DataWriters

When using an asynchronous *Publisher*, if a call to **write()** is blocked due to a resource limit, the block will last until the timeout period expires, which will prevent others from freeing the resource. To avoid this situation, make sure that the *DomainParticipant's* **resource_limits.outstanding_asynchronous_sample_allocation** is always greater than the sum of all asynchronous *DataWriters'* **resource_limits.max_samples** (see [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#)).

8.5.4.2 Configuring Memory Allocation

The <object type> **_allocation** configures the number of <object type>'s that can be stored in the internal Connex DDS database. For example, **local_writer_allocation** configures how many local *DataWriters* can be created for the *DomainParticipant*.

The **DDS_AllocationSettings_t** structure sets the initial and maximum number of each object type that can be stored. Memory is allocated for the storage of the objects, thus **initial_count** will determine how much memory is initially allocated, and **max_count** will determine the maximum amount of memory that Connex DDS is allowed to allocate. The **incremental_count** is used to allocate more memory in chunks when the number of objects created exceed the **initial_count**.

You should modify these parameters only if you want to decrease the initial memory used by Connex DDS when a *DomainParticipant* is created or increase the maximum number of local and remote *Entities* that can be stored in a *DomainParticipant*.

How Connex DDS is allowed to allocate memory for a *DomainParticipant* after initialization depends on how you set these parameters.

1. Static memory allocation

No memory is allocated by Connex DDS after creation. Set **initial_count** = **max_count**. The **incremental_count** should be set to 0.

- Advantage: All memory allocation is done when creating the *DomainParticipant*; no dynamic allocation during run-time. You know immediately if you have enough memory to run in that configuration.
- Disadvantage: Requires a fairly static system and/or good estimates on the number of *Entities* in the distributed system. Connex DDS will fail to execute properly once the number of *Entities* exceed the configure bounds.

2. Dynamic, bounded allocation

Set **initial_count** to configure the initial amount of memory to be allocated. Set **max_count** to the maximum allowable upper bound (see the API Reference HTML documentation).

- Advantage: Initial memory usage may be lower and memory is allocated as needed and only if needed.
- Disadvantage: Connex DDS may allocate memory dynamically which may have an impact on performance.

If you allow Connex DDS to allocate memory dynamically, you can either:

- Use fixed-size increments (set `incremental_count` to the desired fixed size).
 - Advantage: well known amount of memory allocated each time.
 - Disadvantage: may require more frequent allocations.
- Double the amount of extra memory allocated each time memory is needed (set `incremental_count` to -1).
 - Advantage: requires fewer allocations.
 - Disadvantage: may allocate considerably more memory than is really needed.

8.5.4.3 Example

For most applications, the default values for this QosPolicy may be sufficient. However, if an application uses the PARTITION, USER_DATA, TOPIC_DATA, or GROUP_DATA QosPolicies, the default maximum sizes of the data associated with those policies may need to be adjusted as required by the

application. As noted previously, you must make sure that all *DomainParticipants* in the same DDS domain use the same sets of values or it is possible that Connex DDS will not successfully connect two *Entities*.

8.5.4.4 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.4.5 Related QosPolicies

- [DATABASE QosPolicy \(DDS Extension\) \(Section 8.5.1 on page 577\)](#)
- [DISCOVERY_CONFIG QosPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#)
- [MULTI_CHANNEL QosPolicy \(DDS Extension\) \(Section 6.5.14 on page 386\)](#)
- [USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)
- [TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#)
- [GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#)
- [PARTITION QosPolicy \(Section 6.4.5 on page 323\)](#)
- [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#)

8.5.4.6 Applicable DDS Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.4.7 System Resource Considerations

Memory and CPU usage are directly affected by the values set for parameters of this QosPolicy. See the detailed descriptions above for specifics.

8.5.5 EVENT QosPolicy (DDS Extension)

The EVENT QosPolicy configures the internal Connex DDS Event thread.

This QoS allows the you to configure thread properties such as priority level and stack size. You can also configure the maximum number of events that can be posted to the event thread. It contains the members in [Table 8.13 DDS_EventQoSPolicy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.13 DDS_EventQoSPolicy

Type	Field Name	Description
DDS_ThreadSettings_t	thread.mask thread.priority thread.stack_size	Thread settings for the event thread used by Connnext DDS to wake up for a timed event and possibly execute listener callbacks. The values used for these settings are OS-dependent; see the RTI Connnext DDS Core Libraries Platform Notes for details. Note: thread.cpu_list and thread.cpu_rotation are not relevant in this QoS policy.
DDS_Long	initial_count	Initial number of events that can be stored simultaneously.
DDS_Long	max_count	Maximum number of events that can be stored simultaneously.

The Event thread is used to wake up and execute timed events posted to the event queue. In a *DomainParticipant*, different Entities may have constraints that have to be checked at periodic intervals or at specific times. If the constraint is violated, a callback function may need to be executed. Timed events include checking for timeouts and deadlines, and executing internal and user timeout or exception handling routines/callbacks. A combination of a time, constraint, and callback can be considered to be an event. For more information, see [Event Thread \(Section 19.2 on page 838\)](#).

For example, a *DataReader* may have a constraint that requires data to be received within a period of time specified by the [DEADLINE QoS Policy \(Section 6.5.5 on page 363\)](#). For that *DataReader*, an event is stored by the Event thread so that it will wake up periodically to check to see if data has arrived in time. If not, the Event thread will execute the `on_requested_deadline_missed()` *Listener* callback of the *DataReader* (if it was installed and enabled).

A reliable connection between a *DataWriter* and *DataReader* will also post events for sending heartbeats used in the reliable protocol discussed in [Reliable Communications \(Section Chapter 10 on page 629\)](#).

This QoS configures the parameters associated with thread creation as well as the number of events that can be simultaneously stored by the Event thread.

8.5.5.1 Example

In a real-time operating system, the priority of the Event thread should be set relative to the priority of the events that it must handle. For example, you may want the Event thread to have a high priority if the deadlines and callbacks that it handles are time or safety critical. It may be critical that the data of a particular *DataReader* arrives on time or if not, alternative action is taken with minimal latency.

If you create many *Entities* in a *DomainParticipant* with QoS Policies that will post events that check deadlines, liveness or send heartbeats, then you may need to increase the maximum number of events that can be stored by the Event thread.

If your application is sending a lot of reliable data, you should increase the event thread priority to be higher than the sending thread priority.

8.5.5.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.5.3 Related QosPolicies

- [DATABASE QosPolicy \(DDS Extension\) \(Section 8.5.1 on page 577\)](#)
- [RECEIVER_POOL QosPolicy \(DDS Extension\) \(Section 8.5.6 below\)](#)

8.5.5.4 Applicable DDS Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.5.5 System Resource Considerations

Increasing **initial_count** and **max_count** will increase initial and maximum memory used for storing events.

Setting the thread parameters correctly on a real-time operating system is usually critical to the proper overall functionality of the applications on that system. Larger values for the thread.**stack_size** parameter will use up more memory.

By default, a *DomainParticipant* will dynamically allocate memory as needed for events posted to the event thread. However, by setting an maximum value or setting the initial and maximum value to be the same, you can either bound the amount of memory allocated for the event thread or prevent a *DomainParticipant* from dynamically allocating memory for the event thread after initialization.

8.5.6 RECEIVER_POOL QosPolicy (DDS Extension)

The RECEIVER_POOL QosPolicy configures the internal Connex DDS thread used to process the data received from a transport. The Receive thread is described in detail in [Receive Threads \(Section 19.3 on page 839\)](#).

This QosPolicy contains the members in [Table 8.14 DDS_ReceiverPoolQoSPolicy](#).

Table 8.14 DDS_ReceiverPoolQoSPolicy

Type	Field Name	Description
struct DDS_ThreadSettings_t	thread.mask thread.priority thread.stack_size hread.cpu_list thread.cpu_rotation	Thread settings for the receive thread(s) used by Connex DDS to process data received from a transport. The values used for these settings are OS-dependent; see the RTI Connex DDS Core Libraries Platform Notes for details. See also: Controlling CPU Core Affinity for RTI Threads (Section 19.5 on page 842) .
DDS_Long	buffer_size	Size of the receive buffer in bytes. For the default and valid range, see the API Reference HTML documentation. buffer_size must always be at least as large as the maximum message_size_max of any installed non-zero-copy transport. ¹ The buffer_size can be adjusted automatically by the middleware by configuring its value to DDS_LENGTH_AUTO (in C/C++) or ReceiverPoolQoSPolicy.LENGTH_AUTO (in .NET and Java). When set to this AUTO default value, the effective value will automatically be set to the largest message_size_max of all installed transports, without needing any other configuration. Therefore you should not need to change this value.
DDS_Long	buffer_alignment	Byte-alignment of the receive buffer. For the default and valid range, see the API Reference HTML documentation.

This QosPolicy sets the thread properties, like priority level and stack size, for the threads used to receive and process data from transports. Connex DDS uses a separate receive thread per port per transport plugin. To force Connex DDS to use a separate thread to process the data for a *DataReader*, you should set a unique port for the [TRANSPORT_UNICAST QosPolicy \(DDS Extension\) \(Section 6.5.24 on page 412\)](#) or [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#) for the *DataReader*.

Connex DDS creates at least one thread for every transport that is installed and enabled for use by the *DomainParticipant* for receiving data. These threads are used to process data DDS samples received for the participant's *DataReaders*, as well as messages used by Connex DDS itself in support of the application discovery process discussed in [Discovery \(Section Chapter 14 on page 709\)](#).

The user application may configure Connex DDS to create many more threads for receiving data sent via multicast or even to dedicate a thread to process the DDS data samples of a single *DataReader* received on a particular transport. This QosPolicy is used in the creation of all receive threads.

¹A “zero-copy transport” does not use the receive buffer. A transport is zero-copy if the `properties_bitmap` property in the `DDS_Transport_Property_t` is `NDDS_TRANSPORT_PROPERTY_BIT_BUFFER_ALWAYS_LOANED`. The only built-in transport that supports zero-copy is the UDPv4 transport on VxWorks platforms.

8.5.6.1 Example

When new data arrives on a transport, the receive thread may invoke the `on_data_available()` of the *Listener* callback of a *DataReader*. Thus, you may want to adjust the priority of the receive threads with respect to the other threads in the application as appropriate for the proper operation of the system.

8.5.6.2 Properties

This *QosPolicy* cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.6.3 Related QosPolicies

- [DATABASE QosPolicy \(DDS Extension\) \(Section 8.5.1 on page 577\)](#)
- [EVENT QosPolicy \(DDS Extension\) \(Section 8.5.5 on page 602\)](#)

8.5.6.4 Applicable Dds Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.6.5 System Resource Considerations

Increasing the `buffer_size` will increase memory used by a receive thread.

Setting the thread parameters correctly on a real-time operating system is usually critical to the proper overall functionality of the applications on that system. Larger values for the `thread.stack_size` parameter will use up more memory.

8.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension)

Connex DDS comes with three different transport plugins built into the core libraries (for most supported target platforms). These are plugins for UDPv4, shared memory, and UDPv6.

This *QosPolicy* allows you to control which built-in transport plugins are used by a *DomainParticipant*. By default, only the UDPv4 and shared memory plugins are enabled (for most platforms; on some platforms, the shared memory plugin is not available). You can disable one or all of the builtin transports.

In some cases, users will disable the shared memory transport when they do not want applications to use shared memory to communicate when running on the same node.

If one application is configured to use UDPv4 *and* shared memory, while another application is only configured for UDPv4, and these two applications run on the same node, they will not communicate. This is due to an internal optimization which will default to use shared memory

instead of loopback. However if the other peer application does not enable shared memory, there is no common transport; therefore they will not communicate.

It contains the member in [Table 8.15 DDS_TransportBuiltinQosPolicy](#). For the default and valid values, please refer to the API Reference HTML documentation.

Table 8.15 DDS_TransportBuiltinQosPolicy

Type	Field Name	Description
DDS_TransportBuiltinKindMask	mask	A mask with bits that indicate which built-in transports will be installed.

Please see the API Reference HTML documentation (select **Modules, RTI Connex DDS API Reference, Pluggable Transports, Using Transport Plugins and Built-in Transport Plugins**) for more information.

Note: Currently, Connex DDS will only listen for discovery traffic on the first multicast address (element 0) in `multicast_receive_addresses`.

8.5.7.1 Example

See [System Resource Considerations \(Section 8.5.7.5 on the facing page\)](#) for an example of why you may want to use this QosPolicy.

In addition, customers may wish to install and use their own custom transport plugins instead of any of the builtin transports. In that case, this QosPolicy may be used to disable all builtin transports.

8.5.7.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.7.3 Related QosPolicies

- [TRANSPORT_SELECTION QosPolicy \(DDS Extension\) \(Section 6.5.23 on page 411\)](#)
- [TRANSPORT_UNICAST QosPolicy \(DDS Extension\) \(Section 6.5.24 on page 412\)](#)
- [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#)

8.5.7.4 Applicable DDS Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.7.5 System Resource Considerations

You can save memory and other system resources if you disable the built-in transports that your application will not use. For example, if you only run a single application with a single *DomainParticipant* on each machine in your network, then you can disable the shared memory transport since your applications will never use it to send or receive messages.

8.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension)

The multicast address on which a *DataReader* wants to receive its data can be explicitly configured using the [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\)](#) (Section 7.6.5 on page 529). However in systems with many multicast addresses, managing the multicast configuration can become cumbersome. The `TransportMulticastMapping` QosPolicy is designed to make configuration and assignment of the *DataReader's* multicast addresses more manageable. When using this QosPolicy, the middleware will automatically assign a multicast receive address for a *DataReader* from a range by using configurable mapping rules.

DataReaders can be assigned a single multicast receive address using the rules defined in this QosPolicy on the *DomainParticipant*. This multicast receive address is exchanged during simple discovery in the same manner used when the multicast receive address is defined explicitly. No additional configuration on the writer side is needed.

Mapping within a range is done through a mapping function. The middleware provides a default hash (md5) mapping function. This interface is also pluggable, so you can specify a custom mapping function to minimize collisions.

To use this QosPolicy, you must set the **kind** in the [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\)](#) (Section 7.6.5 on page 529) to AUTOMATIC.

This QosPolicy contains the member in [Table 8.16 DDS_TransportMulticastMappingQosPolicy](#).

Table 8.16 DDS_TransportMulticastMappingQosPolicy

Type	Field Name	Description
DDS_TransportMappingSettingsSeq	value	A sequence of multicast communication settings, each of which has the format shown in Table 8.17 DDS_TransportMulticastSettings_t .

Table 8.17 DDS_TransportMulticastSettings_t

Type	Field Name	Description
char *	addresses	A string containing a comma-separated list of IP addresses or IP address ranges to be used to receive multicast traffic for the entity with a topic that matches the topic_expression . See Formatting Rules for Addresses (Section 8.5.8.1 below) .
char *	topic_expression	A regular expression used to map topic names to corresponding addresses. See SQL Extension: Regular Expression Matching (Section 5.4.6.5 on page 228) .
DDS_TransportMulticastMappingFunction_t	mapping_function	<i>Optional.</i> Defines a user-provided pluggable mapping function. See Table 8.18 DDS_TransportMulticastMappingFunction_t .

Table 8.18 DDS_TransportMulticastMappingFunction_t

Type	Field Name	Description
char *	dll	Specifies a dynamic library that contains a mapping function. You may specify a relative or absolute path. If the name is specified as "foo", the library name on Linux systems will be libfoo.so ; on Windows systems it will be foo.dll .
char *	function_name	Specifies the name of a mapping function in the library specified in the above dll . The function must implement the following interface: <pre>int function(const char* topic_name, int numberOfAddresses);</pre> The function must return an integer that indicates the <i>index</i> of the address to use for the given topic_name . For example, if the first address in the list should be used, it must return 0; if the second address in the list should be used, it must return 1, etc.

8.5.8.1 Formatting Rules for Addresses

- The string must contain IPv4 or IPv6 addresses separated by commas. For example:
"239.255.100.1,239.255.100.2,239.255.100.3"
- You may specify ranges of addresses by enclosing the start and end addresses in square brackets. For example: "[239.255.100.1,239.255.100.3]".
- You may combine the two approaches.
For example: "239.255.200.1,[239.255.100.1,239.255.100.3], 239.255.200.3"

- IPv4 addresses must be specified in Dot-decimal notation.
- IPv6 addresses must be specified using 8 groups of 16-bit hexadecimal values separated by colons. For example: FF00:0000:0000:0000:0202:B3FF:FE1E:8329.
- Leading zeroes can be skipped. For example: FF00:0:0:0:202:B3FF:FE1E:8329.
- You may replace a consecutive number of zeroes with a double colon, but only once within an address. For example: FF00::202:B3FF:FE1E:8329.

8.5.8.2 Example

This QoS policy configures the multicast ranges and mapping rules at the *DomainParticipant* level. You can configure a large set of multicast addresses on the *DomainParticipant*.

In addition, you can configure a mapping between topic names and multicast addresses. For example, topic "A" can be assigned to address 239.255.1.1 and topic "B" can be assigned to address 239.255.1.2.

This configuration is quite flexible. For example, you can specify mappings between a subset of topics to a range of multicast addresses. For example, topics "X", "Y" and "Z" can be mapped to [239.255.1.1, 239.255.1.255], or using regular expressions, "X*" and "B-Z" can be mapped to a sub-range of addresses. See [SQL Extension: Regular Expression Matching \(Section 5.4.6.5 on page 228\)](#).

8.5.8.3 Properties

This QoSPolicy cannot be modified after the *DomainParticipant* is created.

8.5.8.4 Related QoS Policies

- [TRANSPORT_MULTICAST QoS Policy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#)

8.5.8.5 Applicable DDS Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.8.6 System Resource Considerations

See [System Resource Considerations \(Section 7.6.5.5 on page 532\)](#).

8.5.9 WIRE_PROTOCOL QoS Policy (DDS Extension)

The WIRE_PROTOCOL QoS Policy configures some global Real-Time Publish Subscribe (RTPS) protocol-related properties for the *DomainParticipant*. The RTPS OMG-standard, interoperability protocol is used by Connex DDS to format and interpret messages between *DomainParticipants*.

It includes the members in [Table 8.19 DDS_WireProtocolQoS Policy](#). For defaults and valid ranges, please refer to the API Reference HTML documentation. (The default values contain the correctly initialized wire

protocol attributes. They should not be modified without an understanding of the underlying Real-Time Publish Subscribe (RTPS) wire protocol.)

Table 8.19 DDS_WireProtocolQosPolicy

Type	Field Name	Description
DDS_Long	participant_id	Unique identifier for participants that belong to the same DDS domain on the same host. See Choosing Participant IDs (Section 8.5.9.1 below) .
DDS_UnsignedLong	rtps_host_id	A machine/OS-specific host ID, unique in the DDS domain. See Host, App, and Instance IDs (Section 8.5.9.2 on page 613) .
	rtps_app_id	A participant-specific ID, unique within the scope of the rtps_host_id. See Host, App, and Instance IDs (Section 8.5.9.2 on page 613) .
	rtps_instance_id	An instance-specific ID of the <i>DomainParticipant</i> that, together with the rtps_app_id, is unique within the scope of the rtps_host_id. See Host, App, and Instance IDs (Section 8.5.9.2 on page 613) .
DDS_RtpsWellKnownPorts_t	rtps_well_known_ports	Determines the well-known multicast and unicast ports for discovery and user traffic. See Ports Used for Discovery (Section 8.5.9.3 on page 613) .
DDS_RtpsReservedPortKindMask	rtps_reserved_ports_mask	Specifies which well-known multicast and unicast ports to reserve when enabling the <i>DomainParticipant</i> .
DDS_WireProtocolQosPolicyAutoKind	rtps_auto_id_kind	Kind of auto mechanism used to calculate the GUID prefix.

Note that [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#) and [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#) configure RTPS and reliability properties on a per *DataWriter* and *DataReader* basis.

8.5.9.1 Choosing Participant IDs

When you create a *DomainParticipant*, you must specify a domain ID, which identifies the communication channel across the whole system. Each *DomainParticipant* in the same DDS domain on the same host also needs a unique integer, known as the **participant_id**.

The **participant_id** uniquely identifies a *DomainParticipant* from other *DomainParticipants* in the same DDS domain on the same host. You can use the same **participant_id** value for *DomainParticipants* in the same DDS domain but running on different hosts.

The **participant_id** is also used to calculate the default unicast user-traffic and the unicast meta-traffic port numbers, as described in [Ports Used for Discovery \(Section 14.5 on page 738\)](#). If you only have one *DomainParticipant* in the same DDS domain on the same host, you will not need to modify this value.

You can either allow Connex DDS to select a participant ID automatically (by setting **participant_id** to -1), or choose a specific participant ID (by setting **participant_id** to the desired value).

- **Automatic Participant ID Selection**

The default value of `participant_id` is -1, which means Connex DDS will select a participant ID for you.

Connex DDS will pick the smallest participant ID, based on the unicast ports available on the transports enabled for discovery, based on the unicast and/or multicast ports available on the transports enabled for discovery and/or user traffic.

The **rtps_reserved_ports_mask** field determines which ports to check when picking the next available participant ID. The reserved ports are calculated based on the formula specified in [Inbound Ports for Meta-Traffic \(Section 14.5.1 on page 739\)](#) and [Inbound Ports for User Traffic \(Section 14.5.2 on page 740\)](#). By default, Connex DDS will reserve the meta-traffic unicast port, the meta-traffic multicast port, and the user traffic unicast port.

Connex DDS will attempt to resolve an automatic port ID either when a *DomainParticipant* is enabled, or when a *DataReader* or a *DataWriter* is created. Therefore, all the transports enabled for discovery must have been registered by this time. Otherwise, the discovery transports registered after resolving the automatic port index may produce port conflicts when the *DomainParticipant* is enabled.

To see what value Connex DDS has selected, either:

- Change the verbosity level of the `NDDS_CONFIG_LOG_CATEGORY_API` category to `NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL` (see [Controlling Messages from Connex DDS \(Section 21.2 on page 865\)](#)).
- Call `get_qos()` and look at the `participant_id` value in the `WIRE_PROTOCOL QosPolicy (DDS Extension)` ([Section 8.5.9 on page 610](#)) after the *DomainParticipant* is enabled.

- **Manual Participant ID Selection**

If you do have multiple *DomainParticipants* on the same host, you should use consecutively numbered participant indices start from 0. This will make it easier to specify the discovery peers using the `initial_peers` parameter of this QosPolicy or the `NDDS_DISCOVERY_PEERS` environment variable. See [Configuring the Peers List Used in Discovery \(Section 14.2 on page 711\)](#) for more information.

Do not use random participant indices since this would make DISCOVERY incredibly difficult to configure. In addition, the `participant_id` has a maximum value of 120 (and will be less for domain IDs other than 0) when using an IP-based transport since the `participant_id` is used to create the port number (see [Ports Used for Discovery \(Section 14.5 on page 738\)](#)), and for IP, a port number cannot be larger than 65536.

For details, see [Ports Used for Discovery \(Section 14.5 on page 738\)](#).

8.5.9.2 Host, App, and Instance IDs

The `rtps_host_id`, `rtps_app_id`, and `rtps_instance_id` values are used by the RTPS protocol to allow Connex DDS to distinguish messages received from different *DomainParticipants*. Their combined values must be globally unique across all existing *DomainParticipants* in the same DDS domain. In addition, if an application dies unexpectedly and is restarted, the IDs used by the new instance of *DomainParticipants* should be different than the ones used by the previous instances. A change in these values allows other *DomainParticipants* to know that they are communicating with a new instance of an application, and not the previous instance.

If the value of `rtps_host_id` is set to `DDS_RTPS_AUTO_ID`, the IPv4 address of the host is used as the host ID. If the host does not have an IPv4 address, the host-id will be automatically set to `0x7F000001`.

If the value of `rtps_app_id` is set to `DDS_RTPS_AUTO_ID`, the process (or task) ID is used. There can be at most 256 distinct participants in a shared address space (process) with a unique `rtps_app_id`.

If the value of `rtps_instance_id` is set to `DDS_RTPS_AUTO_ID`, a counter is assigned that is incremented per new participant. Thus, together with `rtps_app_id`, there can be at most 2^{64} distinct participants in a shared address space with a unique RTPS Globally Unique Identifier (GUID).

8.5.9.3 Ports Used for Discovery

The `rtps_well_known_ports` structure allows you to configure the ports that are used for discovery of inbound meta-traffic (discovery data internal to Connex DDS) and user traffic (from your application).

It includes the members in [Table 8.20 DDS_RtpsWellKnownPorts_t](#). For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 8.20 DDS_RtpsWellKnownPorts_t

Type	Field Name	Description
DDS_ Long	port_base	The base port offset. All mapped well-known ports are offset by this value. Resulting ports must be within the range imposed by the underlying transport.
	domain_id_gain	Tunable gain parameters. See Ports Used for Discovery (Section 14.5 on page 738) .
	participant_id_gain	
	builtin_multicast_port_offset	Additional offset for meta-traffic port. See Inbound Ports for Meta-Traffic (Section 14.5.1 on page 739) .
	builtin_unicast_port_offset	
	user_multicast_port_offset	Additional offset for user traffic port. See Inbound Ports for User Traffic (Section 14.5.2 on page 740) .
	user_unicast_port_offset	

8.5.9.4 Controlling How the GUID is Set (rtps_auto_id_kind)

In order for the discovery process to work correctly, each *DomainParticipant* must have a unique identifier. This QoS policy specifies how that identifier should be generated.

RTPS defines a 96-bit prefix to this identifier; each *DomainParticipant* must have a unique value of this prefix relative to all other participants in its DDS domain. In order to make it easier to control how this 96-bit value is generated, Connex DDS divides it into three integers: a host ID, the value of which is based on the identity of the machine on which the participant is executing, an application ID (whose value is based on the process or task in which the participant is contained), and an instance ID which identifies the participant itself.

This QoS policy provides you with a choice of algorithms for generating these values automatically. In case none of these algorithms suit your needs, you may also choose to specify some or all of them yourself.

The following three fields compose the GUID prefix and by default are set to `DDS_RTSPS_AUTO_ID`. The meaning of this flag depends on the value assigned to `rtps_auto_id_kind`.

- `rtps_host_id`
- `rtps_app_id`
- `rtps_instance_id`

Depending on the **rtps_auto_id_kind** value, there are three different scenarios:

1. In the default and most common scenario, **rtps_auto_id_kind** is set to `DDS_RTPTS_AUTO_ID_FROM_IP`. Doing so, each field is interpreted as follows:
 - **rtps_host_id**: the 32 bit value of the IPv4 of the first up and running interface of the host machine is assigned
 - **rtps_app_id**: the process (or task) ID is assigned
 - **rtps_instance_id**: A counter is assigned that is incremented per new participant

Note: If the IP address assigned to the interface is not unique within the network (for instance, if it is not configured), then is it possible that the GUID (specifically, the **rtps_host_id** portion) may also not be unique.

2. In this scenario, Connexx DDS **rtps_auto_id_kind**: is set to `DDS_RTPTS_AUTO_ID_FROM_MAC`. As the name suggests, this alternative mechanism uses the MAC address instead of the IPv4 address. Since the MAC address size is up to 64 bits, the logical mapping of the host information, the application ID, and the instance identifiers has to change.

Note to Solaris Users: To use `DDS_RTPTS_AUTO_ID_FROM_MAC`, you must run the Connexx DDS application while logged in as 'root.'

Using `DDS_RTPTS_AUTO_ID_FROM_MAC`, the default value of each field is interpreted as follows:

- **rtps_host_id**: the first 32 bits of the MAC address of the first up and running interface of the host machine are assigned
- **rtps_app_id**: the last 32 bits of the MAC address of the first up and running interface of the host machine are assigned
- **rtps_instance_id**: this field is split into two different parts. The process (or task) ID is assigned to the first 24 bits. A counter is assigned to the last 8 bits. This counter is incremented per new participant. In both scenarios, you can change the value of each field independently.

If `DDS_RTPTS_AUTO_ID_FROM_MAC` is used, the **rtps_instance_id** has been logically split into two parts: 24 bits for the process/task ID and 8 bits for the per new participant counter. To give to users the ability to manually set the two parts independently, a bit field mechanism has been introduced for the **rtps_instance_id** field when it is used in combination with `DDS_RTPTS_AUTO_ID_FROM_MAC`. If one of the two parts is set to 0, only this part will be handled by Connexx DDS and you will be able to handle the other one manually.

3. In this scenario, **rtps_auto_id_kind** is set to `RTPTS_AUTO_ID_FROM_UUID`. As the name suggests, this alternative mechanism uses a unique, randomly generated UUID to fill the **rtps_host_id**,

rtps_app_id, or **rtps_instance_id** fields.

Note: `RTPS_AUTO_ID_FROM_UUID` is only supported on iOS architectures.

Some examples are provided to better explain the behavior of this QoS Policy in case you want to change the default behavior with `DDS_RTPS_AUTO_ID_FROM_MAC`.

1. Get the *DomainParticipant* QoS from the `DomainParticipantFactory`:

```
DDS_DomainParticipantFactory_get_default_participant_qos(  
    DDS_DomainParticipantFactory_get_instance(),  
    &participant_qos);
```

2. Change the `WireProtocolQoSPolicy` using one of the following options.

- Use `DDS_RTTPS_AUTO_ID_FROM_MAC` to explicitly set just the application/task identifier portion of the `rtps_instance_id` field:

```
participant_qos.wire_protocol.rtps_auto_id_kind =
    DDS_RTTPS_AUTO_ID_FROM_MAC;
participant_qos.wire_protocol.rtps_host_id =
    DDS_RTTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_app_id =
    DDS_RTTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_instance_id =
    (/* App ID */ (12 << 8) |
     /* Instance ID*/ (DDS_RTTPS_AUTO_ID));
```

- Only set the per participant counter and let *Connex*t DDS handle the application/task identifier:

```
participant_qos.wire_protocol.rtps_auto_id_kind =
    DDS_RTTPS_AUTO_ID_FROM_MAC;
participant_qos.wire_protocol.rtps_host_id =
    DDS_RTTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_app_id =
    DDS_RTTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_instance_id =
    (/* App ID */ (DDS_RTTPS_AUTO_ID) |
     /* Instance ID*/ (12));
```

- Set the entire `rtps_instance_id` field yourself:

```
participant_qos.wire_protocol.rtps_auto_id_kind =
    DDS_RTTPS_AUTO_ID_FROM_MAC;
participant_qos.wire_protocol.rtps_host_id =
    DDS_RTTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_app_id =
    DDS_RTTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_instance_id =
    ( /* App ID */ (12 << 8) |
     /* Instance ID */ (9) )
```

Note: If you are using `DDS_RTTPS_AUTO_ID_FROM_MAC` as `rtps_auto_id_kind` and you decide to manually handle the `rtps_instance_id` field, you must ensure that both parts are non-zero (otherwise Connex

DDS will take responsibility for them). RTI recommends that you always specify the two parts separately in order to avoid errors.

- Let *Connex*t DDS handle the entire `rtps_instance_id` field:

```

participant_qos.wire_protocol.rtps_auto_id_kind =
    DDS_RTPS_AUTO_ID_FROM_MAC;
participant_qos.wire_protocol.rtps_host_id =
    DDS_RTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_app_id =
    DDS_RTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_instance_id =
    DDS_RTPS_AUTO_ID;

```

Note: If you are using `DDS_RTPS_AUTO_ID_FROM_MAC` as `rtps_auto_id_kind` and you decide to manually set the `rtps_instance_id` field, you must ensure that both parts are non-zero (otherwise Connex DDS will take responsibility for them). RTI recommends that you always specify the two parts separately in order to clearly show the difference.

3. Create the *DomainParticipant* as usual using the modified QoS structure instead of the default one.

8.5.9.5 Example

On many real-time operating systems, and even on some non-real-time operating systems, when a node is rebooted, and applications are automatically started, process ids are deterministically assigned. That is, when the system restarts or if an application dies and is restarted, the application will be reassigned the same process or task ID.

This means that Connex DDS's automatic algorithm for creating unique `rtps_app_id`'s will produce the same value between sequential instances of the same application. This will confuse the other *DomainParticipants* on the network into thinking that they are communicating with the previous instance of the application instead of a new instance. Errors usually resulting in a failure to communicate will ensue.

Thus for applications running on nodes that may be rebooted without letting the application shutdown appropriately (destroying the *DomainParticipant*), especially on nodes running real-time operating systems like VxWorks or LynxOS, you will want to set the `rtps_app_id` manually. We suggest that a strictly incrementing counter is stored either on a file system or in non-volatile RAM is used for the `rtps_app_id`.

Whatever method you use, you should make sure that the `rtps_app_id` is unique across all *DomainParticipants* running on a host as well as *DomainParticipants* that were recently running on the host. After a period configured through the `DISCOVERY_CONFIG` QoSPolicy existing applications will eventually flush old *DomainParticipants* that did not properly shutdown from their databases. When that is done, then `rtps_app_id` may be reused.

8.5.9.6 Properties

This QoSPolicy cannot be modified after the *DomainParticipant* is created.

If manually set, it must be set differently for every *DomainParticipant* in the same DDS domain across all applications. The value of `rtps_app_id` should also change between different invocations of the same application (for example, when an application is restarted).

8.5.9.7 Related QosPolicies

- [DISCOVERY_CONFIG QosPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#)

8.5.9.8 Applicable DDS Entities

- [DomainParticipants \(Section 8.3 on page 547\)](#)

8.5.9.9 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

8.6 Clock Selection

Connex DDS uses clocks to measure time and generate timestamps.

The middleware uses two clocks: an internal clock and an external clock.

- The internal clock measures time and handles all timing in the middleware.
- The external clock is used solely to generate timestamps (such as the source timestamp and the reception timestamp), in addition to providing the time given by the *DomainParticipant's* `get_current_time()` operation (see [Getting the Current Time \(Section 8.3.13.2 on page 571\)](#)).

8.6.1 Available Clocks

Two clock implementations are generally available: the *real-time* clock and the *monotonic* clock.

The real-time clock provides the real time of the system. This clock may generally be monotonic, but may not be guaranteed to be so. It is adjustable and may be subject to small and large changes in time. The time obtained from this clock is generally a meaningful time, in that it is the amount of time from a known epoch. For the purposes of clock selection, this clock can be referenced by the names "**realtime**" or "**system**"—*both names map to the same real-time clock*.

The monotonic clock provides times that are monotonic from a clock that is not adjustable. This clock is not subject to changes in the system or realtime clock, which may be adjusted by the user or via time synchronization protocols. However, this clock's time generally starts from an arbitrary point in time, such as system start-up. Note that the monotonic clock is not available for all architectures. Please see the [RTI Connex DDS Core Libraries Platform Notes](#) for the architectures on which it is supported. For the purposes of clock selection, this clock can be referenced by the name "**monotonic**".

8.6.2 Clock Selection Strategy

To configure the clock selection, use the *DomainParticipant's* [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#). [Table 8.21 Clock Selection Properties](#) lists the supported properties.

Table 8.21 Clock Selection Properties

Property	Description
dds.clock.external_clock	Comma-delimited list of clocks to use for the external clock, in the order of preference. Valid clock names are “realtime”, “system”, or “monotonic”.
dds.clock.internal_clock	Comma-delimited list of clocks to use for the internal clock, in the order of preference. Valid clock names are “realtime”, “system”, or “monotonic”.

By default, both the internal and external clocks use the realtime clock.

If you want your application to be robust to changes in the system time, you may use the monotonic clock as the internal clock, and leave the system clock as the external clock. However, note that this may slightly diminish performance, in that both the send and receive paths may need to get times from both clocks.

Since the monotonic clock is not available on all architectures, you may want to specify "monotonic, realtime" for the **internal_clock** property (see [Table 8.21 Clock Selection Properties](#)). By doing so, the middleware will attempt to use the monotonic clock if it is available, and will fall back to the realtime clock if the monotonic clock is not available.

If you want the application to be robust to changes in the system time, you are not relying on source timestamps, and you want to avoid obtaining times from both clocks, you may use the monotonic clock for both the internal and external clocks.

8.7 System Properties

Connex DDS uses the *DomainParticipant's* PropertyQosPolicy to maintain a set of properties that provide system information, such as the hostname.

Unless the default the DDS_DomainParticipantQos structure (see [Setting DomainParticipant QoS Policies \(Section 8.3.6 on page 562\)](#)) is overwritten, the system properties are automatically set in the DDS_DomainParticipantQos structure that is obtained by calling the DomainParticipantFactory's **get_default_participant_qos()** operation or by using the constant DDS_PARTICIPANT_QOS_DEFAULT.

System properties are also automatically set in the DDS_DomainParticipantQos structure loaded from an XML QoS profile unless you disable property inheritance using the attribute **inherit** in the XML tag <property>.

By default, the system properties are propagated to other *DomainParticipants* in the system and can be accessed through the **property** field in the [Table 16.1 Participant Built-in Topic's Data Type \(DDS_ParticipantBuiltinTopicData\)](#).

You can disable propagation of individual properties by setting the property's **propagate** flag to FALSE or by removing the property using the PropertyQosPolicyHelper operation, **remove_property()** (see [Table 6.57 PropertyQoSHelper Operations](#)).

The number of system properties that are initialized for a *DomainParticipant* is platform specific: only **process_id** and **os_arch** are supported on all platforms.

These properties will only be created if Connex DDS can obtain the information for them; see [Table 8.22 System Properties](#).

System properties are affected by the *DomainParticipantResourceLimitsQosPolicy*'s **participant_property_list_max_length** and **participant_property_string_max_length**.

Table 8.22 System Properties

Property Name	Description
dds.sys_info.creation_timestamp	Time when the executable was created. ¹
dds.sys_info.executable_filepath	Name and full path of the executable. ²
dds.sys_info.execution_timestamp	Time when the execution started. ³
dds.sys_info.hostname	Hostname ⁴
dds.sys_info.target	Architecture for which the library was compiled (for example, x64Darwin10gcc4.2.1).
dds.sys_info.process_id	Process ID
dds.sys_info.username	Username that is running the process. ⁵

¹Only supported on Windows and Linux architectures.

²Only supported on Windows and Linux architectures.

³Only supported on Windows and Linux architectures.

⁴Only supported on Windows and Linux architectures.

⁵Only supported on Windows and Linux architectures.

Chapter 9 Building Applications

This chapter provides instructions on how to build Connex DDS applications for the following platforms:

- [UNIX-Based Platforms \(Section 9.3 on page 624\)](#) (including Solaris™, Red Hat® and Yellow Dog™ Linux, QNX®, and LynxOS® systems)
- [Windows Platforms \(Section 9.4 on page 625\)](#)
- [Java Platforms \(Section 9.5 on page 627\)](#)

While you can create applications for other operating systems, the platforms presented in this chapter are a good starting point. We recommend that you first build and test your application on one of these systems.

Instructions for other supported target platforms are provided in the [RTI Connex DDS Core Libraries Platform Notes](#).

To build a non-Java application using Connex DDS, you must specify the following items:

- NDDSHOME environment variable
- Connex DDS header files
- Connex DDS libraries to link
- Compatible system libraries
- Compiler options

To build Java applications using Connex DDS, you must specify the following items:

- NDDSHOME environment variable
- Connex DDS JAR file

- Compatible Java virtual machine (JVM)
- Compiler options

This chapter describes the basic steps you will take to build an application on the above-mentioned platforms. Specific details, such as exactly which libraries to link, compiler flags, etc. are in the [RTI Connex DDS Core Libraries Platform Notes](#).

9.1 Running on a Computer Not Connected to a Network

If you want to run Connex DDS applications on the same computer, *and* that computer is not connected to a network, you must set `NDDS_DISCOVERY_PEERS` so that it will only use shared memory. For example:

```
set NDDS_DISCOVERY_PEERS=4@shmem://
```

(The number 4 is only an example. This is the maximum participant ID.)

9.2 Connex DDS Header Files – All Architectures

You must include the appropriate Connex DDS header files, which are listed in [Table 9.1 Header Files to Include for Connex DDS \(All Architectures\)](#). The header files that need to be included depend on the API being used.

Table 9.1 Header Files to Include for Connex DDS (All Architectures)

Connex DDS API	Header Files
C	<code>#include "ndds/ndds_c.h"</code>
C++	<code>#include "ndds/ndds_cpp.h"</code>
C++/CLI, C#, Java	none

For the compiler to find the included files, the path to the appropriate include directories must be provided. [Table 9.2 Include Paths for Compilation \(All Architectures\)](#) lists the appropriate include path for use with the compiler. The exact path depends on where you installed Connex DDS. See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#).

Table 9.2 Include Paths for Compilation (All Architectures)

Connex DDS API	Include Path Directories
C and C++	<NDDSHOME>/include <NDDSHOME>/include/ndds
C++/CLI, C#, Java	none

The header files that define the data types you want to use within the application also need to be included. For example, [Table 9.3 Header Files to Include for Data Types \(All Architectures\)](#) lists the files to be include for type “Foo” (these are the filenames generated by *RTI Code Generator*, described in [Data Types and DDS Data Samples \(Section Chapter 3 on page 23\)](#)).

Table 9.3 Header Files to Include for Data Types (All Architectures)

Connex DDS API	User Data Type Header Files
C and C++	#include “Foo.h” #include “FooSupport.h”
C++/CLI, C#, Java	none

9.3 UNIX-Based Platforms

Before building a Connex DDS application for a UNIX-based platform (including Solaris, Red Hat and Yellow Dog Linux, QNX, and LynxOS systems), make sure that:

- A supported version of your architecture is installed. See the [RTI Connex DDS Core Libraries Platform Notes](#) for supported architectures.
- Connex DDS 5.x.y is installed (where 5.x.y stands for the version number of the current release). For installation instructions, refer to the [RTI Connex DDS Core Libraries Getting Started Guide](#).
- A “make” tool is installed. RTI recommends GNU Make. If you do not have it, you may be able to download it from your operating system vendor. Learn more at www.gnu.org/software/make/ or download from ftpmirror.gnu.org/make as source code.
- The **NDDSHOME** environment variable is set to the root directory of the Connex DDS installation (such as `/home/user/rti_connex_dds-5.x.y`).
 - To confirm, type this at a command prompt:

```
echo $NDDSHOME
env | grep NDDSHOME
```

- If it is not set or is set incorrectly, type:

```
setenv NDDSHOME <correct directory>
```

To compile a Connex DDS application of any complexity, either modify the auto-generated makefile created by running *RTI Code Generator* or write your own makefile.

9.3.1 Required Libraries

All required system and Connex DDS libraries are listed in the [RTI Connex DDS Core Libraries Platform Notes](#).

You must choose between dynamic (shared) and static libraries. Do not mix the different types of libraries during linking. The benefit of linking against the dynamic libraries is that your final executables' sizes will be significantly smaller. You will also use less memory when you are running several Connex DDS applications on the same node. However, shared libraries require more set-up and maintenance during upgrades and installations.

To see if dynamic libraries are supported for your target architecture, see the [RTI Connex DDS Core Libraries Platform Notes](#)¹.

9.3.2 Compiler Flags

See the [RTI Connex DDS Core Libraries Platform Notes](#) for information on compiler flags.

9.4 Windows Platforms

Before building an application for a Microsoft Windows® platform, make sure that:

- Supported versions of Windows and Visual Studio are installed. See the [Windows section of the RTI Connex DDS Core Libraries Platform Notes](#).
- Connex DDS 5.x.y is installed (where 5.x.y stands for the version numbers of the current release). For installation instructions, refer to the [RTI Connex DDS Core Libraries Getting Started Guide](#).
- The **NDDSHOME** environment variable is set to the root directory of the Connex DDS installation (such as **C:\Program Files\rti_connex_dds-5.x.y**). To confirm, type this at a command prompt:

```
echo %NDDSHOME%
```

- Use the *dynamic* MFC Library (not static).

¹In the *Platform Notes*, see the “Building Instructions...” table for your target architecture.

To avoid communication problems in your Connex DDS application, use the dynamic MFC library, not the static version. (If you use the static version, your Connex DDS application may stop receiving DDS samples once the Windows sockets are initialized.)

To compile a Connex DDS application of any complexity, use a project file in Microsoft Visual Studio. The project settings are described below. The [Windows section of the RTI Connex DDS Core Libraries Platform Notes](#) contains more information.

9.4.1 Using Visual Studio

1. Select the multi-threaded project setting:
 - a. From the **Project** menu, select **Properties**.
 - b. Select the C/C++ folder.
 - c. Select Code Generation.
 - d. Set the **Runtime Library** field to one of the options from [Table 9.4 Runtime Library Settings for Visual Studio](#).

2. Link against the Connex DDS libraries:
 - a. Select the Linker folder on the Project, Properties dialog box.
 - b. Select the Input properties.
 - c. See the [Windows section of the RTI Connex DDS Core Libraries Platform Notes](#) for a list of required libraries. You have a choice of whether to link with Connex DDS's static or dynamic libraries. Decide whether or not you want debugging symbols on. In either case, be sure to use a *space* as a delimiter between libraries, *not* a comma. Add the libraries to the *beginning* of the Additional Dependencies field.
 - d. Select the **General** properties.
 - e. Add the following to the Additional library path field (replace *<architecture>* to match your installed system):

```
$ (NDDSHOME) \lib\<architecture>
```

3. Specify the path to Connex DDS's header file:
 - a. Select the C/C++ folder.
 - b. Select the **General** properties.
 - c. In the Additional include directories: field, add paths to the "include" and "include\ndds" directories.
For example: (your paths may differ, depending on where you installed Connex DDS)

```
c:\Program Files\rti_connexdds-5.x.y\include\  
c:\Program Files\rti_connexdds-5.x.y\include\ndds
```

Table 9.4 Runtime Library Settings for Visual Studio

If You are using this Library Format...	Set the Runtime Library field to...
Release version of static libraries	Multi-threaded (/MT)
Debug version of static libraries	Multi-threaded Debug (/MTd)
Release version of dynamic libraries	Multi-threaded DLL (/MD)
Debug version of dynamic libraries	Multi-threaded Debug DLL (/MDd)

9.5 Java Platforms

Before building an application for a Windows or UNIX Java platform, make sure that:

- Connex DDS 5.x.y is installed (where 5.x.y stands for the version numbers of the current release).
- A supported version of the Java 2 software development kit (J2SDK) is installed. See the [Windows section of the RTI Connex DDS Core Libraries Platform Notes](#).

9.5.1 Java Libraries

Connex DDS requires that certain Java archive (JAR) files be on your classpath when running Connex DDS applications. See the [Platform Notes](#) for more details.

9.5.2 Native Libraries

Connex DDS for Java is implemented using Java Native Interface (JNI), so it is necessary to provide your Connex DDS distributed applications access to certain native shared libraries. See the [RTI Connex DDS Core Libraries Platform Notes](#) for more details.

Part 3: Advanced Concepts

This part of the manual will guide you through some of the more advanced concepts:

- [Reliable Communications \(Section Chapter 10 on page 629\)](#)
- [Collaborative DataWriters \(Section Chapter 11 on page 670\)](#)
- [Mechanisms for Achieving Information Durability and Persistence \(Section Chapter 12 on page 675\)](#)
- [Guaranteed Delivery of Data \(Section Chapter 13 on page 695\)](#)
- [Discovery \(Section Chapter 14 on page 709\)](#)
- [Transport Plugins \(Section Chapter 15 on page 743\)](#)
- [Built-In Topics \(Section Chapter 16 on page 772\)](#)
- [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#)
- [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#)
- [Connex DDS Threading Model \(Section Chapter 19 on page 837\)](#)
- [DDS Sample-Data and Instance-Data Memory Management \(Section Chapter 20 on page 846\)](#)
- [Troubleshooting \(Section Chapter 21 on page 863\)](#)

Chapter 10 Reliable Communications

Connex DDS uses *best-effort* delivery by default. The other type of delivery that Connex DDS supports is called *reliable*. This chapter provides instructions on how to set up and use reliable communication.

This chapter includes the following sections:

- [Sending Data Reliably \(Section 10.1 below\)](#)
- [Overview of the Reliable Protocol \(Section 10.2 on page 631\)](#)
- [Using QosPolicies to Tune the Reliable Protocol \(Section 10.3 on page 635\)](#)

10.1 Sending Data Reliably

The DCPS reliability model recognizes that the optimal balance between time-determinism and data-delivery reliability varies widely among applications and can vary among different publications within the same application. For example, individual DDS samples of *signal* data can often be dropped because their value disappears when the next DDS sample is sent. However, each DDS sample of *command* data must be received and it must be received in the order sent.

The QosPolicies provide a way to customize the determinism/reliability trade-off on a per *Topic* basis, or even on a per *DataWriter/DataReader* basis.

There are two delivery models:

- Best-effort delivery mode “I’m not concerned about missed or unordered DDS samples.”
- Reliable delivery model “Make sure all DDS samples get there, in order.”

10.1.1 Best-effort Delivery Model

By default, Connex DDS uses the best-effort delivery model: there is no effort spent ensuring in-order delivery or resending lost DDS samples. Best-effort *DataReaders* ignore lost DDS samples

in favor of the latest DDS sample. Your application is only notified if it does not receive a new DDS sample within a certain time period (set in the [DEADLINE QoS Policy \(Section 6.5.5 on page 363\)](#)).

The best-effort delivery model is best for time-critical information that is sent continuously. For instance, consider a *DataWriter* for the value of a sensor device (such as the pressure inside a tank), and assume the *DataWriter* sends DDS samples continuously. In this situation, a *DataReader* for this *Topic* is only interested in having the latest pressure reading available—older DDS samples are obsolete.

10.1.2 Reliable Delivery Model

Reliable delivery means the DDS samples are guaranteed to arrive, in the order published.

The *DataWriter* maintains a *send queue* with space to hold the last X number of DDS samples sent. Similarly, a *DataReader* maintains a *receive queue* with space for consecutive X expected DDS samples.

The *send* and *receive queues* are used to temporarily cache DDS samples until Connex DDS is sure the DDS samples have been delivered and are not needed anymore. Connex DDS removes DDS samples from a publication's *send queue* after the DDS sample has been acknowledged by all reliable subscriptions. When positive acknowledgements are disabled (see [DATA_WRITER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#) and [DATA_READER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#)), DDS samples are removed from the send queue after the corresponding keep-duration has elapsed (see [Table 6.37 DDS_RtpsReliableWriterProtocol_t](#)).

If an out-of-order DDS sample arrives, Connex DDS speculatively caches it in the *DataReader's receive queue* (provided there is space in the queue). Only consecutive DDS samples are passed on to the *DataReader*.

DataWriters can be set up to wait for available queue space when sending DDS samples. This will cause the sending thread to block until there is space in the *send queue*. (Or, you can decide to sacrifice sending DDS samples reliably so that the sending rate is not compromised.) If the *DataWriter* is set up to ignore the full queue and sends anyway, then older cached DDS samples will be pushed out of the queue before all *DataReaders* have received them. In this case, the *DataReader* (or its *Subscriber*) is notified of the missing DDS samples through its *Listener* and/or *Conditions*.

Connex DDS automatically sends acknowledgments (ACKNACKs) as necessary to maintain reliable communications. The *DataWriter* may choose to block for a specified duration to wait for these acknowledgments (see [Waiting for Acknowledgments in a DataWriter \(Section 6.3.11 on page 288\)](#)).

Connex DDS establishes a virtual reliable channel between the matching *DataWriter* and all *DataReaders*. This mechanism isolates *DataReaders* from each other, allows the application to control memory usage, and provides mechanisms for the *DataWriter* to balance reliability and determinism. Moreover, the use of *send* and *receive queues* allows Connex DDS to be implemented efficiently without introducing unnecessary delays in the stream.

Note that a successful return code (`DDS_RETCODE_OK`) from `write()` does not necessarily mean that all *DataReaders* have received the data. It only means that the DDS sample has been added to the *DataWriter*'s queue. To see if all *DataReaders* have received the data, look at the [RELIABLE_WRITER_CACHE_CHANGED Status \(DDS Extension\)](#) (Section 6.3.6.8 on page 279) to see if any DDS samples are unacknowledged.

Suppose *DataWriter* A reliably publishes a *Topic* to which *DataReaders* B and C reliably subscribe. B has space in its queue, but C does not. Will *DataWriter* A be notified? Will *DataReader* C receive any error messages or callbacks? The exact behavior depends on the QoS settings:

- If `HISTORY_KEEP_ALL` is specified for C, C will reject DDS samples that cannot be put into the queue and request A to resend missing DDS samples. The *Listener* is notified with the `on_sample_rejected()` callback (see [SAMPLE_REJECTED Status \(Section 7.3.7.8 on page 479\)](#)). If A has a queue large enough, or A is no longer writing new DDS samples, A won't notice unless it checks the [RELIABLE_WRITER_CACHE_CHANGED Status \(DDS Extension\)](#) (Section 6.3.6.8 on page 279).
- If `HISTORY_KEEP_LAST` is specified for C, C will drop old DDS samples and accept new ones. To A, it is as if all DDS samples have been received by C (that is, they have all been acknowledged).

10.2 Overview of the Reliable Protocol

An important advantage of Connex DDS is that it can offer the reliability and other QoS guarantees mandated by DDS on top of a very wide variety of transports, including packet-based transports, unreliable networks, multicast-capable transports, bursty or high-latency transports, etc. Connex DDS is also capable of maintaining liveliness and application-level QoS even in the presence of sporadic connectivity loss at the transport level, an important benefit in mobile networks. Connex DDS accomplishes this by implementing a reliable protocol that sequences and acknowledges application-level messages and monitors the liveliness of the link. This is called the Real-Time Publish-Subscribe (RTPS) protocol; it is an open, international standard.¹

In order to work in this wide range of environments, the reliable protocol defined by RTPS is highly configurable with a set of parameters that let the application fine-tune its behavior to trade-off latency, responsiveness, liveliness, throughput, and resource utilization. This section describes the most important features to the extent needed to understand how the configuration parameters affect its operation.

The most important features of the RTPS protocol are:

¹For a link to the RTPS specification, see the RTI website, www.rti.com.

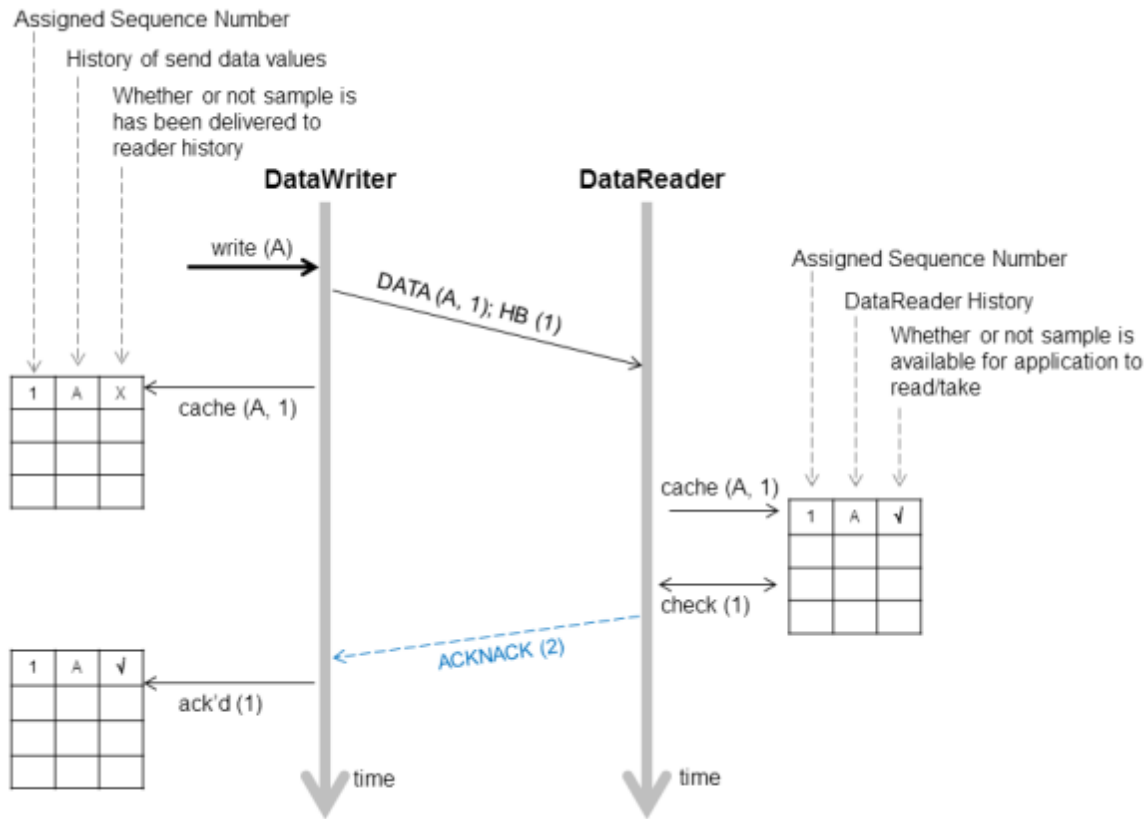
- Support for both push and pull operating modes
- Support for both positive and negative acknowledgments
- Support for high data-rate *DataWriters*
- Support for multicast *DataReaders*
- Support for high-latency environments

In order to support these features, RTPS uses several types of messages: Data messages (DATA), acknowledgments (ACKNACKs), and heartbeats (HBs).

- **DATA** messages contain snapshots of the value of data-objects and associate the snapshot with a sequence number that Connex DDS uses to identify them within the *DataWriter*'s history. These snapshots are stored in the history as a direct result of the application calling **write()** on the *DataWriter*. Incremental sequence numbers are automatically assigned by the *DataWriter* each time **write()** is called. In [Basic RTPS Reliable Protocol \(Section Figure 10.1 on the facing page\)](#) through [Using QosPolicies to Tune the Reliable Protocol \(Section 10.3 on page 635\)](#), these messages are represented using the notation DATA(<value>, <sequenceNum>). For example, DATA(A,1) represents a message that communicates the value 'A' and associates the sequence number '1' with this message. A DATA is used for both keyed and non-keyed data types.
- **HB** messages announce to the *DataReader* that it should have received all snapshots up to the one tagged with a range of sequence numbers and can also request the *DataReader* to send an acknowledgement back. For example, HB(1-3) indicates to the *DataReader* that it should have received snapshots tagged with sequence numbers 1, 2, and 3 and asks the *DataReader* to confirm this.
- **ACKNACK** messages communicate to the *DataWriter* that particular snapshots have been successfully stored in the *DataReader*'s history. ACKNACKs also tell the *DataWriter* which snapshots are missing on the *DataReader* side. The ACKNACK message includes a set of sequence numbers represented as a bit map. The sequence numbers indicate which ones the *DataReader* is missing. (The bit map contains the base sequence number that has not been received, followed by the number of bits in bit map and the optional bit map. The maximum size of the bit map is 256.) All numbers up to (not including) those in the set are considered positively acknowledged. They are represented in [Figure 10.1 Basic RTPS Reliable Protocol on the facing page](#) through [Figure 10.7 Use of heartbeat_period on page 647](#) as ACKNACK(<first-missing>) or ACKNACK(<first-missing>-<last-missing>). For example, ACKNACK(4) indicates that the snapshots with sequence numbers 1, 2, and 3 have been successfully stored in the *DataReader* history, and that 4 has not been received.

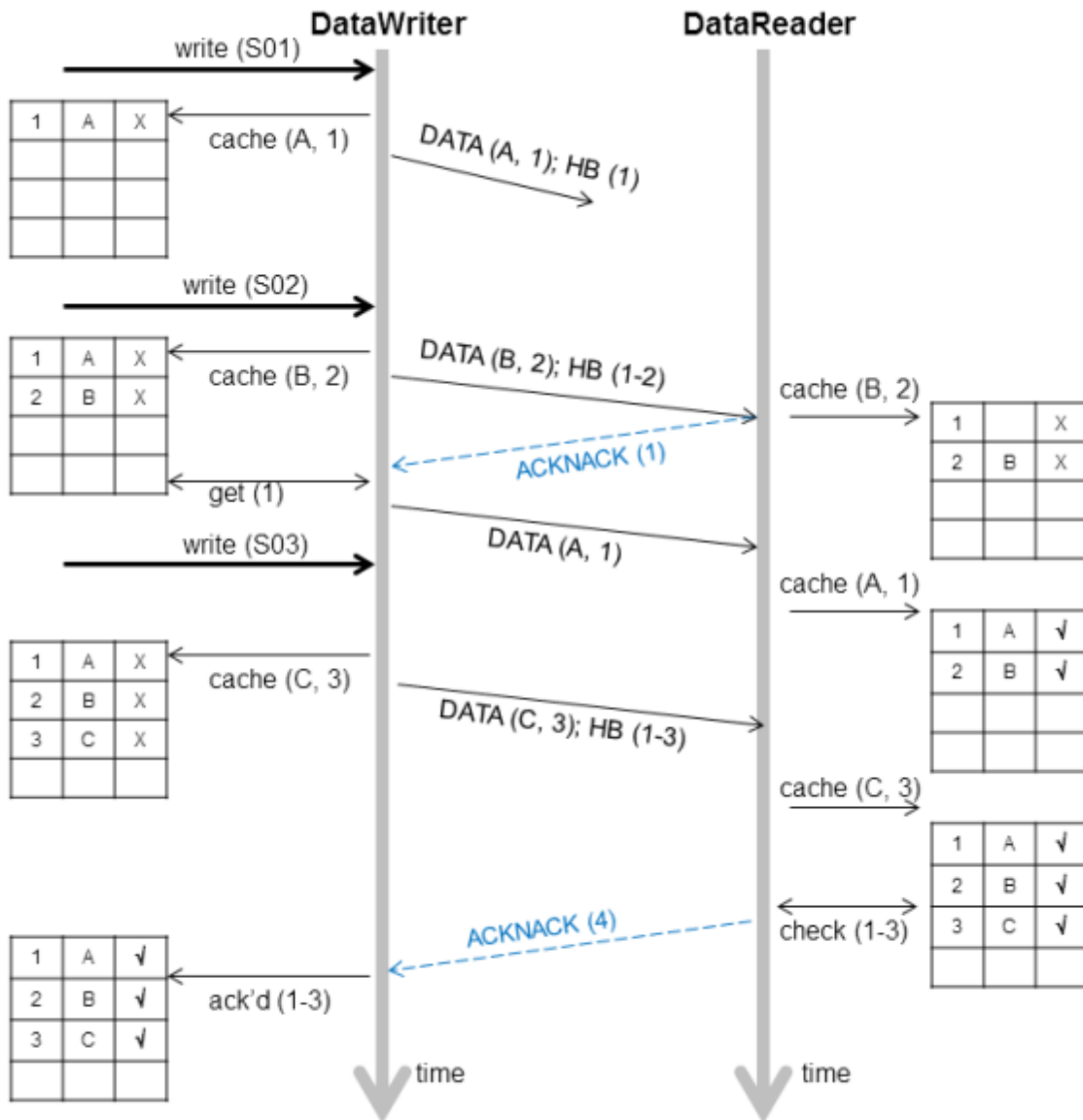
It is important to note that Connex DDS can bundle multiple of the above messages within a single network packet. This 'submessage bundling' provides for higher performance communications.

Figure 10.1 Basic RTPS Reliable Protocol



Basic RTPS Reliable Protocol (Section Figure 10.1 above) illustrates the basic behavior of the protocol when an application calls the `write()` operation on a *DataWriter* that is associated with a *DataReader*. As mentioned, the RTPS protocol can bundle multiple submessages into a single network packet. In Basic RTPS Reliable Protocol (Section Figure 10.1 above) this feature is used to piggyback a HB message to the DATA message. Note that before the message is sent, the data is given a sequence number (1 in this case) which is stored in the *DataWriter*'s send queue. As soon as the message is received by the *DataReader*, it places it into the *DataReader*'s receive queue. From the sequence number the *DataReader* can tell that it has not missed any messages and therefore it can make the data available immediately to the user (and call the *DataReaderListener*). This is indicated by the “✓” symbol. The reception of the HB(1) causes the *DataReader* to check that it has indeed received all updates up to and including the one with `sequenceNumber=1`. Since this is true, it replies with an `ACKNACK(2)` to positively acknowledge all messages up to (but not including) sequence number 2. The *DataWriter* notes that the update has been acknowledged, so it no longer needs to be retained in its send queue. This is indicated by the “✓” symbol.

Figure 10.2 RTPS Reliable Protocol in the Presence of Message Loss



RTPS Reliable Protocol in the Presence of Message Loss (Section Figure 10.2 above) illustrates the behavior of the protocol in the presence of lost messages. Assume that the message containing DATA(A,1) is dropped by the network. When the *DataReader* receives the next message (DATA(B,2); HB(1-2)) the *DataReader* will notice that the data associated with sequence number 1 was never received. It realizes this because the heartbeat HB(1-2) tells the *DataReader* that it should have received all messages up to and including the one with sequence number 2. This realization has two consequences:

- The data associated with sequence number 2 (B) is tagged with ‘X’ to indicate that it is not deliverable to the application (that is, it should not be made available to the application, because the application needs to receive the data associated with DDS sample 1 (A) first).
- An ACKNACK(1) is sent to the *DataWriter* to request that the data tagged with sequence number 1 be resent.

Reception of the ACKNACK(1) causes the *DataWriter* to resend DATA(A,1). Once the *DataReader* receives it, it can ‘commit’ both A and B such that the application can now access both (indicated by the “✓”) and call the *DataReaderListener*. From there on, the protocol proceeds as before for the next data message (C) and so forth.

A subtle but important feature of the RTPS protocol is that ACKNACK messages are only sent as a direct response to HB messages. This allows the *DataWriter* to better control the overhead of these ‘administrative’ messages. For example, if the *DataWriter* knows that it is about to send a chain of DATA messages, it can bundle them all and include a single HB at the end, which minimizes ACKNACK traffic.

10.3 Using QoS Policies to Tune the Reliable Protocol

Reliability is controlled by the QoS Policies in [Table 10.1 QoS Policies for Reliable Communications](#). To enable reliable delivery, read the following sections to learn how to change the QoS for the *DataWriter* and *DataReader*:

- [Enabling Reliability \(Section 10.3.1 on page 637\)](#)
- [Tuning Queue Sizes and Other Resource Limits \(Section 10.3.2 on page 638\)](#)
- [Controlling Heartbeats and Retries with DataWriterProtocol QoS Policy \(Section 10.3.4 on page 645\)](#)
- [Avoiding Message Storms with DataReaderProtocol QoS Policy \(Section 10.3.5 on page 653\)](#)
- [Resending DDS Samples to Late-Joiners with the Durability QoS Policy \(Section 10.3.6 on page 653\)](#)

Then see [Use Cases \(Section 10.3.7 on page 654\)](#) to explore example use cases:

Table 10.1 QoS Policies for Reliable Communications

QoS Policy	Description	Related Entities ¹	Reference
Reliability	To establish reliable communication, this QoS must be set to DDS_RELIABLE_RELIABILITY_QOS for the <i>DataWriter</i> and its <i>DataReaders</i> .	DW, DR	Enabling Reliability (Section 10.3.1 on the facing page) , RELIABILITY QoS Policy (Section 6.5.19 on page 400)
ResourceLimits	This QoS determines the amount of resources each side can use to manage instances and DDS samples of instances. Therefore it controls the size of the <i>DataWriter</i> 's send queue and the <i>DataReader</i> 's receive queue. The send queue stores DDS samples until they have been ACKed by all <i>DataReaders</i> . The <i>DataReader</i> 's receive queue stores DDS samples for the user's application to access.	DW, DR	Tuning Queue Sizes and Other Resource Limits (Section 10.3.2 on page 638) , RESOURCE_LIMITS QoS Policy (Section 6.5.20 on page 405)
History	This QoS affects how a <i>DataWriter/DataReader</i> behaves when its send/receive queue fills up.	DW, DR	Controlling Queue Depth with the History QoS Policy (Section 10.3.3 on page 644) , HISTORY QoS Policy (Section 6.5.10 on page 376)
DataWriterProtocol	This QoS configures <i>DataWriter</i> -specific protocol. The QoS can disable positive ACKs for its <i>DataReaders</i> .	DW	Controlling Heartbeats and Retries with DataWriterProtocol QoS Policy (Section 10.3.4 on page 645) , DATA_WRITER_PROTOCOL QoS Policy (DDS Extension) (Section 6.5.3 on page 347)
DataReaderProtocol	When a reliable <i>DataReader</i> receives a heartbeat from a <i>DataWriter</i> and needs to return an ACKNACK, the <i>DataReader</i> can choose to delay a while. This QoS sets the minimum and maximum delay. It can also disable positive ACKs for the <i>DataReader</i> .	DR	Avoiding Message Storms with DataReaderProtocol QoS Policy (Section 10.3.5 on page 653) , DATA_READER_PROTOCOL QoS Policy (DDS Extension) (Section 7.6.1 on page 511)
DataReaderResourceLimits	This QoS determines additional amounts of resources that the <i>DataReader</i> can use to manage DDS samples (namely, the size of the <i>DataReader</i> 's internal queues, which cache DDS samples until they are ordered for reliability and can be moved to the <i>DataReader</i> 's receive queue for access by the user's application).	DR	Tuning Queue Sizes and Other Resource Limits (Section 10.3.2 on page 638) , DATA_READER_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 7.6.2 on page 517)

¹DW = DataWriter, DR = DataReader

Table 10.1 QosPolicies for Reliable Communications

QosPolicy	Description	Related Entities ¹	Reference
Durability	This QoS affects whether late-joining <i>DataReaders</i> will receive all previously-sent data or not.	DW, DR	Resending DDS Samples to Late-Joiners with the Durability QosPolicy (Section 10.3.6 on page 653) , DURABILITY QosPolicy (Section 6.5.7 on page 368)

10.3.1 Enabling Reliability

You must modify the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) of the *DataWriter* and each of its reliable *DataReaders*. Set the **kind** field to `DDS_RELIABLE_RELIABILITY_QOS`:

- *DataWriter*

```
writer_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
```

- *DataReader*

```
reader_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
```

10.3.1.1 Blocking until the Send Queue Has Space Available

The **max_blocking_time** property in the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) indicates how long a *DataWriter* can be blocked during a **write()**.

If **max_blocking_time** is non-zero and the reliability send queue is full, the write is blocked (the DDS sample is not sent). If **max_blocking_time** has passed and the DDS sample is still *not* sent, **write()** returns `DDS_RETCODE_TIMEOUT` and the DDS sample is not sent.

If the number of unacknowledged DDS samples in the reliability send queue drops below `max_samples` (set in the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#)) before **max_blocking_time**, the DDS sample is sent and **write()** returns `DDS_RETCODE_OK`.

If **max_blocking_time** is zero and the reliability send queue is full, **write()** returns `DDS_RETCODE_TIMEOUT` and the DDS sample is not sent.

¹DW = DataWriter, DR = DataReader

10.3.2 Tuning Queue Sizes and Other Resource Limits

Set the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) appropriately to accommodate however many DDS samples should be saved in the *DataWriter*'s send queue or the *DataReader*'s receive queue. *The defaults may suit your needs*; if so, you do not have to modify this QosPolicy.

Set the `DDS_RtpsReliableWriterProtocol_t` in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#) appropriately to accommodate the number of unacknowledged DDS samples that can be in-flight at a time from a *DataWriter*.

For more information, see the following sections:

- [Understanding the Send Queue and Setting its Size \(Section 10.3.2.1 on the facing page\)](#)
- [Understanding the Receive Queue and Setting Its Size \(Section 10.3.2.2 on page 642\)](#)

Note: The `HistoryQosPolicy`'s `depth` must be less than or equal to the `ResourceLimitsQosPolicy`'s `max_samples_per_instance`; `max_samples_per_instance` must be less than or equal to the `ResourceLimitsQosPolicy`'s `max_samples` (see [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#)), and `max_samples_per_remote_writer` (see [DATA_READER_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 7.6.2 on page 517\)](#)) must be less than or equal to `max_samples`.

- `depth <= max_samples_per_instance <= max_samples`
- `max_samples_per_remote_writer <= max_samples`

Examples:

DataWriter

```
writer_qos.resource_limits.initial_instances = 10;
writer_qos.resource_limits.initial_samples = 200;
writer_qos.resource_limits.max_instances = 100;
writer_qos.resource_limits.max_samples = 2000;
writer_qos.resource_limits.max_samples_per_instance = 20;
writer_qos.history.depth = 20;
```

DataReader

```
reader_qos.resource_limits.initial_instances = 10;
reader_qos.resource_limits.initial_samples = 200;
reader_qos.resource_limits.max_instances = 100;
reader_qos.resource_limits.max_samples = 2000;
reader_qos.resource_limits.max_samples_per_instance = 20;
reader_qos.history.depth = 20;
reader_qos.reader_resource_limits.max_samples_per_remote_writer = 20;
```

10.3.2.1 Understanding the Send Queue and Setting its Size

A *DataWriter*'s send queue is used to store each DDS sample it writes. A DDS sample will be removed from the send queue after it has been acknowledged (through an ACKNACK) by all the reliable *DataReaders*. A *DataReader* can request that the *DataWriter* resend a missing DDS sample (through an ACKNACK). If that DDS sample is still available in the send queue, it will be resent. To elicit timely ACKNACKs, the *DataWriter* will regularly send heartbeats to its reliable *DataReaders*.

A *DataWriter*'s send queue size is determined by its [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#), specifically the **max_samples** field. The appropriate value depends on application parameters such as how fast the publication calls **write()**.

A *DataWriter* has a "send window" that is the maximum number of unacknowledged DDS samples allowed in the send queue at a time. The send window enables configuration of the number of DDS samples queued for reliability to be done independently from the number of DDS samples queued for history. This is of great benefit when the size of the history queue is much different than the size of the reliability queue. For example, you may want to resend a large history to late-joining *DataReaders*, so the send queue size is large. However, you do not want performance to suffer due to a large send queue; this can happen when the send rate is greater than the read rate, and the *DataWriter* has to resend many DDS samples from its large historical send queue. If the send queue size was both the historical and reliability queue size, then both these goals could not be met. Now, with the send window, having a large history with good live reliability performance is possible.

The send window is determined by the *DataWriterProtocolQosPolicy*, specifically the fields **min_send_window_size** and **max_send_window_size** within the **rtps_reliable_writer** field of type *DDS_RtpsReliableWriterProtocol_t*. Other fields control a dynamic send window, where the send window size changes in response to network congestion to maximize the effective send rate. Like for **max_samples**, the appropriate values depend on application parameters.

Strict reliability: If a *DataWriter* does not receive ACKNACKs from one or more reliable *DataReaders*, it is possible for the reliability send queue—either its finite send window, or **max_samples** if its send window is infinite—to fill up. If you want to achieve strict reliability, the **kind** field in the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) for both the *DataReader* and *DataWriter* must be set to **KEEP_ALL**, positive acknowledgments must be enabled for both the *DataReader* and *DataWriter*, and your publishing application should wait until space is available in the reliability queue before writing any more DDS samples. Connex DDS provides two mechanisms to do this:

- Allow the **write()** operation to block until there is space in the reliability queue again to store the DDS sample. The maximum time this call blocks is determined by the **max_blocking_time** field in the [RELIABILITY QosPolicy \(Section 6.5.19 on page 400\)](#) (also discussed in [Blocking until the Send Queue Has Space Available \(Section 10.3.1.1 on page 637\)](#)).
- Use the *DataWriter*'s *Listener* to be notified when the reliability queue fills up or empties again.

When the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#) on the *DataWriter* is set to `KEEP_LAST`, strict reliability is not guaranteed. When there are **depth** number of DDS samples in the queue (set in the [HISTORY QosPolicy \(Section 6.5.10 on page 376\)](#), see [Controlling Queue Depth with the History QosPolicy \(Section 10.3.3 on page 644\)](#)) the oldest DDS sample will be dropped from the queue when a new DDS sample is written. *Note that in such a reliable mode, when the send window is larger than **max_samples**, the *DataWriter* will never block, but strict reliability is no longer guaranteed.* If there is a request for the purged DDS sample from any *DataReaders*, the *DataWriter* will send a heartbeat that no longer contains the sequence number of the dropped DDS sample (it will not be able to send the DDS sample).

Alternatively, a *DataWriter* with `KEEP_LAST` may block on `write()` when its send window is smaller than its send queue. The *DataWriter* will block when its send window is full. Only after the blocking time has elapsed, the *DataWriter* will purge a DDS sample, and then strict reliability is no longer guaranteed.

The send queue size is set in the `max_samples` field of the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#). The appropriate size for the send queue depends on application parameters (such as the send rate), channel parameters (such as end-to-end delay and probability of packet loss), and quality of service requirements (such as maximum acceptable probability of DDS sample loss).

The *DataReader's* receive queue size should generally be larger than the *DataWriter's* send queue size. Receive queue size is discussed in [Understanding the Receive Queue and Setting Its Size \(Section 10.3.2.2 on page 642\)](#).

A good rule of thumb, based on a simple model that assumes individual packet drops are not correlated and time-independent, is that the size of the reliability send queue, N , is as shown in [Calculating Minimum Send Queue Size for a Desired Level of Reliability \(Section Figure 10.3 below\)](#).

Figure 10.3 Calculating Minimum Send Queue Size for a Desired Level of Reliability

$$N = 2RT(\log(1-Q))/\log(p)$$

Simple formula for determining the minimum size of the send queue required for strict reliability

In the above equation, R is the rate of sending DDS samples, T is the round-trip transmission time, p is the probability of a packet loss in a round trip, and Q is the required probability that a DDS sample is eventually successfully delivered. Of course, network-transport dropouts must also be taken into account and may influence or dominate this calculation.

[Table 10.2 Required Size of the Send Queue for Different Network Parameters](#) gives the required size of the send queue for several common scenarios.

Table 10.2 Required Size of the Send Queue for Different Network Parameters

Q ¹	p ²	T ³	R ⁴	N ⁵
99%	1%	0.001 ⁶ sec	100 Hz	1
99%	1%	0.001 sec	2000 Hz	2
99%	5%	0.001 sec	100 Hz	1
99%	5%	0.001 sec	2000 Hz	4
99.99%	1%	0.001 sec	100 Hz	1
99.99%	1%	0.001 sec	2000 Hz	6
99.99%	5%	0.001 sec	100 Hz	1
99.99%	5%	0.001 sec	2000 Hz	8

Note: Packet loss on a network frequently happens in bursts, and the packet loss events are correlated. This means that the probability of a packet being lost is much higher if the previous packet was lost because it indicates a congested network or busy receiver. For this situation, it may be better to use a queue size that can accommodate the longest period of network congestion, as illustrated in [Calculating Minimum Send Queue Size for Networks with Dropouts \(Section Figure 10.4 below\)](#).

Figure 10.4 Calculating Minimum Send Queue Size for Networks with Dropouts

$$N = RD(Q)$$

Send queue size as a function of send rate "R" and maximum dropout time D

¹"Q" is the desired level of reliability measured as the probability that any data update will eventually be delivered successfully. In other words, percentage of DDS samples that will be successfully delivered.

²"p" is the probability that any single packet gets lost in the network.

³"T" is the round-trip transport delay in the network

⁴"R" is the rate at which the publisher is sending updates.

⁵"N" is the minimum required size of the send queue to accomplish the desired level of reliability "Q".

⁶The typical round-trip delay for a dedicated 100 Mbit/second ethernet is about 0.001 seconds.

In the above equation R is the rate of sending DDS samples, $D(Q)$ is a time such that Q percent of the dropouts are of equal or lesser length, and Q is the required probability that a DDS sample is eventually successfully delivered. The problem with the above formula is that it is hard to determine the value of $D(Q)$ for different values of Q .

For example, if we want to ensure that 99.9% of the DDS samples are eventually delivered successfully, and we know that the 99.9% of the network dropouts are shorter than 0.1 seconds, then we would use $N = 0.1 * R$. So for a rate of 100Hz, we would use a send queue of $N = 10$; for a rate of 2000Hz, we would use $N = 200$.

10.3.2.2 Understanding the Receive Queue and Setting Its Size

DDS samples are stored in the *DataReader's* receive queue, which is accessible to the user's application.

A DDS sample is removed from the receive queue after it has been accessed by `take()`, as described in [Accessing DDS Data Samples with Read or Take \(Section 7.4.3 on page 493\)](#). Note that `read()` does not remove DDS samples from the queue.

A *DataReader's* receive queue size is limited by its [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#), specifically the `max_samples` field. The storage of out-of-order DDS samples for each *DataWriter* is also allocated from the *DataReader's* receive queue; this DDS sample resource is shared among all reliable *DataWriters*. That is, **max_samples** includes both ordered and out-of-order DDS samples.

A *DataReader* can maintain reliable communications with multiple *DataWriters* (e.g., in the case of the [OWNERSHIP_STRENGTH QosPolicy \(Section 6.5.16 on page 393\)](#) setting of SHARED). The maximum number of out-of-order DDS samples from any one *DataWriter* that can occupy in the receive queue is set in the `max_samples_per_remote_writer` field of the [DATA_READER_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 7.6.2 on page 517\)](#); this value can be used to prevent a single *DataWriter* from using all the space in the receive queue. **max_samples_per_remote_writer** must be set to be \leq **max_samples**.

The *DataReader* will cache DDS samples that arrive out of order while waiting for missing DDS samples to be resent. (Up to 256 DDS samples can be resent; this limitation is imposed by the wire protocol.) If there is no room, the *DataReader* has to reject out-of-order DDS samples and request them again later after the missing DDS samples have arrived.

The appropriate size of the receive queue depends on application parameters, such as the *DataWriter's* sending rate and the probability of a dropped DDS sample. However, the receive queue size should generally be larger than the send queue size. Send queue size is discussed in [Understanding the Send Queue and Setting its Size \(Section 10.3.2.1 on page 639\)](#).

[Effect of Receive-Queue Size on Performance: Large Queue Size \(Section Figure 10.5 on the facing page\)](#) and [Effect of Receive Queue Size on Performance: Small Queue Size \(Section Figure 10.6 on page 644\)](#) compare two hypothetical *DataReaders*, both interacting with the same *DataWriter*. The queue on the left

represents an ordering cache, allocated from receive queue—DDS samples are held here if they arrive out of order. The *DataReader* in [Effect of Receive-Queue Size on Performance: Large Queue Size \(Section Figure 10.5 below\)](#) has a sufficiently large receive queue (**max_samples**) for the given send rate of the *DataWriter* and other operational parameters. In both cases, we assume that all DDS samples are *taken* from the *DataReader* in the *Listener* callback. (See [Accessing DDS Data Samples with Read or Take \(Section 7.4.3 on page 493\)](#) for information on `take()` and related operations.)

In [Effect of Receive Queue Size on Performance: Small Queue Size \(Section Figure 10.6 on the next page\)](#), **max_samples** is too small to cache out-of-order DDS samples for the same operational parameters. In both cases, the *DataReaders* eventually receive all the DDS samples in order. However, the *DataReader* with the larger **max_samples** will get the DDS samples earlier and with fewer transactions. In particular, DDS sample “4” is never resent for the *DataReader* with the larger queue size.

Figure 10.5 Effect of Receive-Queue Size on Performance: Large Queue Size

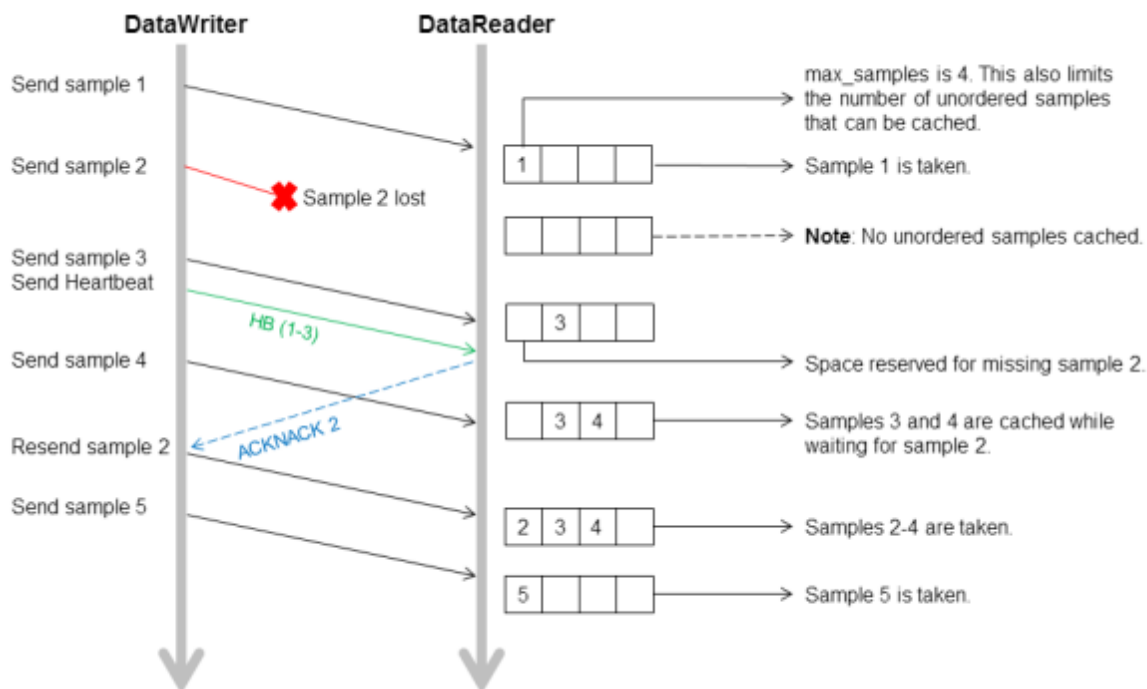
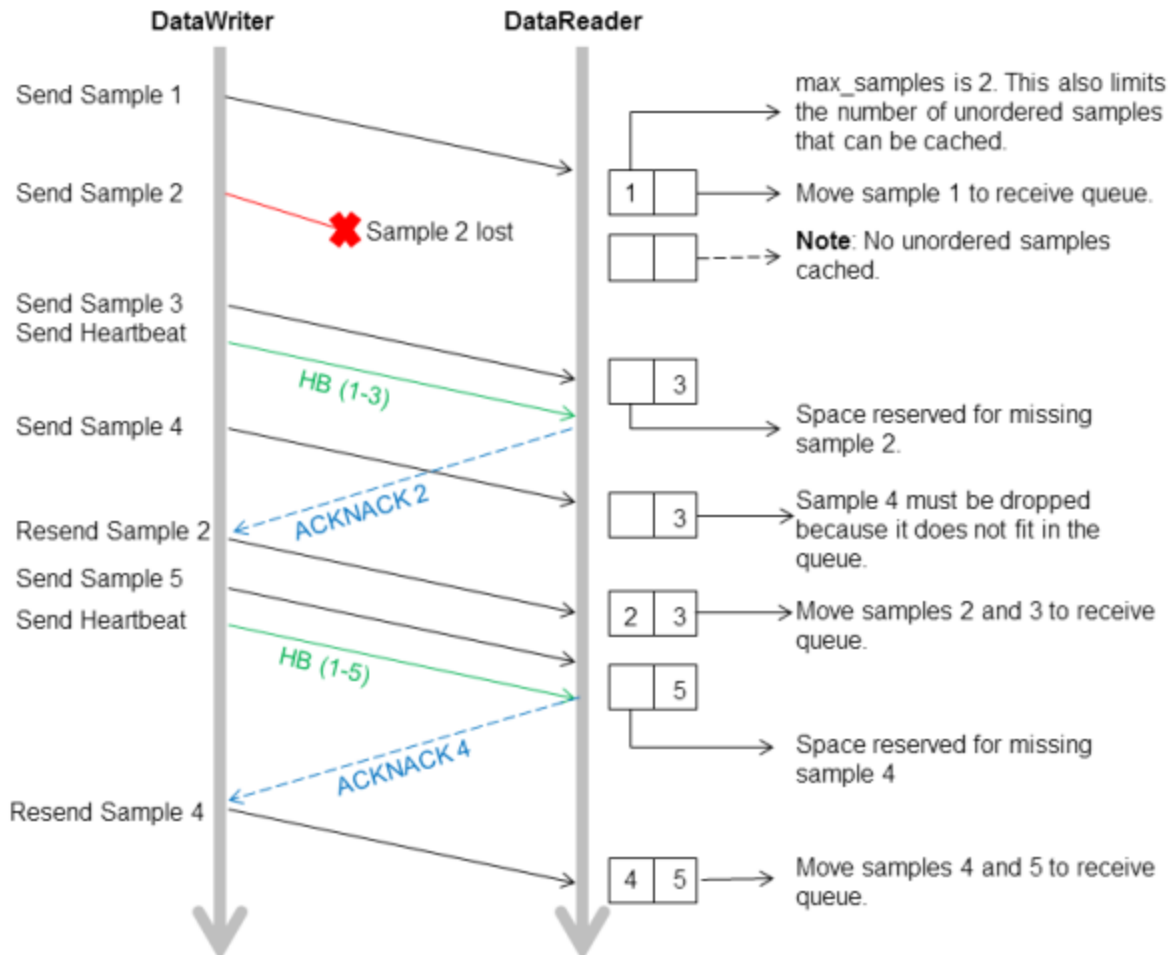


Figure 10.6 Effect of Receive Queue Size on Performance: Small Queue Size



10.3.3 Controlling Queue Depth with the History QoSPolicy

If you want to achieve strict reliability, set the *kind* field in the [HISTORY QoSPolicy](#) (Section 6.5.10 on page 376) for both the *DataReader* and *DataWriter* to `KEEP_ALL`; in this case, the *depth* does not matter.

Or, for non-strict reliability, you can leave the **kind** set to `KEEP_LAST` (the default). This will provide non-strict reliability; some DDS samples may not be delivered if the resource limit is reached.

The **depth** field in the [HISTORY QoSPolicy](#) (Section 6.5.10 on page 376) controls how many DDS samples Connex DDS will attempt to keep on the *DataWriter*'s send queue or the *DataReader*'s receive

queue. For reliable communications, depth should be ≥ 1 . The depth can be set to 1, but cannot be more than the `max_samples_per_instance` in [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#).

Example:

- *DataWriter*

```
writer_qos.history.depth = <number of DDS samples to keep in send queue>;
```

- *DataReader*

```
reader_qos.history.depth = <number of DDS samples to keep in receive queue>;
```

10.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

In the Connext DDS reliability model, the *DataWriter* sends DDS data samples and heartbeats to reliable *DataReaders*. A *DataReader* responds to a heartbeat by sending an ACKNACK, which tells the *DataWriter* what the *DataReader* has received so far.

In addition, the *DataReader* can request missing DDS samples (by sending an ACKNACK) and the *DataWriter* will respond by resending the missing DDS samples. This section describes some advanced timing parameters that control the behavior of this mechanism. Many applications do not need to change these settings. These parameters are contained in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#).

The protocol described in [Overview of the Reliable Protocol \(Section 10.2 on page 631\)](#) uses very simple rules such as piggybacking HB messages to each DATA message and responding immediately to ACKNACKs with the requested repair messages. While correct, this protocol would not be capable of accommodating optimum performance in more advanced use cases.

This section describes some of the parameters configurable by means of the `rtps_reliable_writer` structure in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#) and how they affect the behavior of the RTPS protocol.

10.3.4.1 How Often Heartbeats are Resent (`heartbeat_period`)

If a *DataReader* does not acknowledge a DDS sample that has been sent, the *DataWriter* resends the heartbeat. These heartbeats are resent at the rate set in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#), specifically its `heartbeat_period` field.

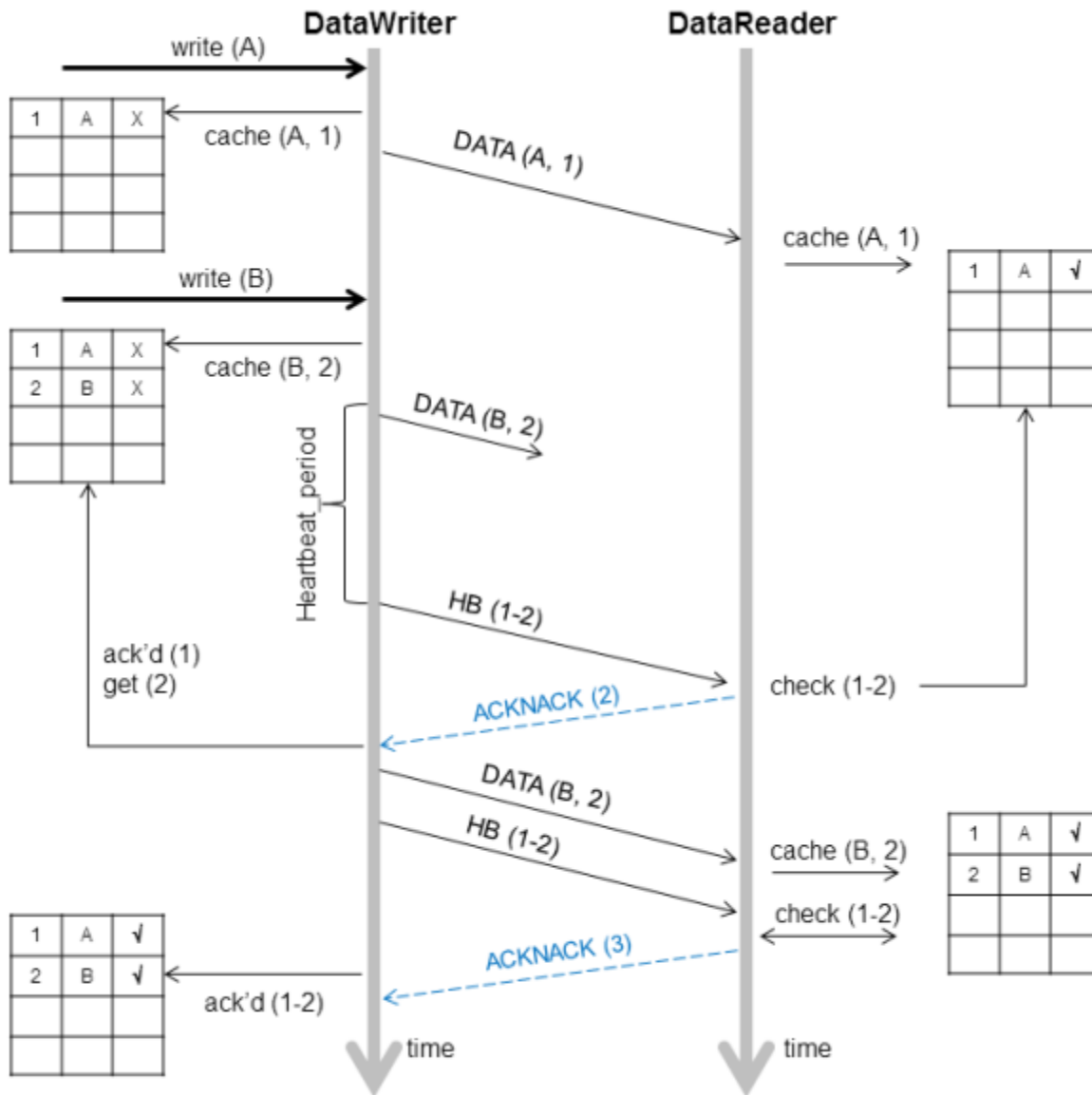
For example, a **`heartbeat_period`** of 3 seconds means that if a *DataReader* does not receive the latest DDS sample (for example, it gets dropped by the network), it might take up to 3 seconds before the *DataReader* realizes it is missing data. The application can lower this value when it is important that recovery from packet loss is very fast.

The basic approach of sending HB messages as a piggyback to DATA messages has the advantage of minimizing network traffic. However, there is a situation where this approach, by itself, may result in large latencies. Suppose there is a *DataWriter* that writes bursts of data, separated by relatively long periods of silence. Furthermore assume that the last message in one of the bursts is lost by the network. This is the case shown for message DATA(B, 2) in [Use of `heartbeat_period` \(Section Figure 10.7 on the facing page\)](#). If HBs were only sent piggybacked to DATA messages, the *DataReader* would not realize it missed the ‘B’ DATA message with sequence number ‘2’ until the *DataWriter* wrote the next message. This may be a long time if data is written sporadically. To avoid this situation, Connex DDS can be configured so that HBs are sent periodically as long as there are DDS samples that have not been acknowledged even if no data is being sent. The period at which these HBs are sent is configurable by setting the `rtps_reliable_writer.heartbeat_period` field in the [DATA_WRITER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#).

Note that a small value for the `heartbeat_period` will result in a small worst-case latency if the last message in a burst is lost. This comes at the expense of the higher overhead introduced by more frequent HB messages.

Also note that the `heartbeat_period` should not be less than the `rtps_reliable_reader.heartbeat_suppression_duration` in the [DATA_READER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#); otherwise those HBs will be lost.

Figure 10.7 Use of heartbeat_period



10.3.4.2 How Often Piggyback Heartbeats are Sent (heartbeats_per_max_samples)

A *DataWriter* will automatically send heartbeats with new DDS samples to request regular ACKNACKs from the *DataReader*. These are called “piggyback” heartbeats.

A piggyback heartbeat is sent every $\lceil \frac{\text{current send-window size}}{\text{heartbeats_per_max_samples}} \rceil$ number of DDS samples written.

The `heartbeats_per_max_samples` field is part of the `rtps_reliable_writer` structure in the `DATA_WRITER_PROTOCOL QoSPolicy (DDS Extension)` (Section 6.5.3 on page 347). If `heartbeats_per_max_samples` is set equal to `max_send_window_size`, this means that a heartbeat will be sent with each DDS sample. A value of 8 means that a heartbeat will be sent with every 'current send-window size/8' DDS samples. Say current send window is 1024, then a heartbeat will be sent once every 128 DDS samples. If you set this to zero, DDS samples are sent without any piggyback heartbeat. The `max_send_window_size` field is part of the `DATA_WRITER_PROTOCOL QoSPolicy (DDS Extension)` (Section 6.5.3 on page 347).

Figure 10.1 Basic RTPS Reliable Protocol and Figure 10.2 RTPS Reliable Protocol in the Presence of Message Loss seem to imply that a heartbeat (HB) is sent as a piggyback to each DATA message. However, in situations where data is sent continuously at high rates, piggybacking a HB to each message may result in too much overhead; not so much on the HB itself, but on the ACKNACKs that would be sent back as replies by the *DataReader*.

There are two reasons to send a HB:

- To request that a *DataReader* confirm the receipt of data via an ACKNACK, so that the *DataWriter* can remove it from its send queue and therefore prevent the *DataWriter*'s history from filling up (which could cause the `write()` operation to temporarily block¹).
- To inform the *DataReader* of what data it should have received, so that the *DataReader* can send a request for missing data via an ACKNACK.

The *DataWriter*'s send queue can buffer many DDS data samples while it waits for ACKNACKs, and the *DataReader*'s receive queue can store out-of-order DDS samples while it waits for missing ones. So it is possible to send HB messages much less frequently than DATA messages. The ratio of piggyback HB messages to DATA messages is controlled by the `rtps_reliable_writer.heartbeats_per_max_samples` field in the `DATA_WRITER_PROTOCOL QoSPolicy (DDS Extension)` (Section 6.5.3 on page 347).

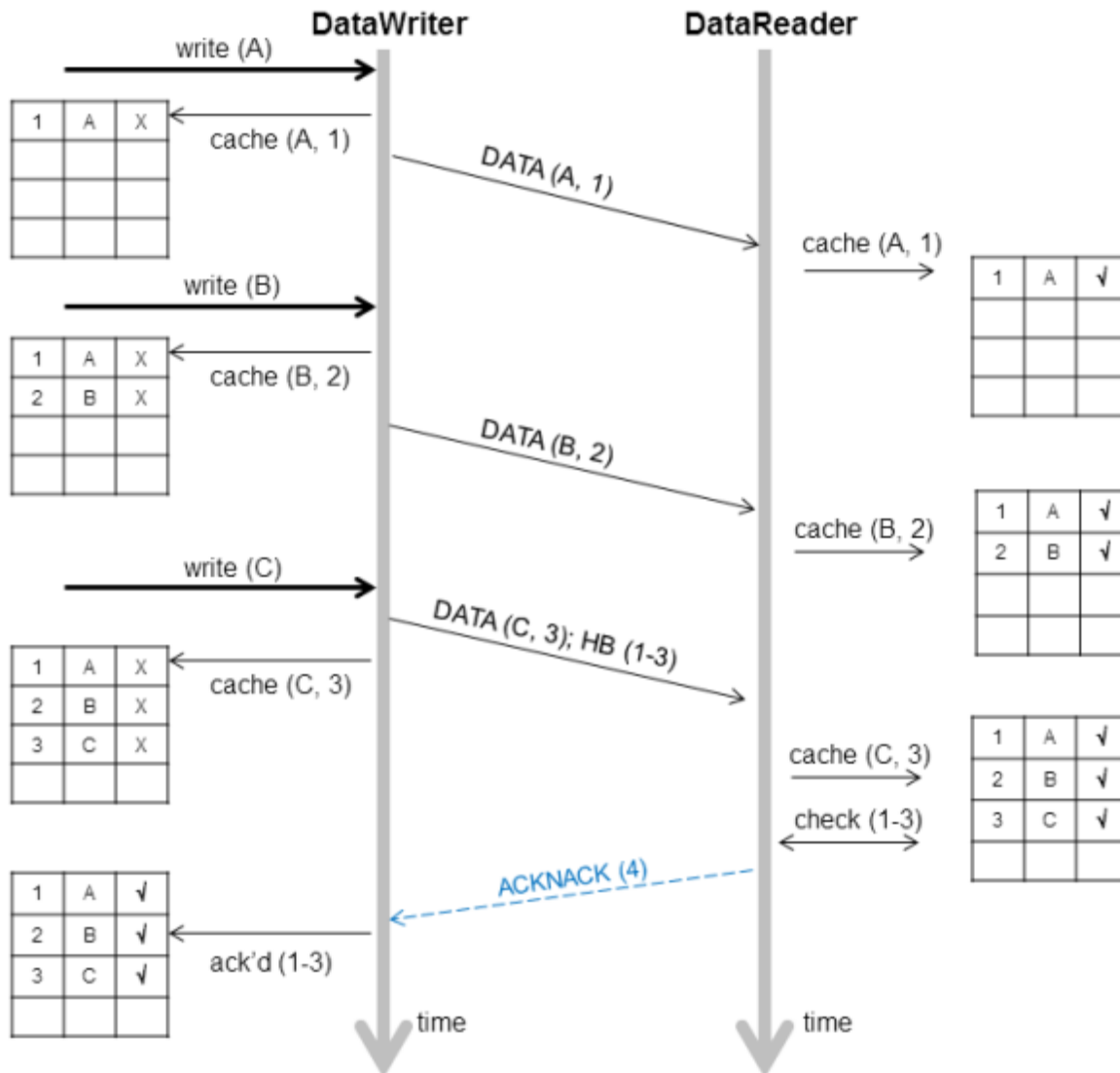
A HB is used to get confirmation from *DataReaders* so that the *DataWriter* can remove acknowledged DDS samples from the queue to make space for new DDS samples. Therefore, if the queue size is large, or new DDS samples are added slowly, HBs can be sent less frequently.

In [Use of `heartbeats_per_max_samples`](#) (Section Figure 10.8 on the facing page), the *DataWriter* sets the `heartbeats_per_max_samples` to certain value so that a piggyback HB will be sent for every three DDS samples. The *DataWriter* first writes DDS sample A and B. The *DataReader* receives both. However, since no HB has been received, the *DataReader* won't send back an ACKNACK. The *DataWriter* will still keep all the DDS samples in its queue. When the *DataWriter* sends DDS sample C, it will send a piggyback HB along with the DDS sample. Once the *DataReader* receives the HB, it will send back an

¹Note that data could also be removed from the *DataWriter*'s send queue if it is no longer relevant due to some other QoS such a HISTORY KEEP_LAST (`HISTORY QoSPolicy` (Section 6.5.10 on page 376)) or LIFESPAN (`LIFESPAN QoS Policy` (Section 6.5.12 on page 381)).

ACKNACK for DDS samples up to sequence number 3, such that the *DataWriter* can remove all three DDS samples from its queue.

Figure 10.8 Use of `heartbeats_per_max_samples`



10.3.4.3 Controlling Packet Size for Resent DDS Samples (`max_bytes_per_nack_response`)

A *DataWriter* may resend multiple missed DDS samples in the same packet. The `max_bytes_per_nack_response` field in the `DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)` (Section 6.5.3 on page 347) limits the size of this ‘repair’ packet. The reliable *DataWriter* will include at least one sample in the repair packet.

For example, if the *DataReader* requests 20 DDS samples, each 10K, and the `max_bytes_per_nack_response` is set to 100K, the *DataWriter* will only send the first 10 DDS samples at most. The *DataReader* will have to ACKNACK again to receive the other DDS samples.

Regardless of this setting, the maximum number of samples that can be part of a repair packet is limited to 32. This limit cannot be changed by configuration. In addition, the number of samples is limited by the value of `NDDS_Transport_Property_t`'s `gather_send_buffer_count_max` (see [Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6 \(Section 15.6.1 on page 763\)](#)).

10.3.4.4 Controlling How Many Times Heartbeats are Resent (`max_heartbeat_retries`)

If a *DataReader* does not respond within `max_heartbeat_retries` number of heartbeats, it will be dropped by the *DataWriter* and the reliable *DataWriter*'s *Listener* will be called with a `RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension)` ([Section 6.3.6.9 on page 281](#)).

If the dropped *DataReader* becomes available again (perhaps its network connection was down temporarily), it will be added back to the *DataWriter* the next time the *DataWriter* receives some message (ACKNACK) from the *DataReader*.

When a *DataReader* is 'dropped' by a *DataWriter*, the *DataWriter* will not wait for the *DataReader* to send an ACKNACK before any DDS samples are removed. However, the *DataWriter* will still send data and HBs to this *DataReader* as normal.

The `max_heartbeat_retries` field is part of the `DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)` ([Section 6.5.3 on page 347](#)).

10.3.4.5 Treating Non-Progressing Readers as Inactive Readers (`inactivate_nonprogressing_readers`)

In addition to `max_heartbeat_retries`, if `inactivate_nonprogressing_readers` is set, then not only are non-responsive *DataReaders* considered inactive, but *DataReaders* sending non-progressing NACKs can also be considered inactive. A *non-progressing NACK* is one which requests the same oldest DDS sample as the previously received NACK. In this case, the *DataWriter* will not consider a non-progressing NACK as coming from an active reader, and hence will inactivate the *DataReader* if no new NACKs are received before `max_heartbeat_retries` number of heartbeat periods has passed.

One example for which it could be useful to turn on `inactivate_nonprogressing_readers` is when a *DataReader*'s (keep-all) queue is full of untaken historical DDS samples. Each subsequent heartbeat would trigger the same NACK, and nominally the *DataReader* would not be inactivated. A user not requiring strict-reliability could consider setting `inactivate_nonprogressing_readers` to allow the *DataWriter* to progress rather than being held up by this non-progressing *DataReader*.

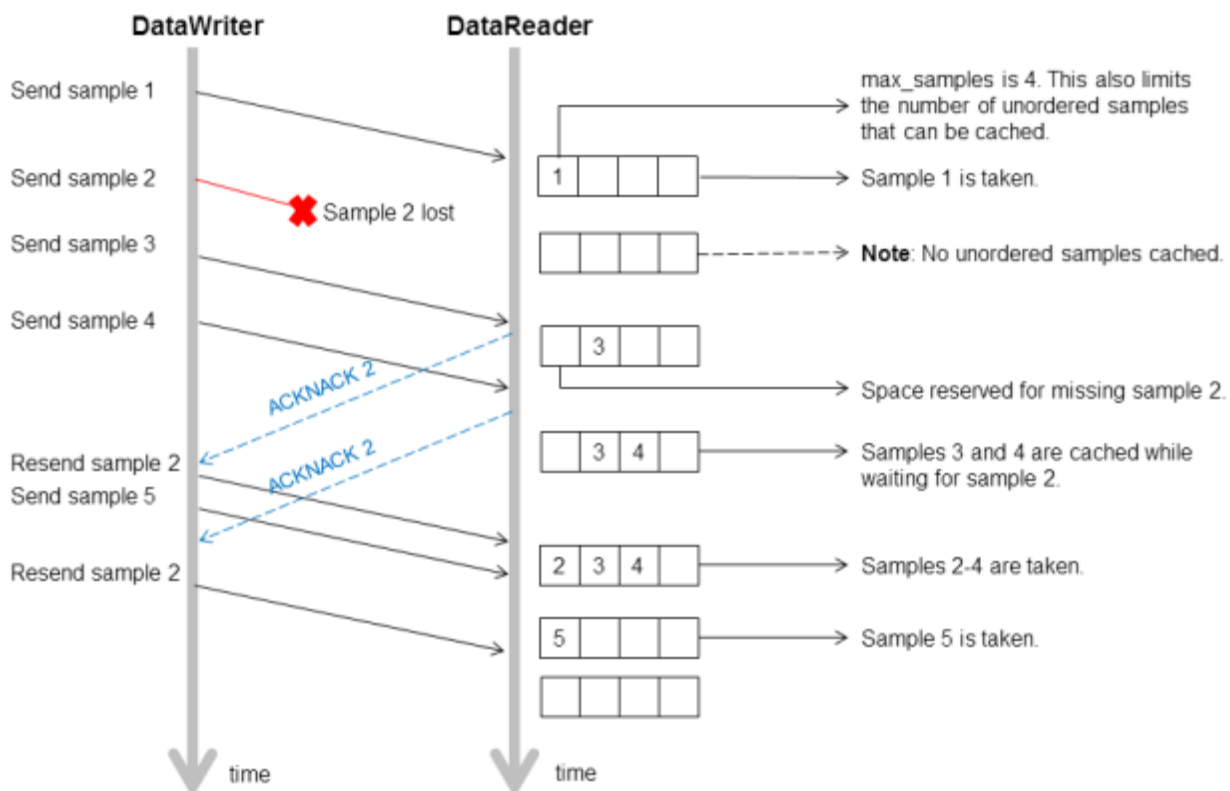
10.3.4.6 Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay)

When a *DataWriter* receives a request for missing DDS samples from a *DataReader* and responds by resending the requested DDS samples, it will ignore additional requests for the same DDS samples during the time period `max_nack_response_delay`.

The `rtps_reliable_writer.max_nack_response_delay` field is part of the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347).

If your send period is smaller than the round-trip delay of a message, this can cause unnecessary DDS sample retransmissions due to redundant ACKNACKs. In this situation, an ACKNACK triggered by an out-of-order DDS sample is not received before the next DDS sample is sent. When a *DataReader* receives the next message, it will send another ACKNACK for the missing DDS sample. As illustrated in [Resending Missing Samples due to Duplicate ACKNACKs](#) (Section Figure 10.9 below), duplicate ACKNACK messages cause another resending of missing DDS sample “2” and lead to wasted CPU usage on both the publication and the subscription sides.

Figure 10.9 Resending Missing Samples due to Duplicate ACKNACKs



While these redundant messages provide an extra cushion for the level of reliability desired, you can conserve the CPU and network bandwidth usage by limiting how often the same ACKNACK messages are sent; this is controlled by **min_nack_response_delay**.

Reliable subscriptions are prevented from resending an ACKNACK within **min_nack_response_delay** seconds from the last time an ACKNACK was sent for the same DDS sample. Our testing shows that the default **min_nack_response_delay** of 0 seconds achieves an optimal balance for most applications on typical Ethernet LANs.

However, if your system has very slow computers and/or a slow network, you may want to consider increasing **min_nack_response_delay**. Sending an ACKNACK and resending a missing DDS sample inherently takes a long time in this system. So you should allow a longer time for recovery of the lost DDS sample before sending another ACKNACK. In this situation, you should increase **min_nack_response_delay**.

If your system consists of a fast network or computers, *and* the receive queue size is very small, then you should keep **min_nack_response_delay** very small (such as the default value of 0). If the queue size is small, recovering a missing DDS sample is more important than conserving CPU and network bandwidth (new DDS samples that are too far ahead of the missing DDS sample are thrown away). A fast system can cope with a smaller **min_nack_response_delay** value, and the reliable DDS sample stream can normalize more quickly.

10.3.4.7 Disabling Positive Acknowledgements (disable_positive_acks_min_sample_keep_duration)

When ACKNACK storms are a primary concern in a system, an alternative to tuning heartbeat and ACKNACK response delays is to disable positive acknowledgments (ACKs) and rely just on NACKs to maintain reliability. Systems with non-strict reliability requirements can disable ACKs to reduce network traffic and directly solve the problem of ACK storms. ACKs can be disabled for the *DataWriter* and the *DataReader*; when disabled for the *DataWriter*, none of its *DataReaders* will send ACKs, whereas disabling it at the *DataReader* allows per-*DataReader* configuration.

Normally when ACKs are enabled, strict reliability is maintained by the *DataWriter*, guaranteeing that a DDS sample stays in its send queue until all *DataReaders* have positively acknowledged it (aside from relevant DURABILITY, HISTORY, and LIFESPAN QoS policies). When ACKs are disabled, strict reliability is no longer guaranteed, but the *DataWriter* should still keep the DDS sample for a sufficient duration for ACK-disabled *DataReaders* to have a chance to NACK it. Thus, a configurable “keep-duration” (**disable_postive_acks_min_sample_keep_duration**) applies for DDS samples written for ACK-disabled *DataReaders*, where DDS samples are kept in the queue for at least that keep-duration. After the keep-duration has elapsed for a DDS sample, the DDS sample is considered to be “acknowledged” by its ACK-disabled *DataReaders*.

The keep duration should be configured for the expected worst-case from when the DDS sample is written to when a NACK for the DDS sample could be received. If set too short, the DDS sample may no longer be queued when a NACK requests it, which is the cost of not enforcing strict reliability.

If the peak send rate is known and writer resources are available, the writer queue can be sized so that writes will not block. For this case, the queue size must be greater than the send rate multiplied by the keep duration.

10.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy

DataWriters send DDS data samples and heartbeats to *DataReaders*. A *DataReader* responds to a heartbeat by sending an acknowledgement that tells the *DataWriter* what the *DataReader* has received so far and what it is missing. If there are many *DataReaders*, all sending ACKNACKs to the same *DataWriter* at the same time, a message storm can result. To prevent this, you can set a delay for each *DataReader*, so they don't all send ACKNACKs at the same time. This delay is set in the [DATA_READER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#).

If you have several *DataReaders* per *DataWriter*, varying this delay for each one can avoid ACKNACK message storms to the *DataWriter*. If you are not concerned about message storms, you do not need to change this QosPolicy.

Example:

```
reader_qos.protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;

reader_qos.protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;

reader_qos.protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;

reader_qos.protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec =
    0.5 * 1000000000ULL; // 0.5 sec
```

As the name suggests, the minimum and maximum response delay bounds the random wait time before the response. Setting both to zero will force immediate response, which may be necessary for the fastest recovery in case of lost DDS samples.

10.3.6 Resending DDS Samples to Late-Joiners with the Durability QosPolicy

The [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) is also somewhat related to Reliability. Connex DDS requires a finite time to "discover" or match *DataReaders* to *DataWriters*. If an application attempts to send data before the *DataReader* and *DataWriter* "discover" one another, then the DDS sample will not actually get sent. Whether or not DDS samples are resent when the *DataReader* and *DataWriter* eventually "discover" one another depends on how the DURABILITY and HISTORY QoS are set. The default setting for the Durability QosPolicy is VOLATILE, which means that the *DataWriter* will not store DDS samples for redelivery to late-joining *DataReaders*.

Connex DDS also supports the TRANSIENT_LOCAL setting for the Durability, which means that the DDS samples will be kept stored for redelivery to late-joining *DataReaders*, as long as the *DataWriter* is

around and the [RESOURCE_LIMITS QoSPolicy \(Section 6.5.20 on page 405\)](#) allows. The DDS samples are not stored beyond the lifecycle of the `DataWriter`.

See also: [Waiting for Historical Data \(Section 7.3.6 on page 469\)](#).

10.3.7 Use Cases

This section contains advanced material that discusses practical applications of the reliability related QoS.

10.3.7.1 Importance of Relative Thread Priorities

For high throughput, the Connex DDS Event thread's priority must be sufficiently high on the sending application. Unlike an unreliable writer, a reliable writer relies on internal Connex DDS threads: the Receive thread processes ACKNACKs from the *DataReaders*, and the Event thread schedules the events necessary to maintain reliable data flow.

- When DDS samples are sent to the same or another application on the same host, the Receive thread priority should be higher than the writing thread priority (priority of the thread calling `write()` on the *DataWriter*). This will allow the Receive thread to process the messages as they are sent by the writing thread. A sustained reliable flow requires the reader to be able to process the DDS samples from the writer at a speed equal to or faster than the writer emits.
- The default Event thread priority is low. This is adequate if your reliable transfer is not sustained; queued up events will eventually be processed when the writing thread yields the CPU. The Connex DDS can automatically grow the event queue to store all pending events. But if the reliable communication is sustained, reliable events will continue to be scheduled, and the event queue will eventually reach its limit. The default Event thread priority is unsuitable for maintaining a fast and sustained reliable communication and should be increased through the `participant_qos.event-thread.priority`. This value maps directly to the OS thread priority, see [EVENT QoSPolicy \(DDS Extension\) \(Section 8.5.5 on page 602\)](#).

The Event thread should also be increased to minimize the reliable latency. If events are processed at a higher priority, dropped packets will be resent sooner.

Now we consider some practical applications of the reliability related QoS:

- [Aperiodic Use Case: One-at-a-Time \(Section 10.3.7.2 on the facing page\)](#)
- [Aperiodic, Bursty \(Section 10.3.7.3 on page 659\)](#)
- [Periodic \(Section 10.3.7.4 on page 664\)](#)

10.3.7.2 Aperiodic Use Case: One-at-a-Time

Suppose you have aperiodically generated data that needs to be delivered reliably, with minimum latency, such as a series of commands (“Ready,” “Aim,” “Fire”). If a writing thread may block between each DDS sample to guarantee reception of the just-sent DDS sample on the reader’s middleware end, a smaller queue will provide a smaller upper bound on the DDS sample delivery time. Adequate writer QoS for this use case are presented in [Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer](#) below.

Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer

```

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->protocol.push_on_write = DDS_BOOLEAN_TRUE;
4.
5. //use these hard coded value unless you use a key
6. qos->resource_limits.initial_samples = qos->resource_limits.max_samples = 1;
7. qos->resource_limits.max_samples_per_instance =
8. qos->resource_limits.max_samples;
9. qos->resource_limits.initial_instances =
10. qos->resource_limits.max_instances = 1;
11.
12. // want to piggyback HB w/ every sample.
13. qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples =
14. qos->resource_limits.max_samples;
15.
16. qos->protocol.rtps_reliable_writer.high_watermark = 1;
17. qos->protocol.rtps_reliable_writer.low_watermark = 0;
18. qos->protocol.rtps_reliable_writer.min_ack_response_delay.sec = 0;
19. qos->protocol.rtps_reliable_writer.min_ack_response_delay.nanosec = 0;
20. //consider making non-zero for reliable multicast
21. qos->protocol.rtps_reliable_writer.max_ack_response_delay.sec = 0;
22. qos->protocol.rtps_reliable_writer.max_ack_response_delay.nanosec = 0;
23.
24. // should be faster than the send rate, but be mindful of OS resolution
25. 25 qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
26. 26 qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =

```

```

27. alertReaderWithinThisMs * 1000000;
28.
29. qos->reliability.max_blocking_time = blockingTime;
30. qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 7;
31.
32. // essentially turn off slow HB period
33. qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;

```

[Line 1 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page\)](#): This is the default setting for a writer, shown here strictly for clarity.

[Line 2 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page\)](#): Setting the History kind to KEEP_ALL guarantees that no DDS sample is ever lost.

[Line 3 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page\)](#): This is the default setting for a writer, shown here strictly for clarity. ‘Push’ mode reliability will yield lower latency than ‘pull’ mode reliability in normal situations where there is no DDS sample loss. (See [DATA_WRITER_PROTOCOL QoS Policy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#).) Furthermore, it does not matter that each packet sent in response to a command will be small, because our data sent with each command is likely to be small, so that maximizing throughput for this data is not a concern.

[Line 5 - Line 10 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page\)](#): For this example, we assume a single writer is writing DDS samples one at a time. If we are not using keys (see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#)), there is no reason to use a queue with room for more than one DDS sample, because we want to resolve a DDS sample completely before moving on to the next. While this negatively impacts throughput, it minimizes memory usage. In this example, a written DDS sample will remain in the queue until it is acknowledged by all active readers (only 1 for this example).

[Line 12 - Line 14 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page\)](#): The fastest way for a writer to ensure that a reader is up-to-date is to force an acknowledgment with every DDS sample. We do this by appending a Heartbeat with every DDS sample. This is akin to a certified mail; the writer learns—as soon as the system will allow—whether a reader has received the letter, and can take corrective action if the reader has not. As with certified mail, this model has significant overhead compared to the unreliable case, trading off lower packet efficiency in favor of latency and fast recovery.

[Line 16-Line 17 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page\)](#): Since the writer takes responsibility for pushing the DDS samples out to the reader, a writer will go into a “heightened alert” mode as soon as the high water mark is reached (which is when any DDS sample is written for this writer) and only come out of this mode when the low water mark is reached (when all DDS samples have been acknowledged for this writer). Note that the selected high and low watermarks are actually the default values.

[Line 18-Line 22 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on page 655\)](#): When a reader requests a lost DDS sample, we respond to the reader immediately in the interest of faster recovery. If the readers receive packets on unicast, there is no reason to wait, since the writer will eventually have to feed individual readers separately anyway. In case of multicast readers, it makes sense to consider further. If the writer delayed its response enough so that all or most of the readers have had a chance to NACK a DDS sample, the writer may coalesce the requests and send just one packet to all the multicast readers. Suppose that all multicast readers do indeed NACK within approximately 100 μ sec. Setting the minimum and maximum delays at 100 μ sec will allow the writer to collect all these NACKs and send a single response over multicast. (See [DATA_WRITER_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#) for information on setting `min_nack_response_delay` and `max_nack_response_delay`.) Note that Connex DDS relies on the OS to wait for this 100 μ sec. Unfortunately, not all operating systems can sleep for such a fine duration. On Windows systems, for example, the minimum achievable sleep time is somewhere between 1 to 20 milliseconds, depending on the version. On VxWorks systems, the minimum resolution of the wait time is based on the tick resolution, which is 1/system clock rate (thus, if the system clock rate is 100 Hz, the tick resolution is 10 millisecond). On such systems, the achievable minimum wait is actually far larger than the desired wait time. This could have an unintended consequence due to the delay caused by the OS; at a minimum, the time to repair a packet may be longer than you specified.

[Line 24-Line 27 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on page 655\)](#): If a reader drops a DDS sample, the writer recovers by notifying the reader of what it has sent, so that the reader may request resending of the lost DDS sample. Therefore, the recovery time depends primarily on how quickly the writer pings the reader that has fallen behind. If commands will not be generated faster than one every few seconds, it may be acceptable for the writer to ping the reader several hundred milliseconds after the DDS sample is sent.

- Suppose that the round-trip time of fairly small packets between the writer and the reader application is 50 microseconds, and that the reader does not delay response to a Heartbeat from the writer (see [DATA_READER_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 7.6.1 on page 511\)](#) for how to change this). If a DDS sample is dropped, the writer will ping the reader after a maximum of the OS delay resolution discussed above and `alertReaderWithinThisMs` (let's say 10 ms for this example). The reader will request the missing DDS sample immediately, and with the code set as above, the writer will feed the missing DDS sample immediately. Neglecting the processing time on the writer or the reader end, and assuming that this retry succeeds, the time to recover the DDS sample from the original publication time is: `alertReaderWithinThisMs + 50 μ sec + 25 μ sec`.

If the OS is capable of micro-sleep, the recovery time can be within 100 μ sec, barely noticeable to a human operator. If the OS minimum wait resolution is much larger, the recovery time is dominated by the wait resolution of the OS. Since ergonomic studies suggest that delays in excess of a 0.25 seconds start hampering operations that require low latency data, even a 10 ms limitation seems to be acceptable.

- What if two packets are dropped in a row? Then the recovery time would be $2 * \text{alertReaderWithinThisMs} + 2 * 50 \mu\text{sec} + 25 \mu\text{sec}$. If **alertReaderWithinThisMs** is 100 ms, the recovery time now exceeds 200 ms, and can perhaps degrade user experience.

[Line 29-Line 30 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on page 655\)](#): What if another command (like another button press) is issued before the recovery? Since we must not drop this new DDS sample, we block the writer until the recovery completes. If **alertReaderWithinThisMs** is 10 ms, and we assume no more than 7 consecutive drops, the longest time for recovery will be just above (**alertReaderWithinThisMs * max_heartbeat_retries**), or 70 ms.

So if we set **blockingTime** to about 80 ms, we will have given enough chance for recovery. Of course, in a dynamic system, a reader may drop out at any time, in which case **max_heartbeat_retries** will be exceeded, and the unresponsive reader will be dropped by the writer. In either case, the writer can continue writing. Inappropriate values will cause a writer to prematurely drop a temporarily unresponsive (but otherwise healthy) reader, or be stuck trying unsuccessfully to feed a crashed reader. In the unfortunate case where a reader becomes temporarily unresponsive for a duration exceeding (**alertReaderWithinThisMs * max_heartbeat_retries**), the writer may issue gaps to that reader when it becomes active again; the dropped DDS samples are irrecoverable. So estimating the worst case unresponsive time of all potential readers is critical if DDS sample drop is unacceptable.

[Line 33 \(Figure 10.10 QoS for an Aperiodic, One-at-a-time Reliable Writer on page 655\)](#): Since the command may not be issued for hours or even days on end, there is no reason to keep announcing the writer's state to the readers.

[Figure 10.11 QoS for an Aperiodic, One-at-a-time Reliable Reader below](#) shows how to set the QoS for the reader side, followed by a line-by-line explanation.

Figure 10.11 QoS for an Aperiodic, One-at-a-time Reliable Reader

```

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3.
4. // 1 is ok for normal use. 2 allows fast infinite loop
5. qos->reader_resource_limits.max_samples_per_remote_writer = 2;
6. qos->resource_limits.initial_samples = 2;
7. qos->resource_limits.initial_instances = 1;
8.
9. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
10. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec =
    0;
11. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;

```

```
12. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec =
0;
```

[Line 1-Line 2 \(Figure 10.11 QoS for an Aperiodic, One-at-a-time Reliable Reader on the previous page\)](#): Unlike a writer, the reader's default reliability setting is best-effort, so reliability must be turned on. Since we don't want to drop anything, we choose KEEP_ALL history.

[Line 4-Line 6 \(Figure 10.11 QoS for an Aperiodic, One-at-a-time Reliable Reader on the previous page\)](#): Since we enforce reliability on each DDS sample, it would be sufficient to keep the queue size at 1, except in the following case: suppose that the reader takes some action in response to the command received, which in turn causes the writer to issue another command right away. Because Connex DDS passes the user data up to the application even before acknowledging the DDS sample to the writer (for minimum latency), the first DDS sample is still pending for acknowledgement in the writer's queue when the writer attempts to write the second DDS sample, and will cause the writing thread to block until the reader completes processing the first DDS sample and acknowledges it to the writer; all are as they should be. But if you want to run this infinite loop at full throttle, the reader should buffer one more DDS sample. Let's follow the packets flow under a normal circumstance:

1. The sender application writes DDS sample 1 to the reader. The receiver application processes it and sends a user-level response 1 to the sender application, but has not yet ACK'd DDS sample 1.
2. The sender application writes DDS sample 2 to the receiving application in response to response 1. Because the reader's queue is 2, it can accept DDS sample 2 even though it may not yet have acknowledged DDS sample 1. Otherwise, the reader may drop DDS sample 2, and would have to recover it later.
3. At the same time, the receiver application acknowledges DDS sample 1, and frees up one slot in the queue, so that it can accept DDS sample 3, which it on its way.

The above steps can be repeated ad-infinitum in a continuous traffic.

[Line 7 \(Figure 10.11 QoS for an Aperiodic, One-at-a-time Reliable Reader on the previous page\)](#): Since we are not using keys, there is just one instance.

[Line 9-Line 12 \(Use Cases \(Section 10.3.7 on page 654\)\)](#): We choose immediate response in the interest of fastest recovery. In high throughput, multicast scenario, delaying the response (with event thread priority set high of course) may decrease the likelihood of NACK storm causing a writer to drop some NACKs. This random delay reduces this chance by staggering the NACK response. But the minimum delay achievable once again depends on the OS.

10.3.7.3 Aperiodic, Bursty

Suppose you have aperiodically generated bursts of data, as in the case of a new aircraft approaching an airport. The data may be the same or different, but if they are written by a single writer, the challenge to this writer is to feed all readers as quickly and efficiently as possible when this burst of hundreds or thousands of DDS samples hits the system.

If you use an unreliable writer to push this burst of data, some of them may be dropped over an unreliable transport such as UDP.

If you try to shape the burst according to however much the slowest reader can process, the system throughput may suffer, and places an additional burden of queueing the DDS samples on the sender application.

If you push the data reliably as fast they are generated, this may cost dearly in repair packets, especially to the slowest reader, which is already burdened with application chores.

Connex DDS pull mode reliability offers an alternative in this case by letting each reader pace its own data stream. It works by notifying the reader what it is missing, then waiting for it to request only as much as it can handle. As in the aperiodic one-at-a-time case ([Aperiodic Use Case: One-at-a-Time \(Section 10.3.7.2 on page 655\)](#)), multicast is supported, but its performance depends on the resolution of the minimum delay supported by the OS. At the cost of greater latency, this model can deliver reliability while using far fewer packets than in the push mode. The writer QoS is given in [Figure 10.12 QoS for an Aperiodic, Bursty Writer below](#), with a line-by-line explanation below.

Figure 10.12 QoS for an Aperiodic, Bursty Writer

```

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->protocol.push_on_write = DDS_BOOLEAN_FALSE;
4.
5. //use these hard coded value until you use key
6. qos->resource_limits.initial_instances =
7. qos->resource_limits.max_instances = 1;
8. qos->resource_limits.initial_samples = qos->resource_limits.max_samples
9.           = worstBurstInSample;
10. qos->resource_limits.max_samples_per_instance =
11. qos->resource_limits.max_samples;
12.
13. // piggyback HB not used
14. qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples = 0;
15.
16. qos->protocol.rtps_reliable_writer.high_watermark = 1;
17. qos->protocol.rtps_reliable_writer.low_watermark = 0;
18.
19. qos->protocol.rtps_reliable_writer.min_nack_response_delay.sec = 0;

```

```

20. qos->protocol.rtps_reliable_writer.min_ack_response_delay.nanosec = 0;
21. qos->protocol.rtps_reliable_writer.max_ack_response_delay.sec = 0;
22. qos->protocol.rtps_reliable_writer.max_ack_response_delay.nanosec = 0;
23. qos->reliability.max_blocking_time = blockingTime;
24.
25. // should be faster than the send rate, but be mindful of OS resolution
26. qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
27. qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =
28.             alertReaderWithinThisMs * 1000000;
29. qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 5;
30.
31. // essentially turn off slow HB period
32. qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;

```

[Line 1 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on the previous page\)](#): This is the default setting for a writer, shown here strictly for clarity.

[Line 2 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on the previous page\)](#): Since we do not want any data lost, we want the History kind set to KEEP_ALL.

[Line 3 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on the previous page\)](#): The default Connex DDS reliable writer will push, but we want the reader to pull instead.

[Line 5-Line 11 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on the previous page\)](#): We assume a single instance, in which case the maximum DDS sample count will be the same as the maximum DDS sample count per writer. In contrast to the one-at-a-time case discussed in [Aperiodic Use Case: One-at-a-Time \(Section 10.3.7.2 on page 655\)](#), the writer's queue is large; as big as the burst size in fact, but no more because this model tries to resolve a burst within a reasonable period, to be computed shortly. Of course, we could block the writing thread in the middle of the burst, but that might complicate the design of the sending application.

[Line 13-Line 14 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on the previous page\)](#): By a 'piggy-back' Heartbeat, we mean only a Heartbeat that is appended to data being pushed from the writer. Strictly speaking, the writer will also append a Heartbeat with each reply to a reader's lost DDS sample request, but we call that a 'framing' Heartbeat. Since data is pulled, **heartbeats_per_max_samples** is ignored.

[Line 16-Line 17 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on the previous page\)](#): Similar to the previous aperiodic writer, this writer spends most of its time idle. But as the name suggests, even a single new DDS sample implies more DDS sample to follow in a burst. Putting the writer into a fast mode quickly will allow readers to be notified soon. Only when all DDS samples have been delivered, the writer can rest.

[Line 19- Line 23 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on page 660\)](#): Similar to the one-at-a-time case, there is no reason to delay response with only one reader. In this case, we can estimate the time to resolve a burst with only a few parameters. Let's say that the reader figures it can safely receive and process 20 DDS samples at a time without being overwhelmed, and that the time it takes a writer to fetch these 20 DDS samples and send a single packet containing these 20 DDS samples, plus the time it takes a reader to receive and process these DDS samples, and send another request back to the writer for the next 20 DDS samples is 11 ms. Even on the same hardware, if the reader's processing time can be reduced, this time will decrease; other factors such as the traversal time through Connex DDS and the transport are typically in microseconds range (depending on machines of course).

For example, let's also say that the worst case burst is 1000 DDS samples. The writing thread will of course not block because it is merely copying each of the 1000 DDS samples to the Connex DDS queue on the writer side; on a typical modern machine, the act of writing these 1000 DDS samples will probably take no more than a few ms. But it would take at least $1000/20 = 50$ resend packets for the reader to catch up to the writer, or 50 times 11 ms = 550 ms. Since the burst model deals with one burst at a time, we would expect that another burst would not come within this time, and that we are allowed to block for at least this period. Including a safety margin, it would appear that we can comfortably handle a burst of 1000 every second or so.

But what if there are multiple readers? The writer would then take more time to feed multiple readers, but with a fast transport, a few more readers may only increase the 11 ms to only 12 ms or so. Eventually, however, the number of readers will justify the use of multicast. Even in pull mode, Connex DDS supports multicast by measuring how many multicast readers have requested DDS sample repair. If the writer does not delay response to NACK, then repairs will be sent in unicast. But a suitable NACK delay allows the writer to collect potentially NACKs from multiple readers, and feed a single multicast packet. But as discussed in [Aperiodic Use Case: One-at-a-Time \(Section 10.3.7.2 on page 655\)](#), by delaying reply to coalesce response, we may end up waiting much longer than desired. On a Windows system with 10 ms minimum sleep achievable, the delay would add at least 10 ms to the 11 ms delay, so that the time to push 1000 DDS samples now increases to 50 times 21 ms = 1.05 seconds. It would appear that we will not be able to keep up with incoming burst if it came at roughly 1 second, although we put fewer packets on the wire by taking advantage of multicast.

[Line 25-Line 28 \(Use Cases \(Section 10.3.7 on page 654\)\)](#): We now understand how the writer feeds the reader in response to the NACKs. But how does the reader realize that it is behind? The writer notifies the reader with a Heartbeat to kick-start the exchange. Therefore, the latency will be lower bound by the writer's fast heartbeat period. If the application is not particularly sensitive to latency, the minimum wait time supported by the OS (10 ms on Windows systems, for example) might be a reasonable value.

[Line 29 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on page 660\)](#): With a fast heartbeat period of 50 ms, a writer will take 500 ms (50 ms times the default `max_heartbeat_retries` of 10) to write-off an unresponsive reader. If a reader crashes while we are writing a lot of DDS samples per second, the writer queue may completely fill up before the writer has a chance to drop the crashed reader. Lowering `max_heartbeat_retries` will prevent that scenario.

[Line 31-Line 32 \(Figure 10.12 QoS for an Aperiodic, Bursty Writer on page 660\)](#): For an aperiodic writer, turning off slow periodic Heartbeats will remove unwanted traffic from the network.

[Figure 10.13 QoS for an Aperiodic, Bursty Reader below](#) shows example code for a corresponding aperiodic, bursty reader.

Figure 10.13 QoS for an Aperiodic, Bursty Reader

```

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->resource_limits.initial_samples =
4. qos->resource_limits.max_samples =
5. qos->reader_resource_limits.max_samples_per_remote_writer = 32;
6.
7. //use these hard coded value until you use key
8. qos->resource_limits.max_samples_per_instance =
9. qos->resource_limits.max_samples;
10. qos->resource_limits.initial_instances =
11. qos->resource_limits.max_instances = 1;
12.
13. // the writer probably has more for the reader; ask right away
14. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
15. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec =
16. 0;
16. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
17. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec =
17. 0;

```

[Line 1-Line 2 \(Figure 10.13 QoS for an Aperiodic, Bursty Reader above\)](#): Unlike a writer, the reader's default reliability setting is best-effort, so reliability must be turned on. Since we don't want to drop anything, we choose KEEP_ALL for the History QoS kind.

[Line 3-Line 5 \(Figure 10.13 QoS for an Aperiodic, Bursty Reader above\)](#): Unlike the writer, the reader's queue can be kept small, since the reader is free to send ACKs for as much as it wants anyway. In general, the larger the queue, the larger the packet needs to be, and the higher the throughput will be. When the reader NACKs for lost DDS sample, it will only ask for this much.

[Line 7-Line 11 \(Figure 10.13 QoS for an Aperiodic, Bursty Reader above\)](#): We do not use keys in this example.

[Line 13-Line 17 \(Figure 10.13 QoS for an Aperiodic, Bursty Reader on the previous page\)](#): We respond immediately to catch up as soon as possible. When there are many readers, this may cause a NACK storm, as discussed in the reader code for one-at-a-time reliable reader.

10.3.7.4 Periodic

In a periodic reliable model, we can use the writer and the reader queue to keep the data flowing at a smooth rate. The data flows from the sending application to the writer queue, then to the transport, then to the reader queue, and finally to the receiving application. Unless the sending application or any one of the receiving applications becomes unresponsive (including a crash) for a noticeable duration, this flow should continue uninterrupted.

The latency will be low in most cases, but will be several times higher for the recovered and many subsequent DDS samples. In the event of a disruption (e.g., loss in transport, or one of the readers becoming temporarily unresponsive), the writer's queue level will rise, and may even block in the worst case. If the writing thread must not block, the writer's queue must be sized sufficiently large to deal with any fluctuation in the system. [Figure 10.14 QoS for a Periodic Reliable Writer below](#) shows an example, with line-by-line analysis below.

Figure 10.14 QoS for a Periodic Reliable Writer

```

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->protocol.push_on_write = DDS_BOOLEAN_TRUE;
4.
5. //use these hard coded value until you use key
6. qos->resource_limits.initial_instances =
7. qos->resource_limits.max_instances = 1;
8.
9. int unresolvedSamplePerRemoteWriterMax =
10.     worstCaseApplicationDelayTimeInMs * dataRateInHz / 1000;
11. qos->resource_limits.max_samples = unresolvedSamplePerRemoteWriterMax;
12. qos->resource_limits.initial_samples = qos->resource_limits.max_samples/2;
13. qos->resource_limits.max_samples_per_instance =
14.     qos->resource_limits.max_samples;
15.
16. int piggybackEvery = 8;
17. qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples =

```

```

18.     qos->resource_limits.max_samples / piggybackEvery;
19.
20.     qos->protocol.rtps_reliable_writer.high_watermark = piggybackEvery * 4;
21.     qos->protocol.rtps_reliable_writer.low_watermark = piggybackEvery * 2;
22.     qos->reliability.max_blocking_time = blockingTime;
23.
24.     qos->protocol.rtps_reliable_writer.min_ack_response_delay.sec = 0;
25.     qos->protocol.rtps_reliable_writer.min_ack_response_delay.nanosec = 0;
26.
27.     qos->protocol.rtps_reliable_writer.max_ack_response_delay.sec = 0;
28.     qos->protocol.rtps_reliable_writer.max_ack_response_delay.nanosec = 0;
29.
30.     qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
31.     qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =
32.     `  alertReaderWithinThisMs * 1000000;
33.     qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 7;
34.
35.     // essentially turn off slow HB period
36.     qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;

```

Line 1 ([Figure 10.14 QoS for a Periodic Reliable Writer on the previous page](#)): This is the default setting for a writer, shown here strictly for clarity.

Line 2 ([Figure 10.14 QoS for a Periodic Reliable Writer on the previous page](#)): Since we do not want any data lost, we set the History kind to KEEP_ALL.

Line 3 ([Figure 10.14 QoS for a Periodic Reliable Writer on the previous page](#)): This is the default setting for a writer, shown here strictly for clarity. Pushing will yield lower latency than pulling.

Line 5-Line 7 ([Figure 10.14 QoS for a Periodic Reliable Writer on the previous page](#)): We do not use keys in this example, so there is only one instance.

Line 9-Line 11 ([Figure 10.14 QoS for a Periodic Reliable Writer on the previous page](#)): Though a simplistic model of queue, this is consistent with the idea that the queue size should be proportional to the data rate and the worst case jitter in communication.

Line 12 ([Figure 10.14 QoS for a Periodic Reliable Writer on the previous page](#)): Even though we have sized the queue according to the worst case, there is a possibility for saving some memory in the normal case. Here, we initially size the queue to be only half of the worst case, hoping that the worst case will not occur. When it does, Connext DDS will keep increasing the queue size as necessary to accommodate new DDS samples, until the maximum is reached. So when our optimistic initial queue size is breached, we

will incur the penalty of dynamic memory allocation. Furthermore, you will wind up using more memory, as the initially allocated memory will be orphaned (note: does not mean a memory leak or dangling pointer); if the initial queue size is M_i and the maximal queue size is M_m , where $M_m = M_i * 2^n$, the memory wasted in the worst case will be $(M_m - 1) * \text{sizeof(DDS sample)}$ bytes. Note that the memory allocation can be avoided by setting the initial queue size equal to its max value.

[Line 13-Line 14 \(Figure 10.14 QoS for a Periodic Reliable Writer on page 664\)](#): If there is only one instance, maximum DDS samples per instance is the same as maximum DDS samples allowed.

[Line 16-Line 18 \(Figure 10.14 QoS for a Periodic Reliable Writer on page 664\)](#): Since we are pushing out the data at a potentially rapid rate, the piggyback heartbeat will be useful in letting the reader know about any missing DDS samples. The **piggybackEvery** can be increased if the writer is writing at a fast rate, with the cost that more DDS samples will need to queue up for possible resend. That is, you can consider the piggyback heartbeat to be taking over one of the roles of the periodic heartbeat in the case of a push. So sending fewer DDS samples between piggyback heartbeats is akin to decreasing the fast heartbeat period seen in previous sections. Please note that we cannot express **piggybackEvery** directly as its own QoS, but indirectly through the maximum DDS samples.

[Line 20-Line 22 \(Figure 10.14 QoS for a Periodic Reliable Writer on page 664\)](#): If **piggybackEvery** was exactly identical to the fast heartbeat, there would be no need for fast heartbeat or the high watermark. But one of the important roles for the fast heartbeat period is to allow a writer to abandon inactive readers before the queue fills. If the high watermark is set equal to the queue size, the writer would not doubt the status of an unresponsive reader until the queue completely fills—blocking on the next write (up to **blockingTime**). By lowering the high watermark, you can control how vigilant a writer is about checking the status of unresponsive readers. By scaling the high watermark to **piggybackEvery**, the writer is expressing confidence that an alive reader will respond promptly within the time it would take a writer to send 4 times **piggybackEvery** DDS samples. If the reader does not delay the response too long, this would be a good assumption. Even if the writer estimated on the low side and does go into fast mode (suspecting that the reader has crashed) when a reader is temporarily unresponsive (e.g., when it is performing heavy computation for a few milliseconds), a response from the reader in question will resolve any doubt, and data delivery can continue uninterrupted. As the reader catches up to the writer and the queue level falls below the low watermark, the writer will pop out to the normal, relaxed mode.

[Line 24-Line 28 \(Figure 10.14 QoS for a Periodic Reliable Writer on page 664\)](#): When a reader is behind (including a reader whose Durability QoS is non-VOLATILE and therefore needs to catch up to the writer as soon as it is created), how quickly the writer responds to the reader's request will determine the catch-up rate. While a multicast writer (that is, a writer with multicast readers) may consider delaying for some time to take advantage of coalesced multicast packets. Keep in mind the OS delay resolution issue discussed in the previous section.

[Line 30-Line 33 \(Figure 10.14 QoS for a Periodic Reliable Writer on page 664\)](#): The fast heartbeat mechanism allows a writer to detect a crashed reader and move along with the remaining readers when a reader does not respond to any of the **max_heartbeat_retries** number of heartbeats sent at the **fast_heartbeat_**

period rate. So if you want a more cautious writer, decrease either numbers; conversely, increasing either number will result in a writer that is more reluctant to write-off an unresponsive reader.

[Line 35-Line 36 \(Figure 10.14 QoS for a Periodic Reliable Writer on page 664\)](#): Since this a periodic model, a separate periodic heartbeat to notify the writer’s status would seem unwarranted; the piggyback heartbeat sent with DDS samples takes over that role.

[Figure 10.15 QoS for a Periodic Reliable Reader below](#) shows how to set the QoS for a matching reader, followed by a line-by-line explanation.

Figure 10.15 QoS for a Periodic Reliable Reader

```

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->resource_limits.initial_samples =
4. qos->resource_limits.max_samples =
5. qos->reader_resource_limits.max_samples_per_remote_writer =
6.   ((2*piggybackEvery - 1) + dataRateInHz * delayInMs / 1000);
7.
8. //use these hard coded value until you use key
9. qos->resource_limits.max_samples_per_instance =
10.    qos->resource_limits.max_samples;
11. qos->resource_limits.initial_instances =
12.    qos->resource_limits.max_instances = 1;
13.
14. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
15. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec =
16. 0;
16. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
17. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec =
18. 0;

```

[Line 1-Line 2 \(Figure 10.15 QoS for a Periodic Reliable Reader above\)](#): Unlike a writer, the reader’s default reliability setting is best-effort, so reliability must be turned on. Since we don’t want to drop anything, we choose KEEP_ALL for the History QoS.

[Line 3-Line 6 \(Figure 10.15 QoS for a Periodic Reliable Reader above\)](#) Unlike the writer, the reader queue is sized not according to the jitter of the reader, but rather how many DDS samples you want to cache speculatively in case of a gap in sequence of DDS samples that the reader must recover. Remember

that a reader will stop giving a sequence of DDS samples as soon as an unintended gap appears, because the definition of strict reliability includes in-order delivery. If the queue size were 1, the reader would have no choice but to drop all subsequent DDS samples received until the one being sought is recovered. Connext DDS uses speculative caching, which minimizes the disruption caused by a few dropped DDS samples. Even for the same duration of disruption, the demand on reader queue size is greater if the writer will send more rapidly. In sizing the reader queue, we consider 2 factors that comprise the lost DDS sample recovery time:

- How long it takes a reader to request a resend to the writer.

The piggyback heartbeat tells a reader about the writer's state. If only DDS samples between two piggybacked DDS samples are dropped, the reader must cache **piggybackEvery** DDS samples before asking the writer for resend. But if a piggybacked DDS sample is also lost, the reader will not get around to asking the writer until the next piggybacked DDS sample is received. Note that in this worst case calculation, we are ignoring stand-alone heartbeats (i.e., not piggybacked heartbeat from the writer). Of course, the reader may drop any number of heartbeats, including the stand-alone heartbeat; in this sense, there is no such thing as the absolute worst case—just reasonable worst case, where the probability of consecutive drops is acceptably low. For the majority of applications, even two consecutive drops is unlikely, in which case we need to cache at most $(2 * \text{piggybackEvery} - 1)$ DDS samples before the reader will ask the writer to resend, assuming no delay ([Line 14-Line 17](#), [Figure 10.15 QoS for a Periodic Reliable Reader on the previous page](#)).

- How long it takes for the writer to respond to the request.

Even ignoring the flight time of the resend request through the transport, the writer takes a finite time to respond to the repair request—mostly if the writer delays reply for multicast readers. In case of immediate response, the processing time on the writer end, as well as the flight time of the messages to and from the writer do not matter unless very larger data rate; that is, it is the product term that matters. In case the delay for multicast is random (that is, the minimum and the maximum delay are not equal), one would have to use the maximum delay to be conservative.

[Line 8-Line 12](#) ([Figure 10.15 QoS for a Periodic Reliable Reader on the previous page](#)): Since we are not using keys, there is just one instance.

[Line 14-Line 17](#) ([Figure 10.15 QoS for a Periodic Reliable Reader on the previous page](#)): If we are not using multicast, or the number of readers being fed by the writer, there is no reason to delay.

10.4 Auto Throttling for DataWriter Performance—Experimental Feature

Auto Throttling is an experimental feature that allows you to configure a *DataWriter* to automatically adjust its writing rate and send window size to provide the best latency/throughput tradeoff as system conditions change.

When *DataWriters* and *DataReaders* are configured to be reliable, lost DDS samples are repaired automatically by *Connex DDS*. However, the repair path consumes bandwidth and increases latency. A high number of lost DDS samples can reduce the throughput and increase the communication latency. With Auto Throttling, the number of repair (lost) DDS samples is reduced by using feedback provided by *DataReaders* in terms of ACK and NACK messages to adjust the *DataWriter's* write rate and send window size.

To configure Auto Throttling, use the following properties:

dds.domain_participant.auto_throttle.enable: Configures the *DomainParticipant* to gather internal measurements (during *DomainParticipant* creation) that are required for the Auto Throttle feature. This allows *DataWriters* belonging to this *DomainParticipant* to use the Auto Throttle feature. Default: false.

dds.data_writer.auto_throttle.enable: Enables automatic throttling in the *DataWriter* so it can automatically adjust the writing rate and the send window size; this minimizes the need for repair DDS samples and improves latency. Default: false.

Note: This property takes effect only in *DataWriters* that belong to a *DomainParticipant* that has set the property **dds.domain_participant.auto_throttle.enable** (described above) to true.

When Auto throttling is enabled, the size of the send window size is adjusted within the interval [**min_send_window_size**, **max_send_window_size**] configured in [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347)

Chapter 11 Collaborative DataWriters

The *Collaborative DataWriters* feature allows you to have multiple *DataWriters* publishing DDS samples from a common logical data source. The *DataReaders* will combine the DDS samples coming from these *DataWriters* in order to reconstruct the correct order in which they were produced at the source. This combination process for the *DataReaders* can be configured using the [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#). It requires the middleware to provide a way to uniquely identify every DDS sample published in a DDS domain independently of the actual *DataWriter* that published the DDS sample.

In *Connex DDS*, every modification (DDS sample) to the global dataspace made by a *DataWriter* within a DDS domain is identified by a pair (virtual GUID, sequence number).

The virtual GUID (Global Unique Identifier) is a 16-byte character identifier associated with the logical data source. DataWriters can be assigned a virtual GUID using **virtual_guid** in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#).

The virtual sequence number is a 64-bit integer that identifies changes within the logical data source.

Several *DataWriters* can be configured with the same virtual GUID. If each of these *DataWriters* publishes a DDS sample with sequence number '0', the DDS sample will only be received once by the *DataReaders* subscribing to the content published by the *DataWriters* (see [Figure 11.1 Global Dataspace Changes on the next page](#)).

Figure 11.1 Global Dataspace Changes



11.1 Collaborative DataWriters Use Cases

- Ordered delivery of DDS samples in high availability scenarios

One example of this is *RTI Persistence Service*¹. When a late-joining *DataReader* configured with [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) set to PERSISTENT or TRANSIENT joins a DDS domain, it will start receiving DDS samples from multiple *DataWriters*. For example, if the original *DataWriter* is still alive, the newly created *DataReader* will receive DDS samples from the original *DataWriter* and one or more *RTI Persistence Service DataWriters* (PRSTDataWriters).

- Ordered delivery of DDS samples in load-balanced scenarios

Multiple instances of the same application can work together to process and deliver DDS samples. When the DDS samples arrive through different data-paths out of order, the *DataReader* will be able to reconstruct the order at the source. An example of this is when multiple instances of *RTI Persistence Service* are used to persist the data. Persisting data to a database on disk can impact performance. By dividing the workload (e.g., DDS samples larger than 10 are persisted by Persistence Service 1, DDS samples smaller or equal to 10 are persisted by Persistence Service 2) across different instances of *RTI Persistence Service* using different databases the user can improve scalability and performance.

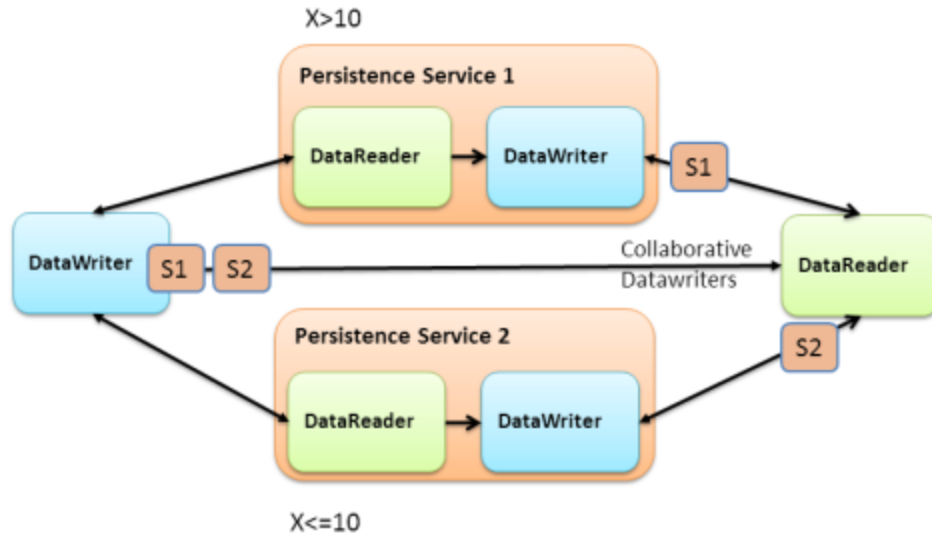
- Ordered delivery of DDS samples with Group Ordered Access

The Collaborative DataWriters feature can also be used to configure the DDS sample ordering process when the *Subscriber* is configured with [PRESENTATION QosPolicy \(Section 6.4.6 on page](#)

¹For more information on *Persistence Service*, see [Part 6: RTI Persistence Service \(Section on page 932\)](#).

330) **access_scope** set to GROUP. In this case, the *Subscriber* must deliver in order the DDS samples published by a group of *DataWriters* that belong to the same *Publisher* and have **access_scope** set to GROUP.

Figure 11.2 Load-Balancing with Persistence Service



11.2 DDS Sample Combination (Synchronization) Process in a DataReader

A *DataReader* will deliver a DDS sample (VGUID_n, VSN_m) to the application only when if one of the following conditions is satisfied:

- (VGUID_n, VSN_{m-1}) has already been delivered to the application.
- All the known *DataWriters* publishing VGUID_n have announced that they do not have (VGUID_n, VSN_{m-1}).
- None of the known *DataWriters* publishing VGUID_n have announced potential availability of (VGUID_n, VSN_{m-1}) and a configurable timeout (**max_data_availability_waiting_time**) expires.

For additional details on how the reconstruction process works see the [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#).

11.3 Configuring Collaborative DataWriters

11.3.1 Associating Virtual GUIDs with DDS Data Samples

There are two ways to associate a virtual GUID with the DDS samples published by a *DataWriter*.

- Per *DataWriter*: Using **virtual_guid** in [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#).
- Per DDS Sample: By setting the **writer_guid** in the identity field of the `WriteParams_t` structure provided to the **write_w_params** operation (see [Writing Data \(Section 6.3.8 on page 283\)](#)). Since the **writer_guid** can be set per DDS sample, the same *DataWriter* can potentially write DDS samples from independent logical data sources. One example of this is *RTI Persistence Service* where a single persistence service *DataWriter* can write DDS samples on behalf of multiple original *DataWriters*.

11.3.2 Associating Virtual Sequence Numbers with DDS Data Samples

You can associate a virtual sequence number with a DDS sample published by a *DataWriter* by setting the **sequence_number** in the **identity** field of the `WriteParams_t` structure provided to the **write_w_params** operation (see [Writing Data \(Section 6.3.8 on page 283\)](#)). Virtual sequence numbers for a given virtual GUID must be strictly monotonically increasing. If you try to write a DDS sample with a sequence number less than or equal to the last sequence number, the write operation will fail.

11.3.3 Specifying which DataWriters will Deliver DDS Samples to the DataReader from a Logical Data Source

The **required_matched_endpoint_groups** field in the [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#) can be used to specify the set of *DataWriter* groups that are expected to provide DDS samples for the same data source (virtual GUID). The quorum count in a group represents the number of *DataWriters* that must be discovered for that group before the *DataReader* is allowed to provide non-consecutive DDS samples to the application.

A *DataWriter* becomes a member of an endpoint group by configuring the **role_name** in [ENTITY_NAME QosPolicy \(DDS Extension\) \(Section 6.5.9 on page 374\)](#).

11.3.4 Specifying How Long to Wait for a Missing DDS Sample

A *DataReader's* [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#) specifies how long to wait for a missing DDS sample. For example, this is important when the first DDS sample is received: how long do you wait to determine the lowest sequence number available in the system?

- The **max_data_availability_waiting_time** defines how much time to wait before delivering a DDS sample to the application without having received some of the previous DDS samples.

- The **max_endpoint_availability_waiting_time** defines how much time to wait to discover *DataWriters* providing DDS samples for the same data source (virtual GUID).

11.4 Collaborative DataWriters and Persistence Service

The *DataWriters* created by persistence service are automatically configured to do collaboration:

- Every DDS sample published by the *Persistence Service DataWriter* keeps its original identity.
- *Persistence Service* associates the role name PERSISTENCE_SERVICE with all the *DataWriters* that it creates. You can overwrite that setting by changing the *DataWriter* QoS configuration in persistence service.

For more information, see [Part 6: RTI Persistence Service \(Section on page 932\)](#).

Chapter 12 Mechanisms for Achieving Information Durability and Persistence

12.1 Introduction

Connex DDS offers the following mechanisms for achieving durability and persistence:

- **Durable Writer History** This feature allows a *DataWriter* to persist its historical cache, perhaps locally, so that it can survive shutdowns, crashes and restarts. When an application restarts, each *DataWriter* that has been configured to have durable writer history automatically load all of the data in this cache from disk and can carry on sending data as if it had never stopped executing. To the rest of the system, it will appear as if the *DataWriter* had been temporarily disconnected from the network and then reappeared.
- **Durable Reader State** This feature allows a *DataReader* to persist its state and remember which data it has already received. When an application restarts, each *DataReader* that has been configured to have durable reader state automatically loads its state from disk and can carry on receiving data as if it had never stopped executing. Data that had already been received by the *DataReader* before the restart will be suppressed so that it is not even sent over the network.
- **Data Durability** This feature is a full implementation of the OMG DDS Persistence Profile. The [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) allows an application to configure a *DataWriter* so that the information written by the *DataWriter* survives beyond the lifetime of the *DataWriter*. In this manner, a late-joining *DataReader* can subscribe to and receive the information even after the *DataWriter* application is no longer executing. To use this feature, you need Persistence Service, a separate application described in [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#).

These features can be configured separately or in combination. To use Durable Writer State and Durable Reader State, you need a relational database, which is not included with Connex DDS. Supported databases are listed in the *Release Notes*. Persistence Service does not require a database when used in TRANSIENT mode (see [RTI Persistence Service \(Section 12.5.1 on page 692\)](#)) or in PERSISTENT mode with file-system storage (see [RTI Persistence Service \(Section 12.5.1 on page 692\)](#) and [Configuring Remote Administration \(Section 27.5 on page 942\)](#)).

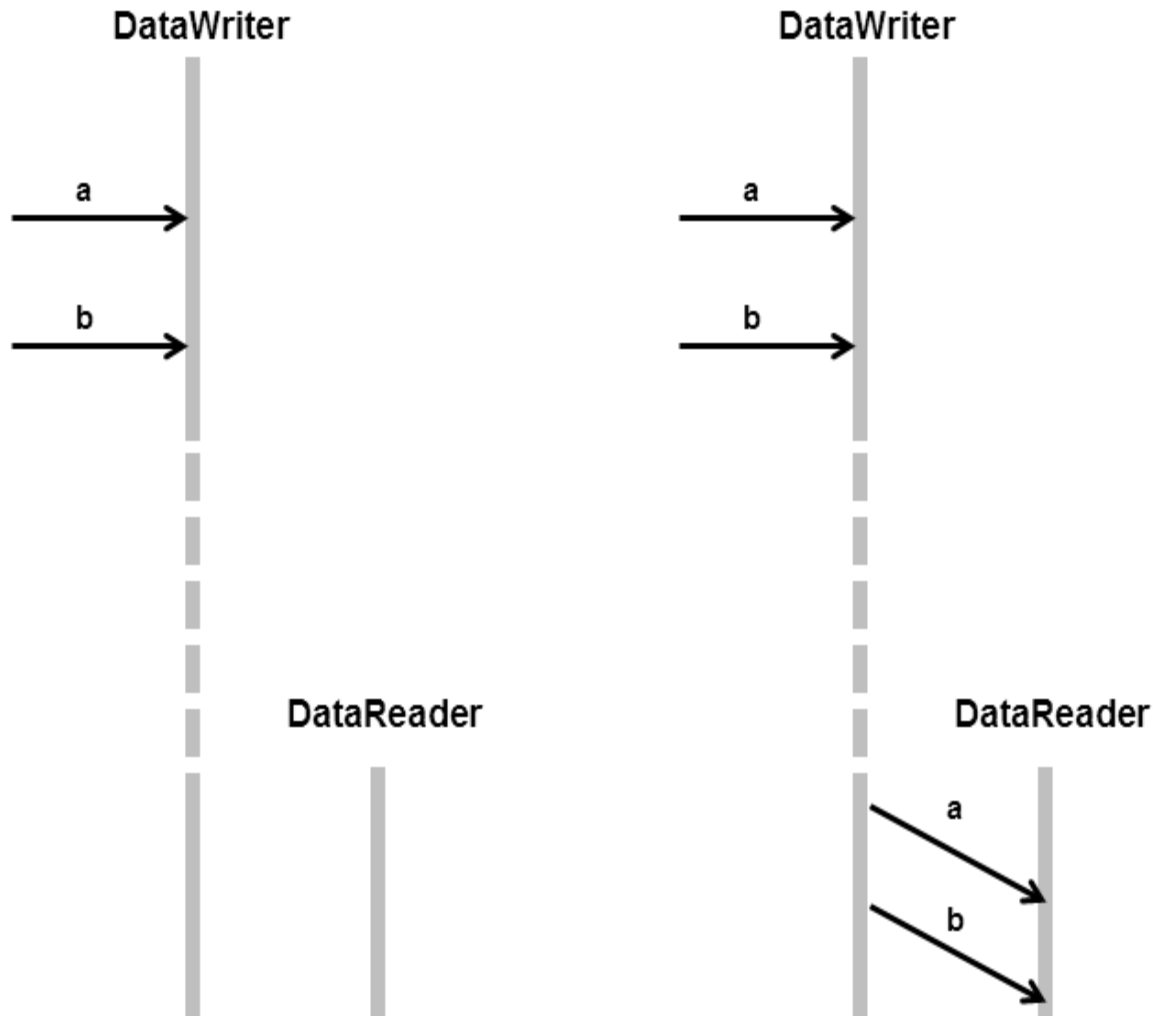
To understand how these features interact we will examine the behavior of the system using the following scenarios:

- [Scenario 1. DataReader Joins after DataWriter Restarts \(Durable Writer History\) \(Section 12.1.1 below\)](#)
- [Scenario 2: DataReader Restarts While DataWriter Stays Up \(Durable Reader State\) \(Section 12.1.2 on the facing page\)](#)
- [Scenario 3. DataReader Joins after DataWriter Leaves Domain \(Durable Data\) \(Section 12.1.3 on page 679\)](#)

12.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History)

In this scenario, a *DomainParticipant* joins the domain, creates a *DataWriter* and writes some data, then the *DataWriter* shuts down (gracefully or due to a fault). The *DataWriter* restarts and a *DataReader* joins the domain. Depending on whether the *DataWriter* is configured with durable history, the late-joining *DataReader* may or may not receive the data published already by the *DataWriter* before it restarted. This is illustrated in [Figure 12.1 Durable Writer History on the facing page](#). For more information, see [Durable Writer History \(Section 12.3 on page 681\)](#)

Figure 12.1 Durable Writer History



*Without Durable Writer History:
the late-joining DataReader will not
receive data (a and b) that was published
before the DataWriter's restart.*

*With Durable Writer History:
the restarted DataWriter will recover its
history and deliver its data to the late-
joining DataReader*

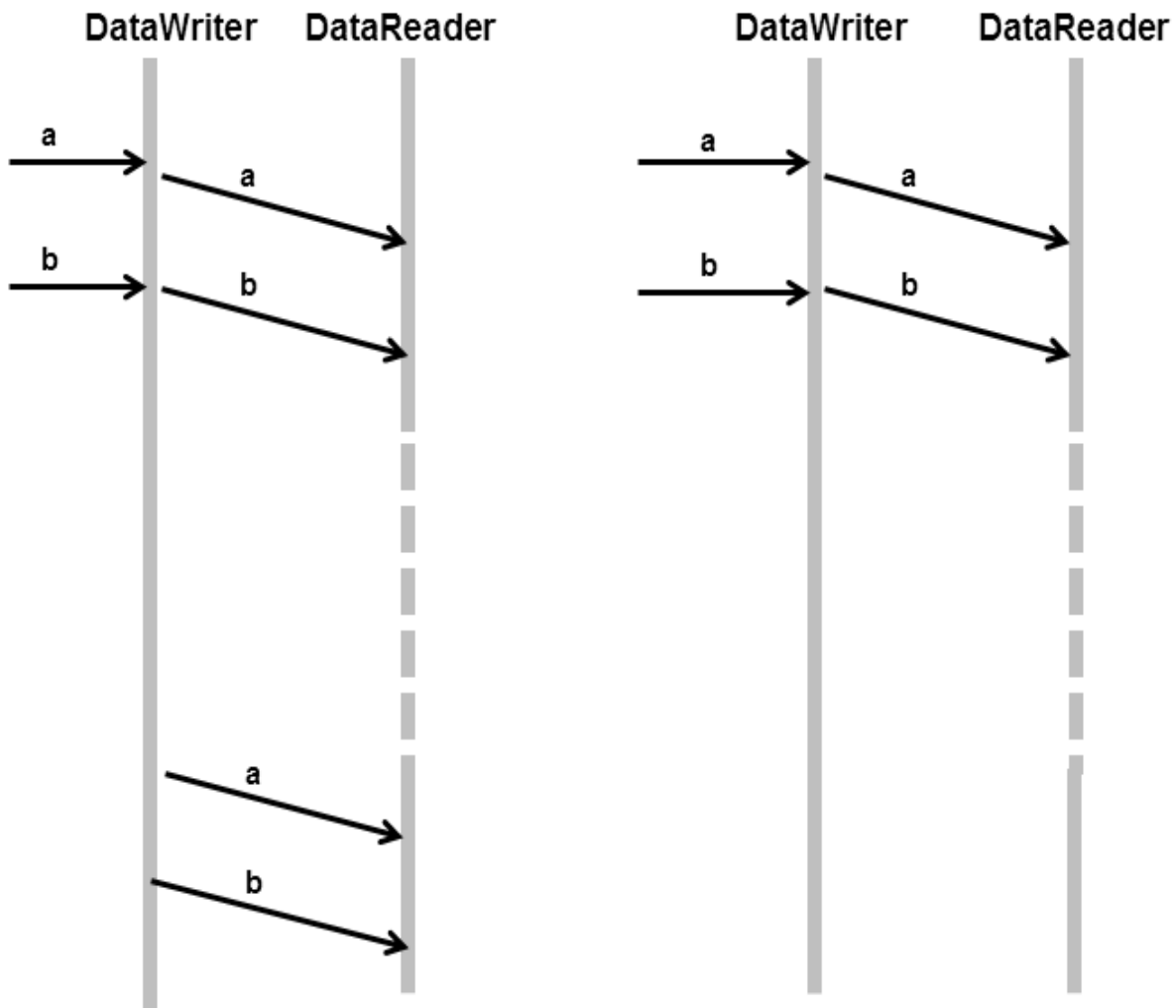
12.1.2 Scenario 2: DataReader Restarts While DataWriter Stays Up (Durable Reader State)

In this scenario, two *DomainParticipants* join a domain; one creates a *DataWriter* and the other a *DataReader* on the same Topic. The *DataWriter* publishes some data ("a" and "b") that is received by the

DataReader. After this, the *DataReader* shuts down (gracefully or due to a fault) and then restarts—all while the *DataWriter* remains present in the domain.

Depending on whether the *DataReader* is configured with Durable Reader State, the *DataReader* may or may not receive a duplicate copy of the data it received before it restarted. This is illustrated in [Figure 12.2 Durable Reader State](#) below. For more information, see [Durable Reader State \(Section 12.4 on page 686\)](#).

Figure 12.2 Durable Reader State



Without Durable Reader State:
the *DataReader* will receive the data that was already received before the restart.

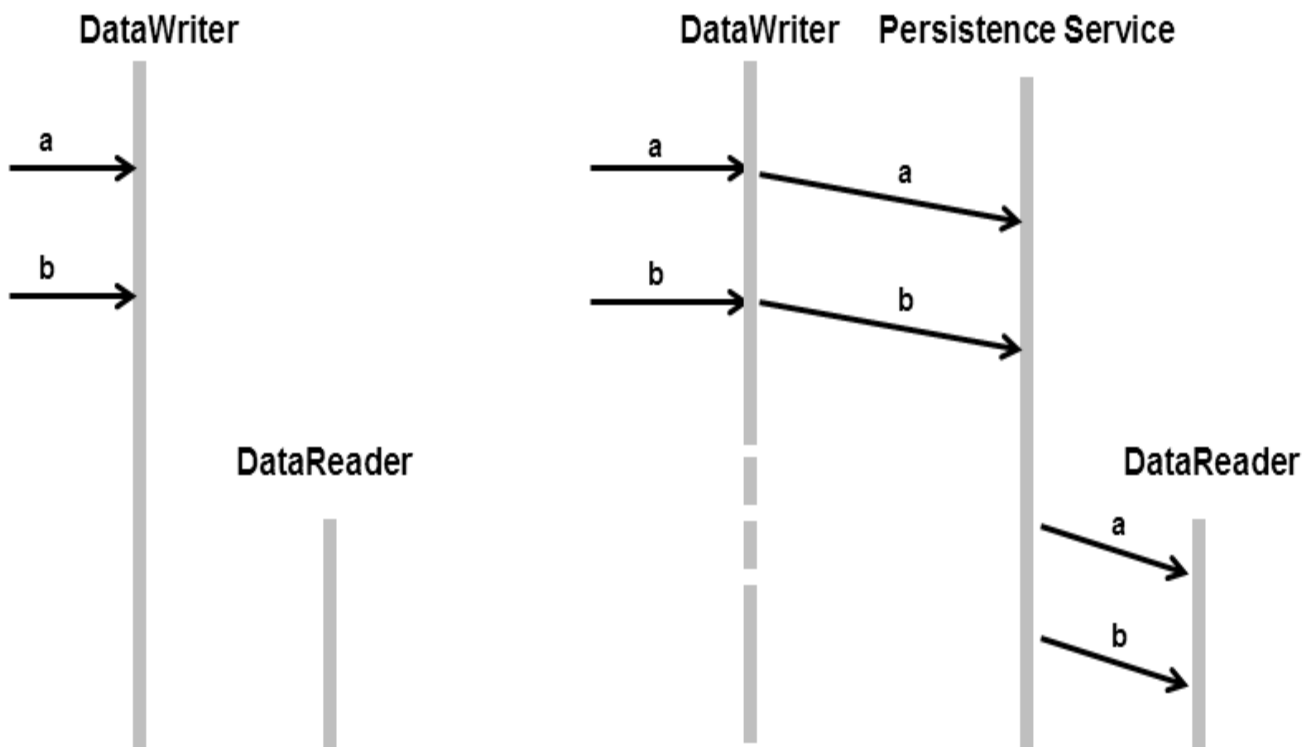
With Durable Reader State:
the *DataReader* remembers that it already received the data and does not request it again.

12.1.3 Scenario 3. DataReader Joins after DataWriter Leaves Domain (Durable Data)

In this scenario, a *DomainParticipant* joins a domain, creates a *DataWriter*, publishes some data on a Topic and then shuts down (gracefully or due to a fault). Later, a *DataReader* joins the domain and subscribes to the data. Persistence Service is running.

Depending on whether Durable Data is enabled for the Topic, the *DataReader* may or may not receive the data previous published by the *DataWriter*. This is illustrated in [Figure 12.3 Durable Data](#) below. For more information, see [Data Durability \(Section 12.5 on page 692\)](#)

Figure 12.3 Durable Data



Without Durable Data:
the late-joining DataReader
will not receive data (a and b)
that was published before the
DataWriter quit.

With Durable Data:
Persistence Service
remembers what data was
published and delivers it to
the late-joining DataReader.

This third scenario is similar to [Scenario 1. DataReader Joins after DataWriter Restarts \(Durable Writer History\)](#) (Section 12.1.1 on page 676) except that in this case the *DataWriter* does not need to restart for the *DataReader* to get the data previously written by the *DataWriter*. This is because Persistence Service acts as an intermediary that stores the data so it can be given to late-joining *DataReaders*.

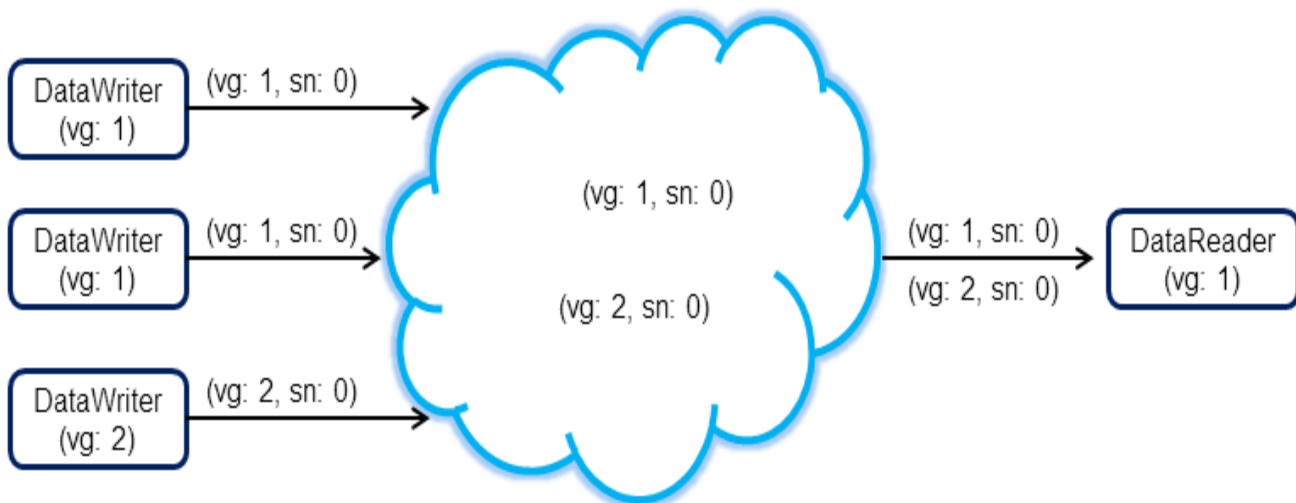
12.2 Durability and Persistence Based on Virtual GUIDs

Every modification to the global dataspace made by a *DataWriter* is identified by a pair (virtual GUID, sequence number).

- The virtual GUID (Global Unique Identifier) is a 16-byte character identifier associated with a *DataWriter* or *DataReader*; it is used to uniquely identify this entity in the global data space.
- The sequence number is a 64-bit identifier that identifies changes published by a specific *DataWriter*.

Several *DataWriters* can be configured with the same virtual GUID. If each of these *DataWriters* publishes a sample with sequence number '0', the sample will only be received once by the *DataReaders* subscribing to the content published by the *DataWriters* (see [Figure 12.4 Global Dataspace Changes](#) below).

Figure 12.4 Global Dataspace Changes

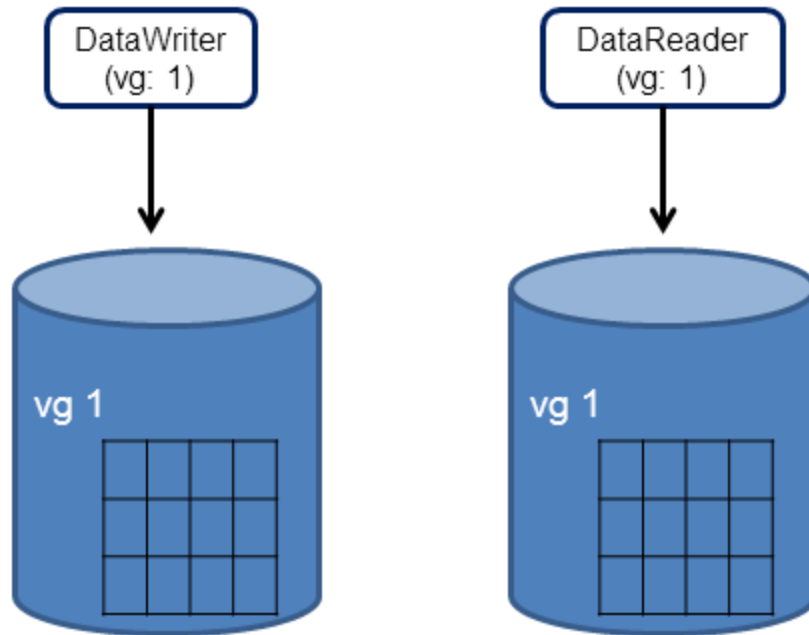


Additionally, Connex DDS uses the virtual GUID to associate a persisted state (state in permanent storage) to the corresponding *Entity*.

For example, the history of a *DataWriter* will be persisted in a database table with a name generated from the virtual GUID of the *DataWriter*. If the *DataWriter* is restarted, it must have associated the same virtual GUID to restore its previous history.

Likewise, the state of a *DataReader* will be persisted in a database table whose name is generated from the *DataReader* virtual GUID (see [Figure 12.5 History/State Persistence Based on Virtual GUID](#) below).

Figure 12.5 History/State Persistence Based on Virtual GUID



- A *DataWriter*'s virtual GUID can be configured using the member **virtual_guid** in the [DATA_WRITER_PROTOCOL QoS Policy \(DDS Extension\)](#) (Section 6.5.3 on page 347).
- A *DataReader*'s virtual GUID can be configured using the member **virtual_guid** in the [DATA_READER_PROTOCOL QoS Policy \(DDS Extension\)](#) (Section 7.6.1 on page 511).

The `DDS_PublicationBuiltinTopicData` and `DDS_SubscriptionBuiltinTopicData` structures include the virtual GUID associated with the discovered publication or subscription (see [Built-in DataReaders \(Section 16.2 on page 773\)](#)).

12.3 Durable Writer History

The [DURABILITY QoS Policy \(Section 6.5.7 on page 368\)](#) controls whether or not, and how, published samples are stored by the *DataWriter* application for *DataReaders* that are found after the samples were initially written. The samples stored by the *DataWriter* constitute the *DataWriter*'s history.

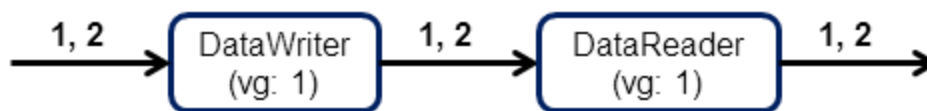
Connex DDS provides the capability to make the *DataWriter* history durable, by persisting its content in a relational database. This makes it possible for the history to be restored when the *DataWriter* restarts. See the [RTI Connex DDS Core Libraries Release Notes](#) for the list of supported relational databases.

The association between the history stored in the database and the *DataWriter* is done using the virtual GUID.

12.3.1 Durable Writer History Use Case

The following use case describes the durable writer history functionality:

1. A *DataReader* receives two samples with sequence number 1 and 2 published by a *DataWriter* with virtual GUID 1.



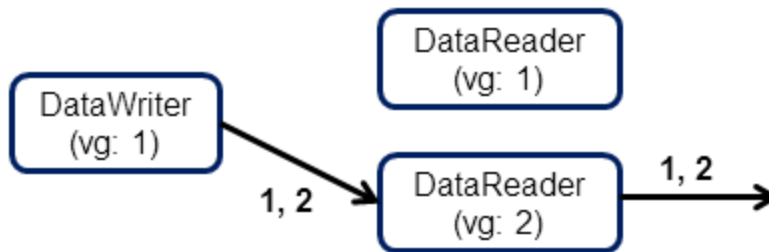
2. The process running the *DataWriter* is stopped and a new late-joining *DataReader* is created.

The new *DataReader* with virtual GUID 2 does not receive samples 1 and 2 because the original *DataWriter* has been destroyed. If the samples must be available to late-joining *DataReaders* after the *DataWriter* deletion, you can use Persistence Service, described in [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#).

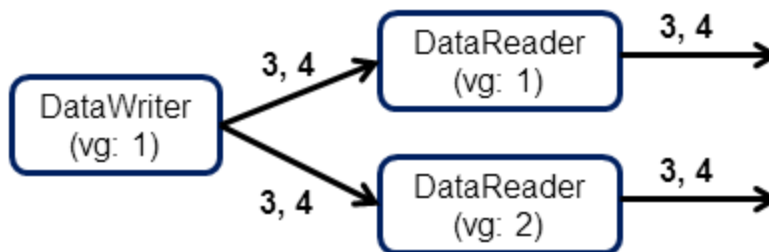


3. The *DataWriter* is restarted using the same virtual GUID.

After being restarted, the *DataWriter* restores its history. The late-joining *DataReader* will receive samples 1 and 2 because they were not received previously. The *DataReader* with virtual GUID 1 will not receive samples 1 and 2 because it already received them.



4. The *DataWriter* publishes two new samples.



The two new samples with sequence numbers 3 and 4 will be received by both *DataReaders*.

12.3.2 How To Configure Durable Writer History

Connex DDS allows a *DataWriter*'s history to be stored in a relational database that provides an ODBC driver.

For each *DataWriter* history that is configured to be durable, Connex DDS will create a maximum of two tables:

- The first table is used to store the samples associated with the writer history. The name of that table is WS<32 uuencoding of the writer virtual GUID>.
- The second table is only created for keyed-topic and it is used to store the instances associated with the writer history. The name of the second table is WI<32 uuencoding of the writer virtual GUID>.

To configure durable writer history, use the [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#) associated with *DataWriters* and *DomainParticipants*.

A 'durable writer history' property defined in the *DomainParticipant* will be applicable to all the *DataWriters* belonging to the *DomainParticipant* unless it is overwritten by the *DataWriter*. [Table 12.1 Durable Writer History Properties](#) lists the supported 'durable writer history' properties.

Table 12.1 Durable Writer History Properties

Property	Description
dds.data_writer.history.plugin_name	<p>Required.</p> <p>Must be set to "dds.data_writer.history.odbc_plugin.builtin" to enable durable writer history in the <i>DataWriter</i>.</p>
dds.data_writer.history.odbc_plugin.dsn	<p>Required.</p> <p>The ODBC DSN (Data Source Name) associated with the database where the writer history must be persisted.</p>
dds.data_writer.history.odbc_plugin.driver	<p>Tells Connext DDS which ODBC driver to load. If the property is not specified, Connext DDS will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).</p>
dds.data_writer.history.odbc_plugin.username	<p>Configures the username/password used to connect to the database.</p>
dds.data_writer.history.odbc_plugin.password	<p>Default: No password or username</p>
dds.data_writer.history.odbc_plugin.shared	<p>When set to 1, Connext DDS will create a single connection per DSN that will be shared across <i>DataWriters</i> within the same <i>Publisher</i>.</p> <p>A <i>DataWriter</i> can be configured to create its own database connection by setting this property to 0 (the default).</p>

Table 12.1 Durable Writer History Properties

Property	Description
dds.data_writer.history.odbc_plugin. instance_cache_max_size	These properties configure the resource limits associated with the ODBC writer history caches. To minimize the number of accesses to the database, Connex DDS uses two caches, one for samples and one for instances. The initial size and the maximum size of these caches are configured using these properties.
dds.data_writer.history.odbc_plugin. instance_cache_init_size	The resource limits, initial_instances , max_instances , initial_samples , max_samples , and max_samples_per_instance defined in RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) are used to configure the maximum number of samples and instances that can be stored in the relational database. Defaults:
dds.data_writer.history.odbc_plugin. sample_cache_max_size	instance_cache_max_size : max_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) instance_cache_init_size : initial_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) sample_cache_max_size : 32 sample_cache_init_size : 32
dds.data_writer.history.odbc_plugin. sample_cache_init_size	If in_memory_state (see below in this table) is 1, instance_cache_max_size is always equal to max_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) —it cannot be changed.
dds.data_writer.history.odbc_plugin. restore	This property indicates whether or not the persisted writer history must be restored once the <i>DataWriter</i> is restarted. If this property is 0, the content of the database associated with the <i>DataWriter</i> being restarted will be deleted. If it is 1, the <i>DataWriter</i> will restore its previous state from the database content. Default: 1
dds.data_writer.history.odbc_plugin. in_memory_state	This property determines how much state will be kept in memory by the ODBC writer history in order to avoid accessing the database. If this property is 1, then the property instance_cache_max_size (see above in this table) is always equal to max_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.20 on page 405) —it cannot be changed. In addition, the ODBC writer history will keep in memory a fixed state overhead of 24 bytes per sample. This mode provides the best ODBC writer history performance. However, the restore operation will be slower and the maximum number of samples that the writer history can manage is limited by the available physical memory. If it is 0, all the state will be kept in the underlying database. In this mode, the maximum number of samples in the writer history is not limited by the physical memory available. Default: 1

Durable Writer History is not supported for Multi-channel *DataWriters* (see [Multi-channel DataWriters \(Section Chapter 18 on page 824\)](#)) or when Batching is enabled (see [BATCH QosPolicy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#)); an error is reported if this type of *DataWriter* tries to configure Durable Writer History.

See also: [Durable Reader State \(Section 12.4 below\)](#).

Example C++ Code

```

/* Get default QoS */
...
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
            "dds.data_writer.history.plugin_name",
            "dds.data_writer.history.odbc_plugin.builtin",
            DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
            "dds.data_writer.history.odbc_plugin.dsn",
            "<user DSN>",
            DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
            "dds.data_writer.history.odbc_plugin.driver",
            "<ODBC library>",
            DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
            "dds.data_writer.history.odbc_plugin.shared",
            "<0|1>",
            DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
/* Create Data Writer */
...

```

12.4 Durable Reader State

Durable reader state allows a *DataReader* to locally store its state in disk and remember the data that has already been processed by the application¹. When an application restarts, each *DataReader* configured to have durable reader state automatically reads its state from disk. Data that has already been processed by the application before the restart will not be provided to the application again.

Important: The *DataReader* does not persist the full contents of the data in its historical cache; it only persists an identification (e.g. sequence numbers) of the data the application has processed. This distinction is not meaningful if your application always uses the ‘take’ methods to access your data, since these methods remove the data from the cache at the same time they deliver it to your application. (See [Read vs. Take](#)

¹The circumstances under which a data sample is considered “processed by the application” are described in the sections that follow.

(Section 7.4.3.1 on page 494)) However, if your application uses the ‘read’ methods, leaving the data in the *DataReader's* cache after you've accessed it for the first time, those previously viewed samples will not be restored to the *DataReader's* cache in the event of a restart.

Connex DDS requires a relational database to persist the state of a *DataReader*. This database is accessed using ODBC. See the [RTI Connex DDS Core Libraries Release Notes](#) for the list of supported relational databases.

12.4.1 Durable Reader State With Protocol Acknowledgment

For each *DataReader* configured to have durable state, Connex DDS will create one database table with the following naming convention: **RS<32 uuencoding of the reader virtual GUID>**. This table will store the last sequence number processed from each virtual *GUID*. For *DataReaders* on keyed topics requesting instance-ordering (see [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#)), this state will be stored per instance per virtual *GUID*.

Criteria to consider a sample “processed by the application”

- For the read/take methods that require calling **return_loan()**, a sample 's1' with sequence number 's1_seq_num' and virtual GUID 'vg1' is considered processed by the application when the *DataReader's* **return_loan()** operation is called for sample 's1' or any other sample with the same virtual GUID and a sequence number greater than 's1_seq_num'. For example:

```
retcode = Foo_reader->take(data_seq, info_seq,
    DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
if (retcode == DDS_RETCODE_NO_DATA) {
    return;
} else if (retcode != DDS_RETCODE_OK) {
    /* report error */
    return;
}
for (i = 0; i < data_seq.length(); ++i) {
    /* Operate with the data */
}
/* Return the loan */
retcode = Foo_reader->return_loan(data_seq, info_seq);
if (retcode != DDS_RETCODE_OK) {
    /* Report and error */
}
/* At this point the samples contained in data_seq
will be considered as received. If the DataReader
restarts, the samples will not be received again */
```

- For the read/take methods that do not require calling **return_loan()**, a sample 's1' with sequence number 's1_seq_num' and virtual GUID 'vg1' will be considered processed after the application

reads or takes the sample 's1' or any other sample with the same virtual GUID *and* with a sequence number greater than 's1_seq_num'. For example:

```
retcode = Foo_reader->take_next_sample(data,info);
/* At this point the sample contained in data will be
   considered as received. All the samples with a sequence
   number smaller than the sequence number associated with
   data will also be considered as received.
   If the DataReader restarts, these sample will not
   be received again */
```

If you access the samples in the *DataReader* cache out of order—for example via *QueryCondition*, specifying an instance state, or reading by instance when the PRESENTATION QoS is not set to INSTANCE_PRESENTATION_QOS—then the samples that have not yet been taken or read by the application may still be considered as “processed by the application”.

12.4.1.1 Bandwidth Utilization

To optimize network usage, if a *DataReader* configured with durable reader state is restarted and it discovers a *DataWriter* with a virtual GUID ‘vg’, the *DataReader* will ACK all the samples with a sequence number smaller than ‘sn’, where ‘sn’ is the first sequence number that has not been being processed by the application for ‘vg’.

Notice that the previous algorithm can significantly reduce the number of duplicates on the wire. However, it does not suppress them completely in the case of keyed *DataReaders* where the durable state is kept per (instance, virtual GUID). In this case, and assuming that the application has read samples out of order (e.g., by reading different instances), the ACK is sent for the lowest sequence number processed across all instances and may cause samples already processed to flow on the network again. These redundant samples waste bandwidth, but they will be dropped by the *DataReader* and not be delivered to the application.

12.4.2 Durable Reader State with Application Acknowledgment

This section assumes you are familiar with the concept of *Application Acknowledgment* as described in [Application Acknowledgment \(Section 6.3.12 on page 288\)](#).

For each *DataReader* configured to be durable and that uses application acknowledgement (see [Application Acknowledgment \(Section 6.3.12 on page 288\)](#)), Connex DDS will create one database table with the following naming convention: **RS<32 uuencoding of the reader virtual GUID>**. This table will store the list of sequence number *intervals* that have been acknowledged for each virtual GUID. The size of the column that stores the sequence number intervals is limited to 32767 bytes. If this size is exceeded for a given virtual GUID, the operation that persists the *DataReader* state into the database will fail.

12.4.2.1 Bandwidth Utilization

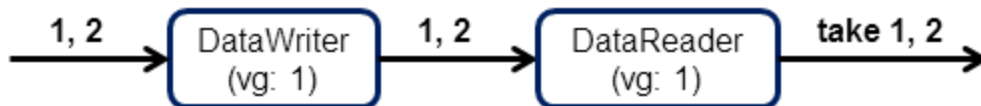
To optimize network usage, if a *DataReader* configured with durable reader state is restarted and it discovers a *DataWriter* with a virtual GUID 'vg', the *DataReader* will send an APP_ACK message with all the samples that were auto-acknowledged or explicitly acknowledged in previous executions.

Notice that this algorithm can significantly reduce the number of duplicates on the wire. However, it does not suppress them completely since the *DataReader* may send a NACK and receive some samples from the *DataWriter* before the *DataWriter* receives the APP_ACK message.

12.4.3 Durable Reader State Use Case

The following use case describes the durable reader state functionality:

1. A *DataReader* receives two samples with sequence number 1 and 2 published by a *DataWriter* with virtual GUID 1. The application takes those samples.



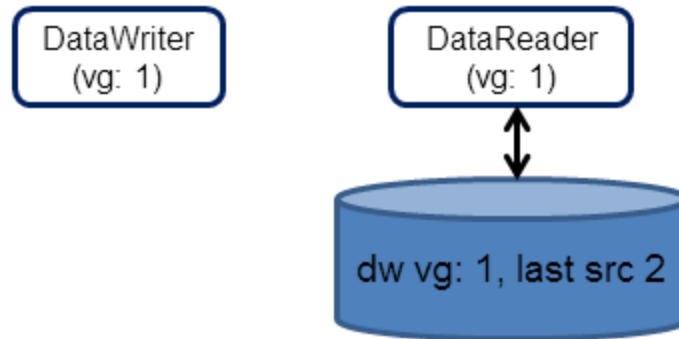
2. After the application returns the loan on samples 1 and 2, the *DataReader* considers them as processed and it persists the state change.



3. The process running the *DataReader* is stopped.

4. The *DataReader* is restarted.

Because all the samples with sequence number smaller or equal than 2 were considered received, the reader will not ask for these samples from the *DataWriter*.



12.4.4 How To Configure a DataReader for Durable Reader State

To configure a *DataReader* with durable reader state, use the [PROPERTY QosPolicy \(DDS Extension\)](#) (Section 6.5.17 on page 394) associated with *DataReaders* and *DomainParticipants*.

A property defined in the *DomainParticipant* will be applicable to all the *DataReaders* contained in the participant unless it is overwritten by the *DataReaders*. [Table 12.2 Durable Reader State Properties](#) lists the supported properties.

Table 12.2 Durable Reader State Properties

Property	Description
dds.data_reader.state.odbc.dsn	Required. The ODBC DSN (Data Source Name) associated with the database where the <i>DataReader</i> state must be persisted.
dds.data_reader.state.filter_redundant_samples	To enable durable reader state, this property must be set to 1. When set to 0, the reader state is not maintained and Connex DDS does not filter duplicate samples that may be coming from the same virtual writer. Default: 1
dds.data_reader.state.odbc.driver	This property indicates which ODBC driver to load. If the property is not specified, Connex DDS will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).

Table 12.2 Durable Reader State Properties

Property	Description
dds.data_reader.state.odbc.username	These two properties configure the username and password used to connect to the database. Default: No password or username
dds.data_reader.state.odbc.password	
dds.data_reader.state.restore	This property indicates if the persisted <i>DataReader</i> state must be restored or not once the <i>DataReader</i> is restarted. If this property is 0, the previous state will be deleted from the database. If it is 1, the <i>DataReader</i> will restore its previous state from the database content. Default: 1
dds.data_reader.state.checkpoint_frequency	This property controls how often the reader state is stored into the database. A value of <i>N</i> means store the state once every <i>N</i> samples. A high frequency will provide better performance. However, if the reader is restarted it may receive some duplicate samples. These samples will be filtered by Connexx DDS and they will not be propagated to the application. Default: 1
dds.data_reader.state.persistence_service.request_depth	This property indicates how many of the most recent historical samples the persisted <i>DataReader</i> wants to receive upon start-up. Default: 0

Example (C++ code):

```

/* Get default QoS */

...
retcode = DDSPropertyQosPolicyHelper::add_property(
    readerQos.property,
    "dds.data_reader.state.odbc.dsn",
    "<user DSN>", DDS_BOOLEAN_FALSE);

if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property(readerQos.property,
    "dds.data_reader.state.odbc.driver",
    "<ODBC library>", DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property(readerQos.property,
    "dds.data_reader.state.restore", "<0|1>",
    DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}

```



```

}
/* Create Data Reader */
...

```

12.5 Data Durability

The data durability feature is an implementation of the OMG DDS Persistence Profile. The [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) allows an application to configure a *DataWriter* so that the information written by the *DataWriter* survives beyond the lifetime of the *DataWriter*.

Connex DDS implements TRANSIENT and PERSISTENT durability using an external service called Persistence Service, available for purchase as a separate RTI product.

Persistence Service receives information from *DataWriters* configured with TRANSIENT or PERSISTENT durability and makes that information available to late-joining *DataReaders*—even if the original *DataWriter* is not running.

The samples published by a *DataWriter* can be made durable by setting the **kind** field of the [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) to one of the following values:

- **DDS_TRANSIENT_DURABILITY_QOS**: Connex DDS will store previously published samples in memory using Persistence Service, which will send the stored data to newly discovered *DataReaders*.
- **DDS_PERSISTENT_DURABILITY_QOS**: Connex DDS will store previously published samples in permanent storage, like a disk, using Persistence Service, which will send the stored data to newly discovered *DataReaders*.

A *DataReader* can request TRANSIENT or PERSISTENT data by setting the **kind** field of the corresponding [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#). A *DataReader* requesting PERSISTENT data will not receive data from *DataWriters* or Persistence Service applications that are configured with TRANSIENT durability.

12.5.1 RTI Persistence Service

Persistence Service is a Connex DDS application that is configured to persist topic data. Persistence Service is included with the Connex DDS Professional, Evaluation, and Basic package types. For each one of the topics that must be persisted for a specific domain, the service will create a *DataWriter* (known as PRSTDataWriter) and a *DataReader* (known as PRSTDataReader). The samples received by the PRSTDataReaders will be published by the corresponding PRSTDataWriters to be available for late-joining *DataReaders*.

For more information on Persistence Service, please see:

- [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)
- [Configuring Persistence Service \(Section Chapter 27 on page 934\)](#)
- [Running RTI Persistence Service \(Section Chapter 28 on page 962\)](#)

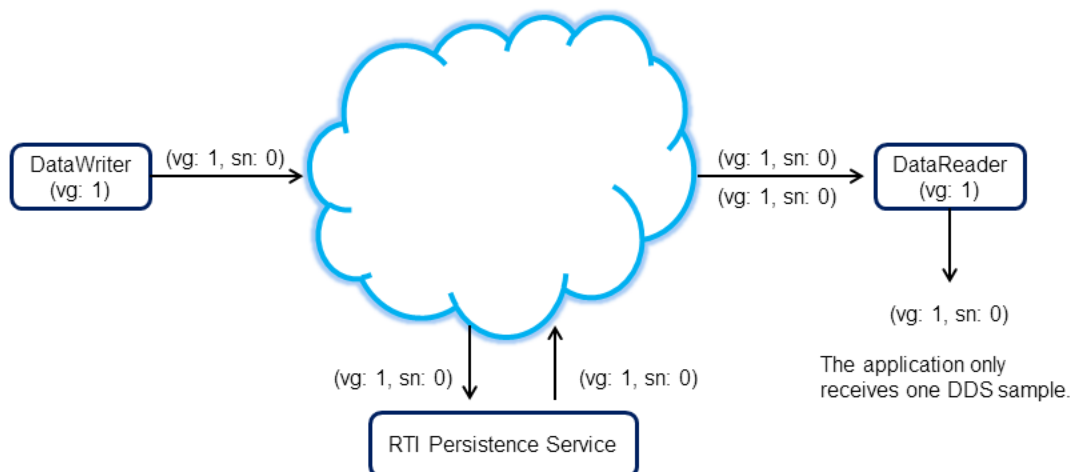
Persistence Service can be configured to operate in PERSISTENT or TRANSIENT mode:

- **TRANSIENT mode** The PRSTDataReaders and PRSTDataWriters will be created with TRANSIENT durability and Persistence Service will keep the received samples in memory. Samples published by a TRANSIENT *DataWriter* will survive the *DataWriter* lifecycle but will not survive the lifecycle of Persistence Service (unless you are running multiple copies).
- **PERSISTENT mode** The PRSTDataWriters and PRSTDataReaders will be created with PERSISTENT durability and Persistence Service will store the received samples in files or in an external relational database. Samples published by a PERSISTENT *DataWriter* will survive the *DataWriter* lifecycle as well as any restarts of Persistence Service.

Peer-to-Peer Communication:

By default, a PERSISTENT/TRANSIENT *DataReader* will receive samples directly from the original *DataWriter* if it is still alive. In this scenario, the *DataReader* may also receive the same samples from Persistence Service. Duplicates will be discarded at the middleware level. This Peer-To-Peer communication pattern is illustrated in [Figure 12.6 Peer-to-Peer Communication below](#). To use this peer-to-peer communication pattern, set the **direct_communication** field in the DURABILITY QosPolicy (Section 6.5.7 on page 368) to TRUE. A PERSISTENT/TRANSIENT *DataReader* will receive information directly from PERSISTENT/TRANSIENT *DataWriters*.

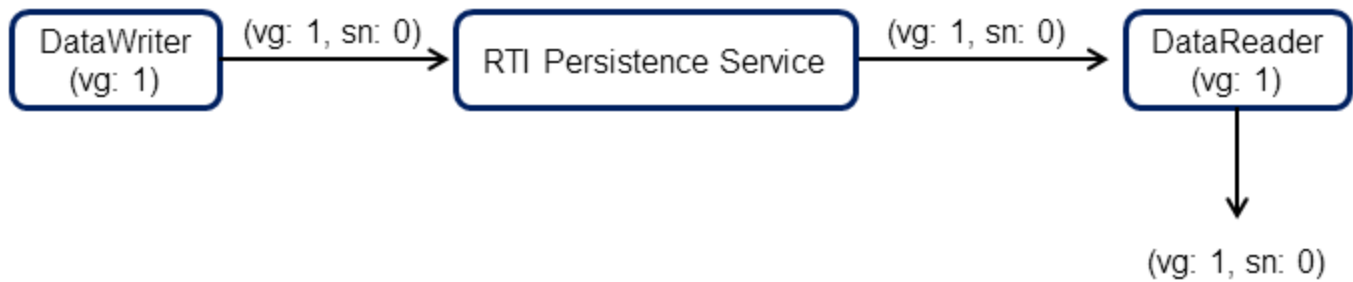
Figure 12.6 Peer-to-Peer Communication



Relay Communication

A PERSISTENT/TRANSIENT *DataReader* may also be configured to not receive samples from the original *DataWriter*. In this case the traffic is relayed by Persistence Service. This ‘relay communication’ pattern is illustrated in [Figure 12.7 Relay Communication](#) below. To use relay communication, set the **direct_communication** field in the DURABILITY QosPolicy (Section 6.5.7 on page 368) to FALSE. A PERSISTENT/TRANSIENT *DataReader* will receive all the information from Persistence Service.

Figure 12.7 Relay Communication



Chapter 13 Guaranteed Delivery of Data

13.1 Introduction

Some application scenarios need to ensure that the information produced by certain producers is delivered to all the intended consumers. This chapter describes the mechanisms available in *Connext DDS* to guarantee the delivery of information from producers to consumers such that the delivery is robust to many kinds of failures in the infrastructure, deployment, and even the producing/consuming applications themselves.

Guaranteed information delivery is not the same as protocol-level reliability (described in [Reliable Communications \(Section Chapter 10 on page 629\)](#)) or information durability (described in [Mechanisms for Achieving Information Durability and Persistence \(Section Chapter 12 on page 675\)](#)). Guaranteed information delivery is an end-to-end application-level QoS, whereas the others are middleware-level QoS. There are significant differences between these two:

- With protocol-level reliability alone, the producing application knows that the information is received by the protocol layer on the consuming side. However the producing application cannot be certain that the consuming application read that information or was able to successfully understand and process it. The information could arrive in the consumer's protocol stack and be placed in the *DataReader* cache but the consuming application could either crash before it reads it from the cache, not read its cache, or read the cache using queries or conditions that prevent that particular DDS data sample from being accessed. Furthermore, the consuming application could access the DDS sample, but not be able to interpret its meaning or process it in the intended way.
- With information durability alone, there is no way to specify or characterize the intended consumers of the information. Therefore the infrastructure has no way to know when the information has been consumed by all the intended recipients. The information may be persisted such that it is not lost and is available to future applications, but the infrastructure and producing applications have no way to know that all the intended consumers have joined the system, received the information, and processed it successfully.

The guaranteed data-delivery mechanism provided in *Connex DDS* overcomes the limitations described above by providing the following features:

- **Required subscriptions.** This feature provides a way to configure, identify and detect the applications that are intended to consume the information. See [Required Subscriptions \(Section 6.3.13 on page 294\)](#).
- **Application-level acknowledgments.** This feature provides the means ensure that the information was successfully processed by the application-layer in a consumer application. See [Application Acknowledgment \(Section 6.3.12 on page 288\)](#).
- **Durable subscriptions.** This feature leverages the RTI Persistence Service to persist DDS DDS samples intended for the required subscriptions such that they are delivered even if the originating application is not available. See [Configuring Durable Subscriptions in Persistence Service \(Section 27.9 on page 955\)](#).

These features used in combination with the mechanisms provided for Information Durability and Persistence (see [Mechanisms for Achieving Information Durability and Persistence \(Section Chapter 12 on page 675\)](#)) enable the creation of applications where the information delivery is guaranteed despite application and infrastructure failures. [Scenarios \(Section 13.2 on page 700\)](#) describes various guaranteed-delivery scenarios and how to configure the applications to achieve them.

When implementing an application that needs guaranteed data delivery, we have to consider three key aspects:

Key Aspects to Consider	Related Features and QoS
Identifying the required consumers of information	Required subscriptions Durable subscriptions EntityName QoS policy Availability QoS policy
Ensuring the intended consumer applications process the data successfully	Application-level acknowledgment Acknowledgment by a quorum of required and durable subscriptions Reliability QoS policy (acknowledgment mode) Availability QoS policy
Ensuring information is available to late joining applications	Persistence Service Durable Subscriptions Durability QoS Durable Writer History

13.1.1 Identifying the Required Consumers of Information

The first step towards ensuring that information is processed by the intended consumers is the ability to specify and recognize those intended consumers. This is done using the *required subscriptions* feature (Required Subscriptions (Section 6.3.13 on page 294)) configured via the `ENTITY_NAME` QoS Policy (DDS Extension) (Section 6.5.9 on page 374) and `AVAILABILITY` QoS Policy (DDS Extension) (Section 6.5.1 on page 337)).

Connex DDS `DataReader` entities (as well as `DataWriter` and `DomainParticipant` entities) can have a *name* and a **role_name**. These names are configured using the `ENTITY_NAME` QoS Policy (DDS Extension) (Section 6.5.9 on page 374), which is propagated via DDS discovery and is available as part of the builtin-topic data for the Entity (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)).

The DDS `DomainParticipant`, `DataReader` and `DataWriter` entities created by RTI-provided applications and services, specifically services such as *RTI Persistence Service*, automatically configure the `ENTITY_NAME` QoS policy according to their function. For example the `DataReaders` created by *RTI Persistence Service* have their **role_name** set to “PERSISTENCE_SERVICE”.

Unless explicitly set by the user, the `DomainParticipant`, `DataReader` and `DataWriter` entities created by end-user applications have their **name** and **role_name** set to NULL. However applications may modify this using the `ENTITY_NAME` QoS Policy (DDS Extension) (Section 6.5.9 on page 374).

Connex DDS uses the **role_name** of `DataReaders` to identify the consumer’s logical function. For this reason *Connex* DDS’s *required subscriptions* feature relies on the **role_name** to identify intended consumers of information. The use of the `DataReader`’s **role_name** instead of the **name** is intentional. From the point of view of the information producer, the important thing is not the concrete `DataReader` (identified by its **name**, for example, “Logger123”) but rather its logical function in the system (identified by its **role_name**, for example “LoggingService”).

A `DataWriter` that needs to ensure its information is delivered to all the intended consumers uses the `AVAILABILITY` QoS Policy (DDS Extension) (Section 6.5.1 on page 337) to configure the role names of the consumers that must receive the information.

The `AVAILABILITY` QoS Policy set on a `DataWriter` lets an application configure the required consumers of the data produced by the `DataWriter`. The required consumers are specified in the **required_matched_endpoint_groups** attribute within the `AVAILABILITY` QoS Policy. This attribute is a sequence of DDS `EndpointGroup` structures. Each `EndpointGroup` represents a required information consumer characterized by the consumer’s **role_name** and **quorum_count**. The **role_name** identifies a logical consumer; the **quorum_count** specifies the minimum number of consumers with that **role_name** that must acknowledge the DDS sample before the `DataWriter` can consider it delivered to that required consumer.

For example, an application that wants to ensure data written by a `DataWriter` is delivered to at least two Logging Services and one Display Service would configure the `DataWriter`’s `AVAILABILITY` QoS Policy with a **required_matched_endpoint_groups** consisting of two elements. The first element would specify a required consumer with the **role_name** “LoggingService” and a **quorum_count** of 2. The

second element would specify a required consumer with the **role_name** “DisplayService” and a **quorum_count** of 1. Furthermore, the application would set the logging service *DataReader* ENTITY_NAME policy to have a **role_name** of “LoggingService” and similarly the display service *DataReader* ENTITY_NAME policy to have the **role_name** of “DisplayService.”

A *DataWriter* that has been configured with an AVAILABILITY QoS policy will not remove DDS samples from the *DataWriter* cache until they have been “delivered” to both the already-discovered *DataReaders* and the minimum number (**quorum_count**) of *DataReaders* specified for each role. In particular, DDS samples will be retained by the *DataWriter* if the **quorum_count** of matched *DataReaders* with a particular **role_name** have not been discovered yet.

We used the word “delivered” in quotes above because the level of assurance a *DataWriter* has that a particular DDS sample has been delivered depends on the setting of the [RELIABILITY QoSPolicy \(Section 6.5.19 on page 400\)](#). We discuss this next in [Ensuring Consumer Applications Process the Data Successfully \(Section 13.1.2 below\)](#).

13.1.2 Ensuring Consumer Applications Process the Data Successfully

[Identifying the Required Consumers of Information \(Section 13.1.1 on the previous page\)](#) described mechanisms by which an application could configure who the required consumers of information are. This section is about the criteria, mechanisms, and assurance provided by *Connex DDS* to ensure consumers have the information delivered to them and process it in a successful manner.

RTI provides four levels of information delivery guarantee. You can set your desired level using the [RELIABILITY QoSPolicy \(Section 6.5.19 on page 400\)](#). The levels are:

- **Best-effort, relying only on the underlying transport**The *DataWriter* considers the DDS sample delivered/acknowledged as soon as it is given to the transport to send to the *DataReader*’s destination. Therefore, the only guarantee is the one provided by the underlying transport itself. Note that even if the underlying transport is reliable (e.g., shared memory or TCP) the reliability is limited to the transport-level buffers. There is no guarantee that the DDS sample will arrive to the *DataReader* cache because after the transport delivers to the *DataReader*’s transport buffers, it is possible for the DDS sample to be dropped because it exceeds a resource limit, fails to deserialize properly, the receiving application crashes, etc.
- **Reliable with protocol acknowledgment**The DDS-RTPS reliability protocol used by *Connex DDS* provides acknowledgment at the RTPS protocol level: a *DataReader* will acknowledge it has deserialized the DDS sample correctly and stored it in the *DataReader*’s cache. However, there is no guarantee the application actually processed the DDS sample. The application might crash before processing the DDS sample, or it might simply fail to read it from the cache.
- **Reliable with Application Acknowledgment (Auto)**Application Acknowledgment in Auto mode causes *Connex DDS* to send an additional application-level acknowledgment (above and beyond the RTPS protocol level acknowledgment) after the consuming application has read the DDS

sample from the *DataReader* cache and the application has subsequently called the *DataReader*'s **return_loan()** operation (see [Loaning and Returning Data and SampleInfo Sequences \(Section 7.4.2 on page 492\)](#)) for that DDS sample. This mode guarantees that the application has fully read the DDS sample all the way until it indicates it is done with it. However it does not provide a guarantee that the application was able to successfully interpret or process the DDS sample. For example, the DDS sample could be a command to execute a certain action and the application may read the DDS sample and not understand the command or may not be able to execute the action.

- **Reliable with Application Acknowledgment (Explicit)** Application Acknowledgment in Explicit mode causes *Connex DDS* to send an application-level acknowledgment only after the consuming application has read the DDS sample from the *DataReader* cache and subsequently called the *DataReader*'s **acknowledge_sample()** operation (see [Acknowledging DDS Samples \(Section 7.4.4 on page 502\)](#)) for that DDS sample. This mode guarantees that the application has fully read the DDS sample and completed operating on it as indicated by explicitly calling **acknowledge_sample()**. In contrast with the Auto mode described above, the application can delay the acknowledgment of the DDS sample beyond the time it holds onto the data buffers, allowing it to be process in a more flexible manner. Similar to the Auto mode, it does not provide a guarantee that the application was able to successfully interpret or process the DDS sample. For example, the DDS sample could be a command to execute a certain action and the application may read the DDS sample and not understand the command or may not be able to execute the action. Applications that need guarantees that the data was successfully processed and interpreted should use a request-reply interaction, which is available as part of the *Connex DDS Professional*, *Evaluation*, and *Basic* package types (see [Part 4: Request-Reply Communication Pattern \(Section on page 873\)](#)).

13.1.3 Ensuring Information is Available to Late-Joining Applications

The third aspect of guaranteed data delivery addresses situations where the application needs to ensure that the information produced by a particular *DataWriter* is available to *DataReaders* that join the system after the data was produced. The need for data delivery may even extend beyond the lifetime of the producing application; that is, it may be required that the information is delivered to applications that join the system after the producing application has left the system.

Connex DDS provides four mechanisms to handle these scenarios:

- **The DDS Durability QoS Policy.** The [DURABILITY QoS Policy \(Section 6.5.7 on page 368\)](#) specifies whether DDS samples should be available to late joiners. The policy is set on the *DataWriter* and the *DataReader* and supports four kinds: `VOLATILE`, `TRANSIENT_LOCAL`, `TRANSIENT`, or `PERSISTENT`. If the *DataWriter*'s Durability QoS policy is set to `VOLATILE` kind, the *DataWriter*'s DDS samples will not be made available to any late joiners. If the *DataWriter*'s policy kind is set to `TRANSIENT_LOCAL`, `TRANSIENT`, or `PERSISTENT`, the DDS samples will be made available for late-joining *DataReaders* who also set their `DURABILITY QoS` policy kind to something other than `VOLATILE`.

- Durable Writer History.** A *DataWriter* configured with a DURABILITY QoS policy kind other than VOLATILE keeps its data in a local cache so that it is available when the late-joining application appears. The data is maintained in the *DataWriter*'s cache until it is considered to be no longer needed. The precise criteria depends on the configuration of additional QoS policies such as [LIFESPAN QoS Policy \(Section 6.5.12 on page 381\)](#), [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#), [RESOURCE_LIMITS QoS Policy \(Section 6.5.20 on page 405\)](#), etc. For the purposes of guaranteeing information delivery it is important to note that the *DataWriter*'s cache can be configured to be a memory cache or a durable (disk-based) cache. A memory cache will not survive an application restart. However, a durable (disk-based) cache can survive the restart of the producing application. The use a durable writer history, including the use of an external ODBC database as a cache is described in [Durable Writer History \(Section 12.3 on page 681\)](#).
- RTI Persistence Service.** This service allows the information produced by a *DataWriter* to survive beyond the lifetime of the producing application. *Persistence Service* is a stand-alone application that runs on many supported platforms. This service complies with the Persistent Profile of the OMG DDS specification. The service uses DDS to subscribe to the *DataWriters* that specify a [DURABILITY QoS Policy \(Section 6.5.7 on page 368\)](#) kind of TRANSIENT or PERSISTENT. *Persistence Service* receives the data from those *DataWriters*, stores the data in its internal caches, and makes the data available via *DataWriters* (which are automatically created by *Persistence Service*) to late-joining *DataReaders* that specify a Durability kind of TRANSIENT or PERSISTENT. *Persistence Service* can operate as a relay for the information from the original writer, preserving the **source_timestamp** of the data, as well as the original DDS sample virtual writer GUID (see [RTI Persistence Service \(Section 12.5.1 on page 692\)](#)). In addition, you can configure *Persistence Service* itself to use a memory-based cache or a durable (disk-based or database-based) cache. See [Configuring Persistent Storage \(Section 27.6 on page 943\)](#). Configuration of redundant and load-balanced persistence services is also supported.
- Durable Subscriptions.** This is a *Persistence Service* configuration setting that allows configuration of the required subscriptions ([Identifying the Required Consumers of Information \(Section 13.1.1 on page 697\)](#)) for the data stored by *Persistence Service* ([Managing Data Instances \(Working with Keyed Data Types\) \(Section 6.3.14 on page 296\)](#)). Configuring required subscriptions for *Persistence Service* ensures that the service will store the DDS samples until they have been delivered to the configured number (**quorum_count**) of *DataReaders* that have each of the specified roles.

13.2 Scenarios

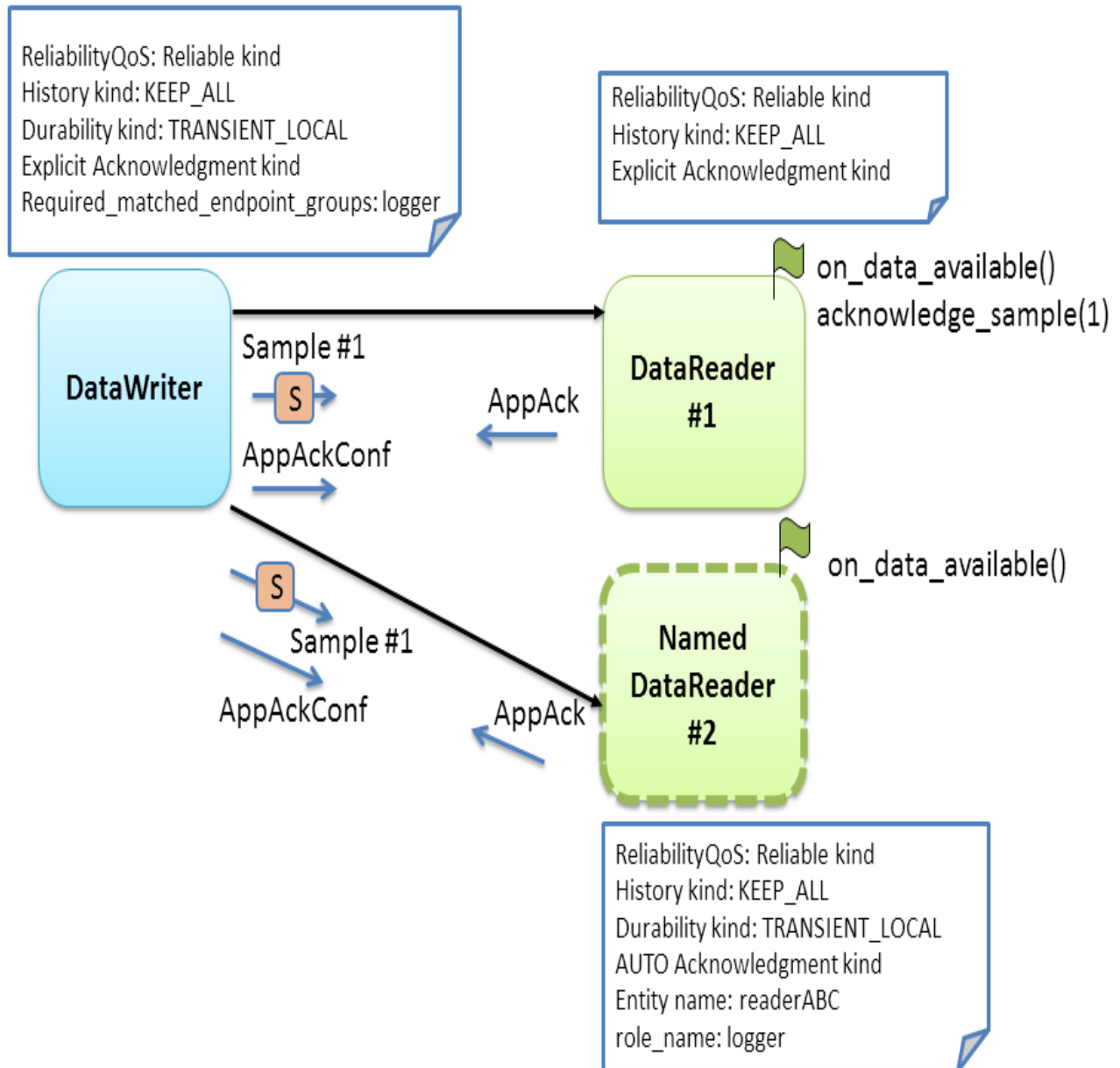
In each of the scenarios below, we assume both the *DataWriter* and *DataReader* are configured for strict reliability (RELIABLE ReliabilityQoSPolicyKind and KEEP_ALL HistoryQoSPolicyKind, see [Controlling Queue Depth with the History QoS Policy \(Section 10.3.3 on page 644\)](#)). As a result, when the *DataWriter*'s cache is full of unacknowledged DDS samples, the **write()** operation will block until DDS samples are acknowledged by all the intended consumers.

13.2.1 Scenario 1: Guaranteed Delivery to a-priori Known Subscribers

A common use case is to guarantee delivery to a set of known subscribers. These subscribers may be already running and have been discovered, they may be temporarily non-responsive, or it could be that some of those subscribers are still not present in the system. See [Figure 13.1 Guaranteed Delivery Scenario 1 on the next page](#).

To guarantee delivery, the list of required subscribers should be configured using the [AVAILABILITY QosPolicy \(DDS Extension\) \(Section 6.5.1 on page 337\)](#) on the *DataWriters* to specify the **role_name** and **quorum_count** for each required subscription. Similarly the [ENTITY_NAME QosPolicy \(DDS Extension\) \(Section 6.5.9 on page 374\)](#) should be used on the *DataReaders* to specify their **role_name**. In addition we use [Application Acknowledgment \(Section 6.3.12 on page 288\)](#) to guarantee the DDS sample was delivered and processed by the *DataReader*.

Figure 13.1 Guaranteed Delivery Scenario 1



The *DataWriter's* and *DataReader's* RELIABILITY QoS Policy can be configured for either AUTO or EXPLICIT application acknowledgment kind. As the *DataWriter* publishes the DDS sample, it will await acknowledgment from the *DataReader* (through the protocol-level acknowledgment) and from the subscriber application (through the additional application-level acknowledgment). The *DataWriter* will only

consider the DDS sample acknowledged when it has been acknowledged by all discovered active *DataReaders* and also by the **quorum_count** of each required subscription.

In this specific scenario, *DataReader* #1 is configured for EXPLICIT application acknowledgment. After reading and processing the DDS sample, the subscribing application calls **acknowledge_sample()** or **acknowledge_all()** (see [Acknowledging DDS Samples \(Section 7.4.4 on page 502\)](#)). As a result, Connex DDS will send an application-level acknowledgment to the *DataWriter*, which will in its turn confirm the acknowledgment.

If the DDS sample was lost in transit, the reliability protocol will repair the DDS sample. Since it has not been acknowledged, it remains available in the writer's queue to be automatically resent by Connex DDS. The DDS sample will remain available until acknowledged by the application. If the subscribing application crashes while processing the DDS sample and restarts, Connex DDS will repair the unacknowledged DDS sample. DDS samples which already been processed and acknowledged will not be resent.

In this scenario, *DataReader* #2 may be a late joiner. When it starts up, because it is configured with TRANSIENT_LOCAL Durability, the reliability protocol will re-send the DDS samples previously sent by the writer. These DDS samples were considered unacknowledged by the *DataWriter* because they had not been confirmed yet by the required subscription (identified by its **role_name**: 'logger').

DataReader #2 does not explicitly acknowledge the DDS samples it reads. It is configured to use AUTO application acknowledgment, which will automatically acknowledge DDS samples that have been read or taken after the application calls the *DataReader return_loan* operation.

This configuration works well for situations where the *DataReader* may not be immediately available or may restart. However, this configuration does not provide any guarantee if the *DataWriter* restarts. When the *DataWriter* restarts, DDS samples previously unacknowledged are lost and will no longer be available to any late joining *DataReaders*.

13.2.2 Scenario 2: Surviving a Writer Restart when Delivering DDS Samples to a priori Known Subscribers

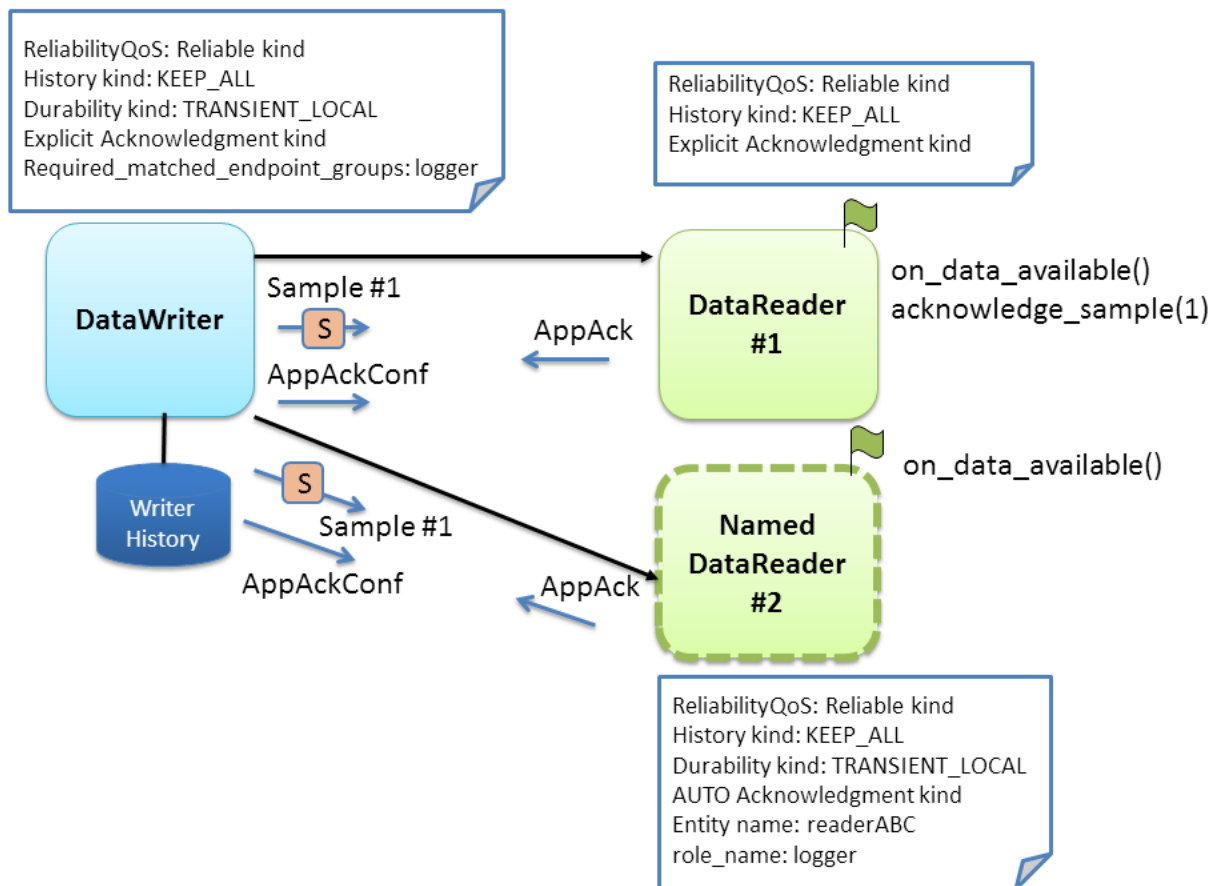
Scenario 1 describes a use case where DDS samples are delivered to a list of a priori known subscribers. In that scenario, Connex DDS will deliver DDS samples to the late-joining or restarting subscriber. However, if the producer is re-started the DDS samples it had written will no longer be available to future subscribers.

To handle a situation where the producing application is restarted, we will use the [Durable Writer History \(Section 12.3 on page 681\)](#) feature. See [Figure 13.2 Guaranteed Delivery Scenario 2 on the next page](#).

A *DataWriter* can be configured to maintain its data and state in durable storage. This configuration is done using the PROPERTY QoS policy as described in [How To Configure Durable Writer History \(Section 12.3.2 on page 683\)](#). With this configuration the DDS data samples written by the *DataWriter* and any necessary internal state is persisted by the *DataWriter* into durable storage. As a result, when the *DataWriter* restarts, DDS samples which had not been acknowledged by the set of required subscriptions

will be resent and late-joining *DataReaders* specifying DURABILITY kind different from VOLATILE will receive the previously-written DDS samples.

Figure 13.2 Guaranteed Delivery Scenario 2



13.2.3 Scenario 3: Delivery Guaranteed by Persistence Service (Store and Forward) to a priori Known Subscribers

Previous scenarios illustrated that using the DURABILITY, RELIABILITY, and AVAILABILITY QoS policies we can ensure that as long as the *DataWriter* is present in the system, DDS samples written by a *DataWriter* will be delivered to the intended consumers. The use of the durable writer history in the previous scenario extended this guarantee even in the presence of a restart of the application writing the data.

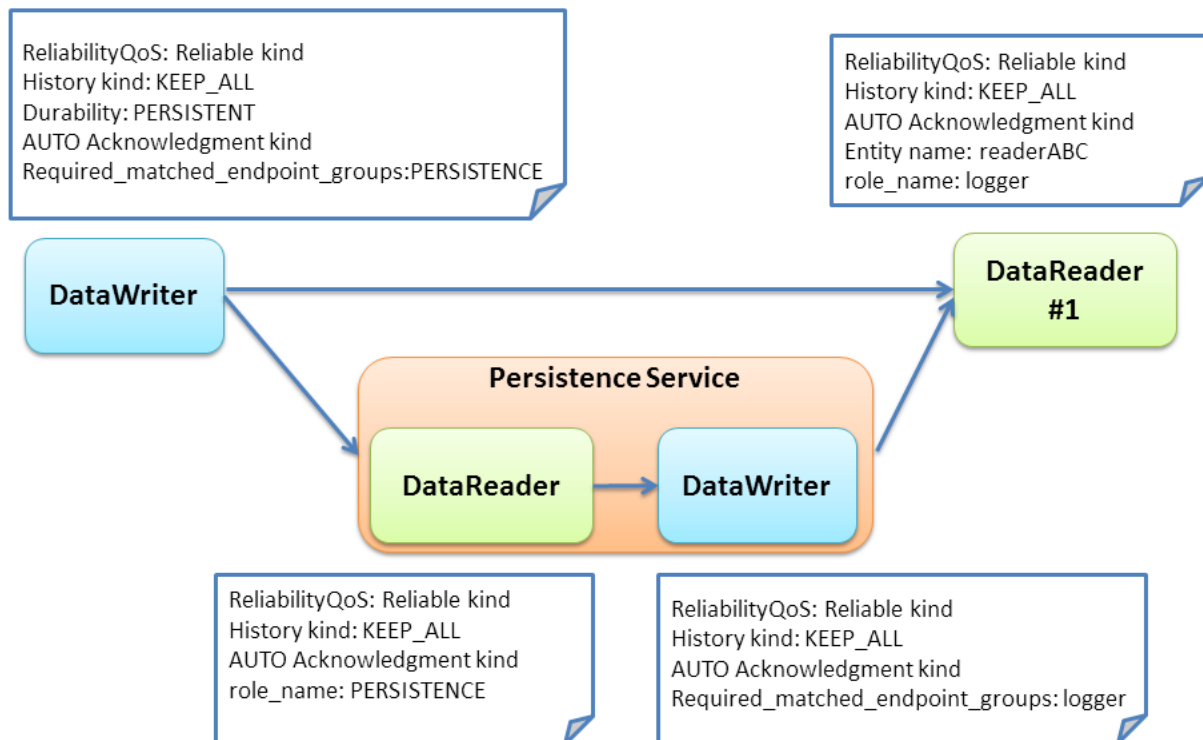
This scenario addresses the situation where the originating application that produced the data is no longer available. For example, the network could have become partitioned, the application could have been terminated, it could have crashed and not have been restarted, etc.

In order to deliver data to applications that appear after the producing application is no longer available on the network it is necessary to have another service that stores those DDS samples and delivers them. This is the purpose of the *RTI Persistence Service*.

The *RTI Persistence Service* can be configured to automatically discover *DataWriters* that specify a DURABILITY QoS with **kind** TRANSIENT or PERSISTENT and automatically create pairs (*DataReader*, *DataWriter*) that receive and store that information (see [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)). All the *DataReaders* created by the *RTI Persistence Service* have the ENTITY_QOS policy set with the **role_name** of “PERSISTENCE_SERVICE”. This allows an application to specify *Persistence Service* as one of the required subscriptions for its *DataWriters*.

In this third scenario, we take advantage of this capability to configure the *DataWriter* to have the *RTI Persistence Service* as a required subscription. See [Figure 13.3 Guaranteed Delivery Scenario 3 below](#).

Figure 13.3 Guaranteed Delivery Scenario 3



The *RTI Persistence Service* can also have its *DataWriters* configured with required subscriptions. This feature is known as *Persistence Service* “durable subscriptions”. *DataReader #1* is pre configured in *Persistence Service* as a Durable Subscription. (Alternatively, *DataReader #1* could have registered itself dynamically as Durable Subscription using the *DomainParticipant register_durable_subscription()* operation).

We also configure the RELIABILITY QoS policy setting of the AcknowledgmentKind to APPLICATION_AUTO_ACKNOWLEDGMENT_MODE in order to ensure DDS samples are stored in the *Persistence Service* and properly processed on the consuming application prior to them being removed from the DataWriter cache.

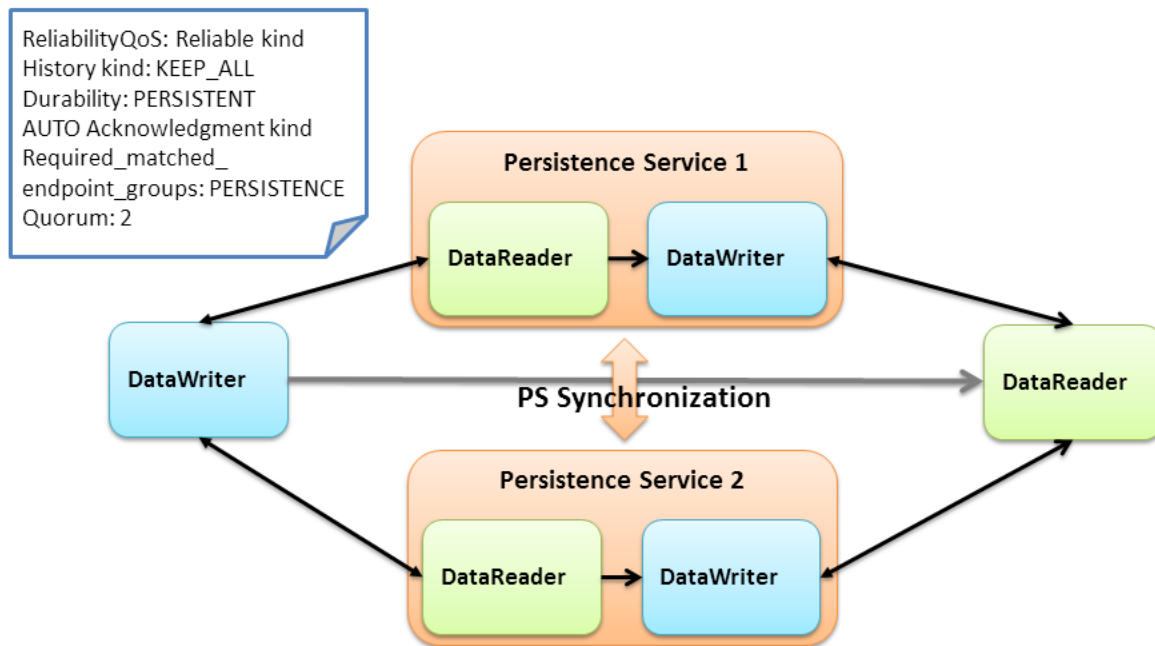
With this configuration in place the *DataWriter* will deliver DDS samples to the *DataReader* and to the *Persistence Service* reliably and wait for the Application Acknowledgment from both. Delivery of DDS samples to DataReader #1 and the *Persistence Service* occurs concurrently. The *Persistence Service* in turn takes responsibility to deliver the DDS samples to the configured “logger” durable subscription. If the original publisher is no longer available, DDS samples can still be delivered by the *Persistence Service* to DataReader #1 and any other late-joining *DataReaders*.

When DataReader #1 acknowledges the DDS sample through an application-acknowledgment message, both the original *DataWriter* and *Persistence Service* will receive the application-acknowledgment. Connext DDS takes advantage of this to reduce or eliminate delivery of duplicate DDS samples, that is, the *Persistence Service* can notice that DataReader #1 has acknowledged a DDS sample and refrain from separately sending the same DDS sample to DataReader #1.

13.2.3.1 Variation: Using Redundant Persistence Services

Using a single Persistence Service to guarantee delivery can still raise concerns about having the Persistence Service as a single point of failure. To provide a level of added redundancy, the publisher may be configured to await acknowledgment from a quorum of multiple persistence services (**role_name** remains PERSISTENCE). Using this configuration we can achieve higher levels of redundancy

Figure 13.4 Guaranteed Delivery Scenario 3 with Redundant Persistence Service



The RTI *Persistence Services* will automatically share information to keep each other synchronized. This includes both the data and also the information on the durable subscriptions. That is, when a Persistence Service discovers a durable subscription, information about durable subscriptions is automatically replicated and synchronized among persistence services (CITE: New section to be written in Persistence Service Chapter).

13.2.3.2 Variation: Using Load-Balanced Persistent Services

The *Persistence Service* will store DDS samples on behalf of many DataWriters and, depending on the configuration, it might write those DDS samples to a database or to disk. For this reason the *Persistence Service* may become a bottleneck in systems with high durable DDS sample throughput.

It is possible to run multiple instances of the *Persistence Service* in a manner where each is only responsible for the guaranteed delivery of certain subset of the durable data being published. These *Persistence Service* can also be run different computers and in this manner achieve much higher throughput. For example, depending on the hardware, using typical hard-drives a single a *Persistence Service* may be able to store only 30000 DDS samples per second. By running 10 persistence services in 10 different computers we would be able to handle storing 10 times that system-wide, that is, 300000 DDS samples per second.

The data to be persisted can be partitioned among the persistence services by specifying different Topics to be persisted by each *Persistence Service*. If a single Topic has more data that can be handled by a single *Persistence Service* it is also possible to specify a content-filter so that only the data within that Topic that

matches the filter will be stored by the *Persistence Service*. For example assume the Topic being persisted has an member named “x” of type float. It is possible to configure two *Persistence Services* one with the filter “x>10”, and the other “x <=10”, such that each only stores a subject of the data published on the Topic. See also: [Configuring Durable Subscriptions in Persistence Service \(Section 27.9 on page 955\)](#).

Chapter 14 Discovery

This section discusses how Connex DDS objects on different nodes find out about each other using the default Simple Discovery Protocol (SDP). It describes the sequence of messages that are passed between Connex DDS on the sending and receiving sides.

This section includes:

- [What is Discovery? \(Section 14.1 on the next page\)](#)
- [Configuring the Peers List Used in Discovery \(Section 14.2 on page 711\)](#)
- [Discovery Implementation \(Section 14.3 on page 717\)](#)
- [Debugging Discovery \(Section 14.4 on page 735\)](#)
- [Ports Used for Discovery \(Section 14.5 on page 738\)](#)

The discovery process occurs automatically, so you do not have to implement any special code. We recommend that all users read [What is Discovery? \(Section 14.1 on the next page\)](#) and [Configuring the Peers List Used in Discovery \(Section 14.2 on page 711\)](#). The remaining sections contain advanced material for those who have a particular need to understand what is happening ‘under the hood.’ This information can help you debug a system in which objects are not communicating.

You may also be interested in reading [Transport Plugins \(Section Chapter 15 on page 743\)](#) , as well as learning about these QosPolicies:

- [TRANSPORT_SELECTION QosPolicy \(DDS Extension\) \(Section 6.5.23 on page 411\)](#)
- [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\) \(Section 8.5.7 on page 606\)](#)
- [TRANSPORT_UNICAST QosPolicy \(DDS Extension\) \(Section 6.5.24 on page 412\)](#)
- [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#)

14.1 What is Discovery?

Discovery is the behind-the-scenes way in which Connex DDS objects (*DomainParticipants*, *DataWriters*, and *DataReaders*) on different nodes find out about each other. Each *DomainParticipant* maintains a database of information about all the active *DataReaders* and *DataWriters* that are in the same DDS domain. This database is what makes it possible for *DataWriters* and *DataReaders* to communicate. To create and refresh the database, each application follows a common discovery process.

This chapter describes the default discovery mechanism known as the Simple Discovery Protocol, which includes two phases: [Simple Participant Discovery \(Section 14.1.1 below\)](#) and [Simple Endpoint Discovery \(Section 14.1.2 on the facing page\)](#). (Discovery can also be performed using the Enterprise Discovery Protocol—this requires a separately purchased package, *RTI Enterprise Discovery Service*.)

The goal of these two phases is to build, for each *DomainParticipant*, a complete picture of all the entities that belong to the remote participants that are in its peers list. The peers list is the list of nodes with which a participant may communicate. It starts out the same as the *initial_peers* list that you configure in the [DISCOVERY QosPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#). If the *accept_unknown_peers* flag in that same QosPolicy is TRUE, then other nodes may also be added as they are discovered; if it is FALSE, then the peers list will match the *initial_peers* list, plus any peers added using the *DomainParticipant*'s `add_peer()` operation.

14.1.1 Simple Participant Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Participant Discovery Protocol (SPDP).

During the Participant Discovery phase, *DomainParticipants* learn about each other. The *DomainParticipant*'s details are communicated to all other *DomainParticipants* in the same DDS domain by sending participant declaration messages, also known as *participant DATA* submessages. The details include the *DomainParticipant*'s unique identifying key (GUID or Globally Unique ID described below), transport locators (addresses and port numbers), and QoS. These messages are sent on a periodic basis using best-effort communication.

Participant DATAs are sent periodically to maintain the liveliness of the *DomainParticipant*. They are also used to communicate changes in the *DomainParticipant*'s QoS. Only changes to QoS Policies that are part of the *DomainParticipant*'s built-in data (namely, the [USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#)) need to be propagated.

When a *DomainParticipant* is deleted, a *participant DATA (delete)* submessage with the *DomainParticipant*'s identifying GUID is sent.

The GUID is a unique reference to an entity. It is composed of a GUID prefix and an Entity ID. By default, the GUID prefix is calculated from the IP address and the process ID. (For more on how the GUID is calculated, see [Controlling How the GUID is Set \(rtps_auto_id_kind\) \(Section 8.5.9.4 on page 614\)](#).) The IP address and process ID are stored in the *DomainParticipant*'s `WIRE_PROTOCOL`

[QosPolicy \(DDS Extension\) \(Section 8.5.9 on page 610\)](#). The `entityID` is set by Connex DDS (you may be able to change it in a future version).

Once a pair of remote participants have discovered each other, they can move on to the Endpoint Discovery phase, which is how *DataWriters* and *DataReaders* find each other.

14.1.2 Simple Endpoint Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Endpoint Discovery Protocol (SEDP).

During the Endpoint Discovery phase, Connex DDS matches *DataWriters* and *DataReaders*. Information (GUID, QoS, etc.) about your application's *DataReaders* and *DataWriters* is exchanged by sending publication/subscription declarations in DATA messages that we will refer to as *publication DATAs* and *subscription DATAs*. The Endpoint Discovery phase uses reliable communication.

As described in [Discovery Implementation \(Section 14.3 on page 717\)](#), these declaration or DATA messages are exchanged until each *DomainParticipant* has a complete database of information about the participants in its peers list and their entities. Then the discovery process is complete and the system switches to a steady state. During steady state, *participant DATAs* are still sent periodically to maintain the liveness status of participants. They may also be sent to communicate QoS changes or the deletion of a *DomainParticipant*.

When a remote *DataWriter/DataReader* is discovered, Connex DDS determines if the local application has a matching *DataReader/DataWriter*. A 'match' between the local and remote entities occurs only if the *DataReader* and *DataWriter* have the same *Topic*, same data type, and compatible QoS Policies (which includes having the same partition name string, see [PARTITION QosPolicy \(Section 6.4.5 on page 323\)](#)). Furthermore, if the *DomainParticipant* has been set up to ignore certain *DataWriters/DataReaders*, those entities will not be considered during the matching process. See [Ignoring Publications and Subscriptions \(Section 16.4.2 on page 786\)](#) for more on ignoring specific publications and subscriptions.

This 'matching' process occurs as soon as a remote entity is discovered, even if the entire database is not yet complete: that is, the application may still be discovering other remote entities.

A *DataReader* and *DataWriter* can only communicate with each other if each one's application has hooked up its local entity with the matching remote entity. That is, both sides must agree to the connection.

[Discovery Implementation \(Section 14.3 on page 717\)](#) describes the details about the discovery process.

14.2 Configuring the Peers List Used in Discovery

As part of the participant phase of the discovery process, Connex DDS will announce itself within the DDS domain. Connex DDS will try to contact all possible participants in the 'initial peers list,' specified in the *DomainParticipant's* [DISCOVERY QosPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#). Note, however, it is not known if there are actually Connex DDS applications running on the hosts in the initial peers list. The initial peers list may include both unicast and multicast peer locators.

After startup, you can add to the ‘peers list’ with the **add_peer()** operation (see [Adding and Removing Peers List Entries \(Section 8.5.2.3 on page 581\)](#)). The ‘peers list’ may also grow as peers are automatically discovered (if `accept_unknown_peers` is TRUE, see [Controlling Acceptance of Unknown Peers \(Section 8.5.2.6 on page 583\)](#)).

When you call **get_default_participant_qos()** for a *DomainParticipantFactory*, the values used for the *DiscoveryQosPolicy*’s `initial_peers` and `multicast_receive_addresses` may come from the following:

- A file named **NDDS_DISCOVERY_PEERS**, which is formatted as described in [NDDS_DISCOVERY_PEERS File Format \(Section 14.2.3 on page 717\)](#). The file must be in the same directory as your application’s executable.
- An environment variable named **NDDS_DISCOVERY_PEERS**, defined as a comma-separated list of peer descriptors (see [NDDS_DISCOVERY_PEERS Environment Variable Format \(Section 14.2.2 on page 716\)](#)).
- The value specified in the default XML QoS profile (see [Configuring QoS with XML \(Section 17.4 on page 803\)](#)).

If **NDDS_DISCOVERY_PEERS** (file or environment variable) does *not* contain a multicast address, then **multicast_receive_addresses** is cleared and the RTI discovery process will not listen for discovery messages via multicast.

If **NDDS_DISCOVERY_PEERS** (file or environment variable) contains one or more multicast addresses, the addresses are stored in **multicast_receive_addresses**, starting at element 0. They will be stored in the order in which they appear in **NDDS_DISCOVERY_PEERS**.

Note: Setting **initial_peers** in the default XML QoS Profile does not modify the value of **multicast_receive_address**.

If both the file and environment variable are found, the file takes precedence and the environment variable will be ignored.¹ The settings in the default XML QoS Profile take precedence over the file and environment variable. In the absence of a file, environment variable, or default XML QoS profile values, Connext DDS will use a default value. See the API Reference HTML documentation for details (in the section on the *DISCOVERY QoS Policy*).

If initial peers are specified in both the currently loaded QoS XML profile and in the **NDDS_DISCOVERY_PEERS** file, the values in the profile take precedence.

The file, environment variable, and default XML QoS Profile make it easy to reconfigure which nodes will take part in the discovery process—without recompiling your application.

¹This is true even if the file is empty.

The file, environment variable, and default XML QoS Profile are the possible sources for the *default* initial peers list. You can, of course, explicitly set the initial list by changing the values in the QoS provided to the `DomainParticipantFactory's create_participant()` operation, or by adding to the list after startup with the `DomainParticipant's add_peer()` operation (see [Adding and Removing Peers List Entries \(Section 8.5.2.3 on page 581\)](#)).

If you set `NDDS_DISCOVERY_PEERS` and You Want to Communicate over Shared Memory:

Suppose you want to communicate with other Connex DDS applications on the same host and you are explicitly setting `NDDS_DISCOVERY_PEERS` (generally in order to use unicast discovery with applications on other hosts).

If the local host platform does *not* support the shared memory transport, then you can include the name of the local host in the `NDDS_DISCOVERY_PEERS` list. (To check if your platform supports shared memory, see the [RTI Connex DDS Core Libraries Platform Notes](#).)

If the local host platform supports the shared memory transport, then you must do one of the following:

- Include "`shmem://`" in the `NDDS_DISCOVERY_PEERS` list. This will cause shared memory to be used for discovery and data traffic for applications on the same host.

or:

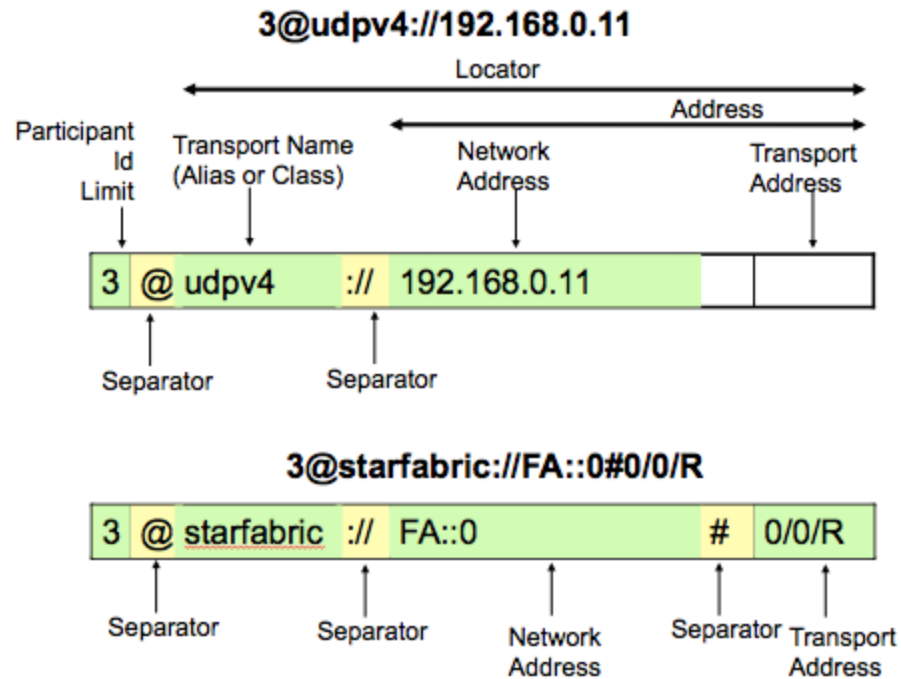
- Include the name of the local host in the `NDDS_DISCOVERY_PEERS` list, and disable the shared memory transport in the `TRANSPORT_BUILTIN QosPolicy (DDS Extension)` ([Section 8.5.7 on page 606](#)) of the `DomainParticipant`. This will cause UDP loopback to be used for discovery and data traffic for applications on the same host.

14.2.1 Peer Descriptor Format

A peer descriptor string specifies a range of participants at a given locator. Peer descriptor strings are used in the `DISCOVERY QosPolicy (DDS Extension)` ([Section 8.5.2 on page 580](#)) `initial_peers` field (see [Setting the 'Initial Peers' List \(Section 8.5.2.2 on page 581\)](#)) and the `DomainParticipant's add_peer()` and `remove_peer()` operations (see [Adding and Removing Peers List Entries \(Section 8.5.2.3 on page 581\)](#)).

The anatomy of a peer descriptor is illustrated in [Example Peer Descriptor Address Strings \(Section Figure 14.1 on the next page\)](#) using a special "StarFabric" transport example.

Figure 14.1 Example Peer Descriptor Address Strings



A peer descriptor consists of:

- *[optional]* A participant ID limit. If a simple integer is specified, it indicates the maximum participant ID to be contacted by the Connex DDS discovery mechanism at the given locator. If that integer is enclosed in square brackets (e.g., [2]), then *only* that Participant ID will be used. You can also specify a range in the form of [a-b]: in this case only the Participant IDs in that specific range are contacted. If omitted, a default value of 4 is implied and participant IDs 0, 1, 2, 3, and 4 will be contacted.
- A locator, as described in [Locator Format \(Section 14.2.1.1 below\)](#).

These are separated by the '@' character. The separator may be omitted if a participant ID limit is not explicitly specified.

The "participant ID limit" only applies to unicast locators; it is ignored for multicast locators (and therefore should be omitted for multicast peer descriptors).

14.2.1.1 Locator Format

A locator string specifies a transport and an address in string format. Locators are used to form peer descriptors. A locator is equivalent to a peer descriptor with the default participant ID limit (4).

A locator consists of:

- *[optional]* Transport name (alias or class). This identifies the set of transport plug-ins (transport aliases) that may be used to parse the address portion of the locator. Note that a transport class name is an implicit alias used to refer to all the transport plug-in instances of that class.
- *[optional]* An address, as described in [Address Format \(Section 14.2.1.2 below\)](#).

These are separated by the "://" string. The separator is specified if and only if a transport name is specified.

If a transport name is specified, the address may be omitted; in that case all the unicast addresses (across all transport plug-in instances) associated with the transport class are implied. Thus, a locator string may specify several addresses.

If an address is specified, the transport name and the separator string may be omitted; in that case all the available transport plug-ins for the *Entity* may be used to parse the address string.

The transport names for the built-in transport plug-ins are:

- shmem - Shared Memory Transport
- udpv4 - UDPv4 Transport
- udpv6 - UDPv6 Transport

14.2.1.2 Address Format

An address string specifies a transport-independent network address that qualifies a transport-dependent address string. Addresses are used to form locators. Addresses are also used in the [DISCOVERY QosPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#) `multicast_receive_addresses` and the `DDS_TransportMulticastSettings_t::receive_address` fields. An address is equivalent to a locator in which the transport name and separator are omitted.

An address consists of:

- *[optional]* A network address in IPv4 or IPv6 string notation. If omitted, the network address of the transport is implied.
- *[optional]* A transport address, which is a string that is passed to the transport for processing. The transport maps this string into `NDDS_Transport_Property_t::address_bit_count` bits. If omitted, the network address is used as the fully qualified address.

The network and transport address are separated by the '#' character. If a separator is specified, it must be followed by a non-empty string that is passed to the transport plug-in. If the separator is omitted, it is treated as a transport address with an implicit network address (of the transport plugin). The implicit network address is the address used when registering the transport: e.g., the UDPv4 implicit network address is 0.0.0.0.0.0.0.0.0.0.

The bits resulting from the transport address string are prepended with the network address. The least significant NDDS_Transport_Property_t::address_bit_count bits of the network address are ignored.

14.2.2 NDDS_DISCOVERY_PEERS Environment Variable Format

You can set the default value for the initial peers list in an environment variable named NDDS_DISCOVERY_PEERS. Multiple peer descriptor entries must be separated by commas. [Table 14.1 NDDS_DISCOVERY_PEERS Environment Variable Examples](#) shows some examples. The examples use an implied maximum participant ID of 4 unless otherwise noted. (If you need instructions on how to set environment variables, see the [RTI Connex DDS Core Libraries Getting Started Guide](#)).

Table 14.1 NDDS_DISCOVERY_PEERS Environment Variable Examples

NDDS_DISCOVERY_PEERS	Description of Host(s)
239.255.0.1	multicast
localhost	localhost
192.168.1.1	10.10.30.232 (IPv4)
FAA0::1	FAA0::0 (IPv6)
himalaya.gangotri	himalaya and gangotri
1@himalaya,1@gangotri	himalaya and gangotri (with a maximum participant ID of 1 on each host)
FAA0::0#localhost	FAA0::0#localhost (could be a UDPv4 transport plug-in registered at network address of FAA0::0) (IPv6)
udpv4://himalaya	himalaya accessed using the "udpv4" transport plug-in (IPv4)
udpv4://FAA0::0#localhost	localhost using the "udpv4" transport plug-in registered at network address FAA0::0
0/0/R #0/0/R	0/0/R (StarFabric)
starfabric://0/0/R starfabric://#0/0/R	0/0/R (StarFabric) using the "starfabric" (StarFabric) transport plug-ins
starfabric://FBB0::0#0/0/R	0/0/R (StarFabric) using the "starfabric" (StarFabric) transport plug-ins registered at network address FAA0::0
starfabric://	all unicast addresses accessed via the "starfabric" (StarFabric) transport plug-ins
shmem://FCC0::0	all unicast addresses accessed via the "shmem" (shared memory) transport plug-ins registered at network address FCC0::0

14.2.3 NDDS_DISCOVERY_PEERS File Format

You can set the default value for the initial peers list in a file named `NDDS_DISCOVERY_PEERS`. The file must be in the your application's current working directory.

The file is optional. If it is found, it supersedes the values in any environment variable of the same name.

Entries in the file must contain a sequence of peer descriptors separated by whitespace or the comma (',') character. The file may also contain comments starting with a semicolon (;) character until the end of the line.

Example file contents:

```
;; NDDS_DISCOVERY_PEERS - Discovery Configuration File
;; Multicast builtin.udpv4://239.255.0.1 ; default discovery multicast addr

;; Unicast
localhost,192.168.1.1           ; A comma can be used a separator
FAA0::1 FAA0::0#localhost      ; Whitespace can be used as a separator
1@himalaya                     ; Max participant ID of 1 on 'himalaya'
1@gangotri

;; UDPv4
udpv4://himalaya               ; 'himalaya' via 'udpv4' transport plugin(s)
udpv4://FAA0::0#localhost      ; 'localhost' via 'updv4' transport plugin
                                ; registered at network address FAA0::0

;; Shared Memory
shmem://                       ; All 'shmem' transport plugin(s)
builtin.shmem://              ; The builtin builtin 'shmem' transport plugin
shmem://FCC0::0               ; Shared memory transport plugin registered
                                ; at network address FCC0::0

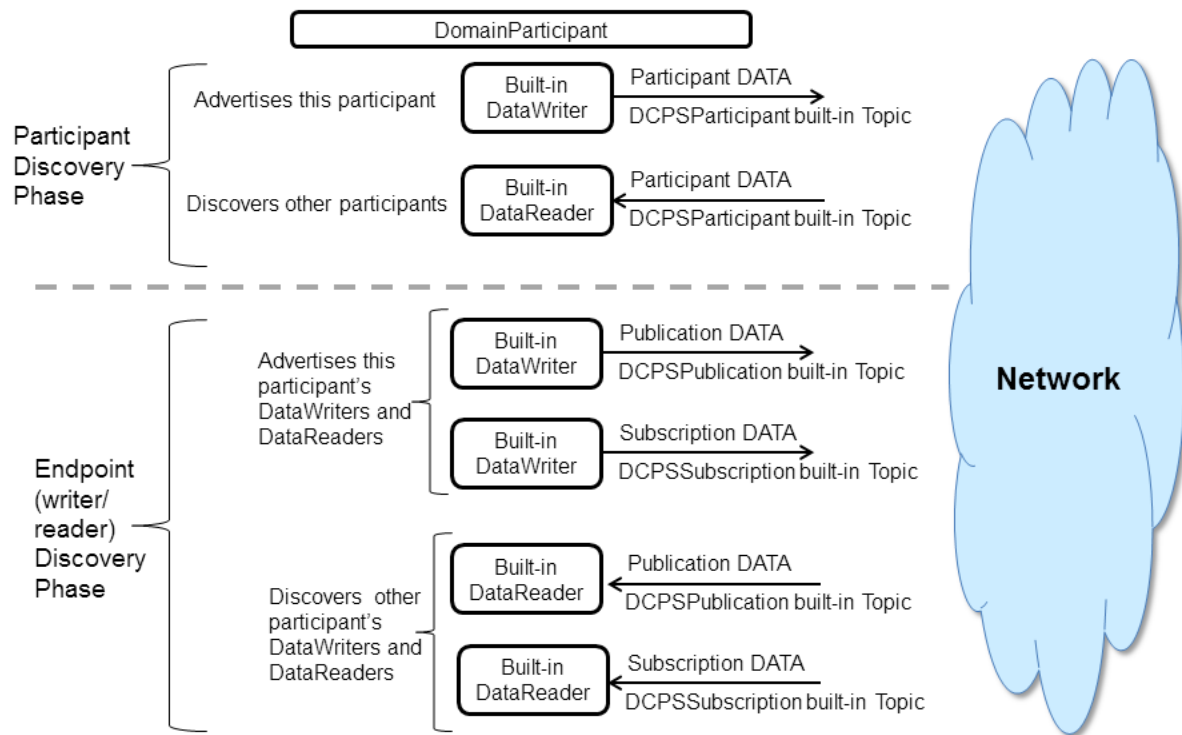
;; StarFabric
0/0/R                          ; StarFabric node 0/0/R
starfabric://0/0/R            ; 0/0/R accessed via 'starfabric'
                                ; transport plugin(s)
starfabric://FBB0::0#0/0/R    ; StarFabric transport plugin registered
                                ; at network address FBB0::0
starfabric://                 ; All 'starfabric' transport plugin(s)
```

14.3 Discovery Implementation

Note: this section contains advanced material not required by most users.

Discovery is implemented using built-in *DataWriters* and *DataReaders*. These are the same class of entities your application uses to send/receive data. That is, they are also of type `DDSDataWriter/DDSDataReader`. For each *DomainParticipant*, three built-in *DataWriters* and three *built-in DataReaders* are automatically created for discovery purposes. [Figure 14.2 Built-in Writers and Readers for Discovery on the next page](#) shows how these objects are used. (For more on built-in *DataReaders* and *DataWriters*, see [Built-In Topics \(Section Chapter 16 on page 772\)](#)).

Figure 14.2 Built-in Writers and Readers for Discovery



For each `DomainParticipant`, there are six objects automatically created for discovery purposes. The top two objects are used to send/receive `participant DATA` messages, which are used in the `Participant Discovery` phase to find remote `DomainParticipants`. This phase uses best-effort communications. Once the participants are aware of each other, they move on to the `Endpoint Discovery` phase to learn about each other's `DataWriters` and `DataReaders`. This phase uses reliable communications.

The implementation is split into two separate protocols:

Simple Participant Discovery Protocol (SPDP)

+ Simple Endpoint Discovery Protocol (SEDP)

= Simple Discovery Protocol (SDP)

14.3.1 Participant Discovery

When a `DomainParticipant` is created, a `DataWriter` and a `DataReader` are automatically created to exchange `participant DATA` messages in the network. These `DataWriters` and `DataReaders` are "special" because the `DataWriter` can send to a given list of destinations, regardless of whether there is a Connex DDS application at the destination, and the `DataReader` can receive data from any source, whether the source is previously known or not. In other words, these special readers and writers do not need to discover the remote entity and perform a match before they can communicate with each other.

When a *DomainParticipant* joins or leaves the network, it needs to notify its peer participants. The list of remote participants to use during discovery comes from the peer list described in the [DISCOVERY QoSPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#). The remote participants are notified via *participant DATA* messages. In addition, if a participant's QoS is modified in such a way that other participants need to know about the change (that is, changes to the [USER_DATA QoSPolicy \(Section 6.5.26 on page 417\)](#)), a new *participant DATA* will be sent immediately.

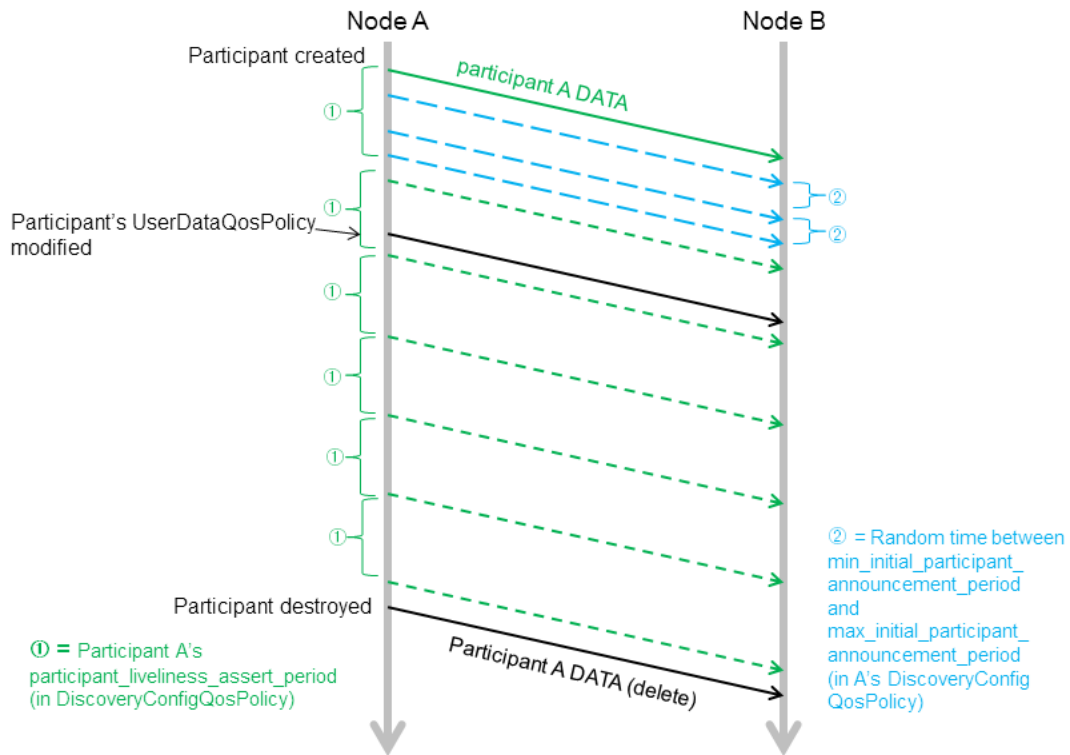
Participant DATAs are also used to maintain a participant's liveness status. These are sent at the rate set in the `participant_liveness_assert_period` in the [DISCOVERY_CONFIG QoSPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#).

Let's examine what happens when a new remote participant is discovered. If the new remote participant is in the local participant's peer list, the local participant will add that remote participant into its database. If the new remote participant is not in the local application's peer list, it may still be added, if the `accept_unknown_peers` field in the [DISCOVERY QoSPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#) is set to TRUE.

Once a remote participant has been added to the Connex DDS database, Connex DDS keeps track of that remote participant's `participant_liveness_lease_duration`. If a *participant DATA* for that participant (identified by the GUID) is not received at least once within the `participant_liveness_lease_duration`, the remote participant is considered stale, and the remote participant, together with all its entities, will be removed from the database of the local participant.

To keep from being purged by other participants, each participant needs to periodically send a *participant DATA* to refresh its liveness. The rate at which the *participant DATA* is sent is controlled by the `participant_liveness_assert_period` in the participant's [DISCOVERY_CONFIG QoSPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#). This exchange, which keeps Participant A from appearing 'stale,' is illustrated in [Figure 14.3 Periodic 'participant DATAs' on the next page](#). [Figure 14.4 Ungraceful Termination of a Participant on page 721](#) shows what happens when Participant A terminates ungracefully and therefore needs to be seen as 'stale.'

Figure 14.3 Periodic 'participant DATAs'

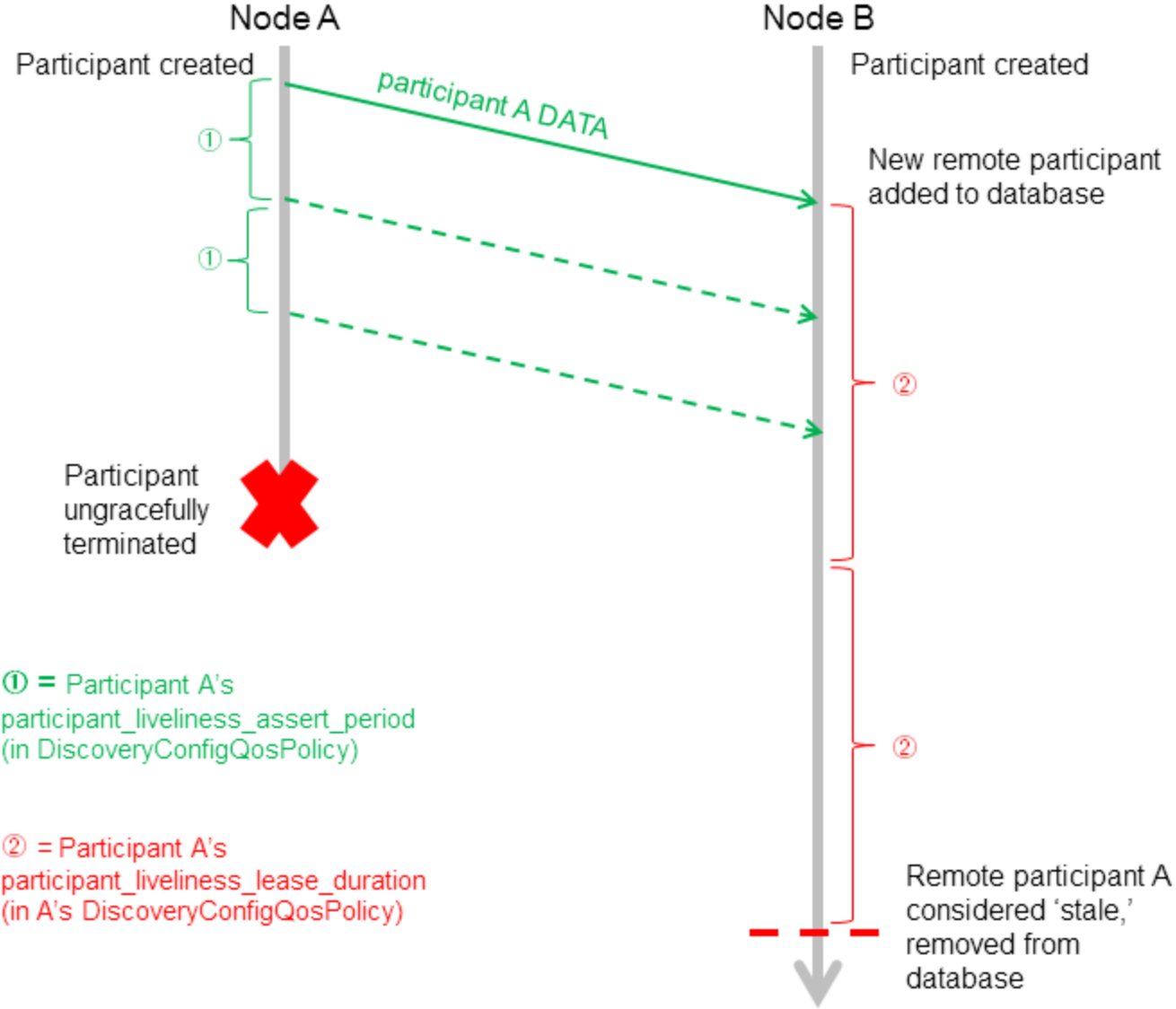


The `DomainParticipant` on Node A sends a 'participant DATA' to Node B, which is in Node A's peers list. This occurs regardless of whether or not there is a Connex DDS application on Node B.

① The green short dashed lines are periodic participant DATAs. The time between these messages is controlled by **the** `participant_liveliness_assert_period` in the `DiscoveryConfigQosPolicy`.

② In addition to the periodic participant DATAs, 'initial repeat messages' (shown in blue, with longer dashes) are sent from A to B. These messages are sent at a random time between `min_initial_participant_announcement_period` and `max_initial_participant_announcement_period` (in A's `DiscoveryConfigQosPolicy`). The number of these initial repeat messages is set in `initial_participant_announcements`.

Figure 14.4 Ungraceful Termination of a Participant



Participant A is removed from participant B's database if it is not refreshed within the liveliness lease duration. Dashed lines are periodic participant DATA messages.

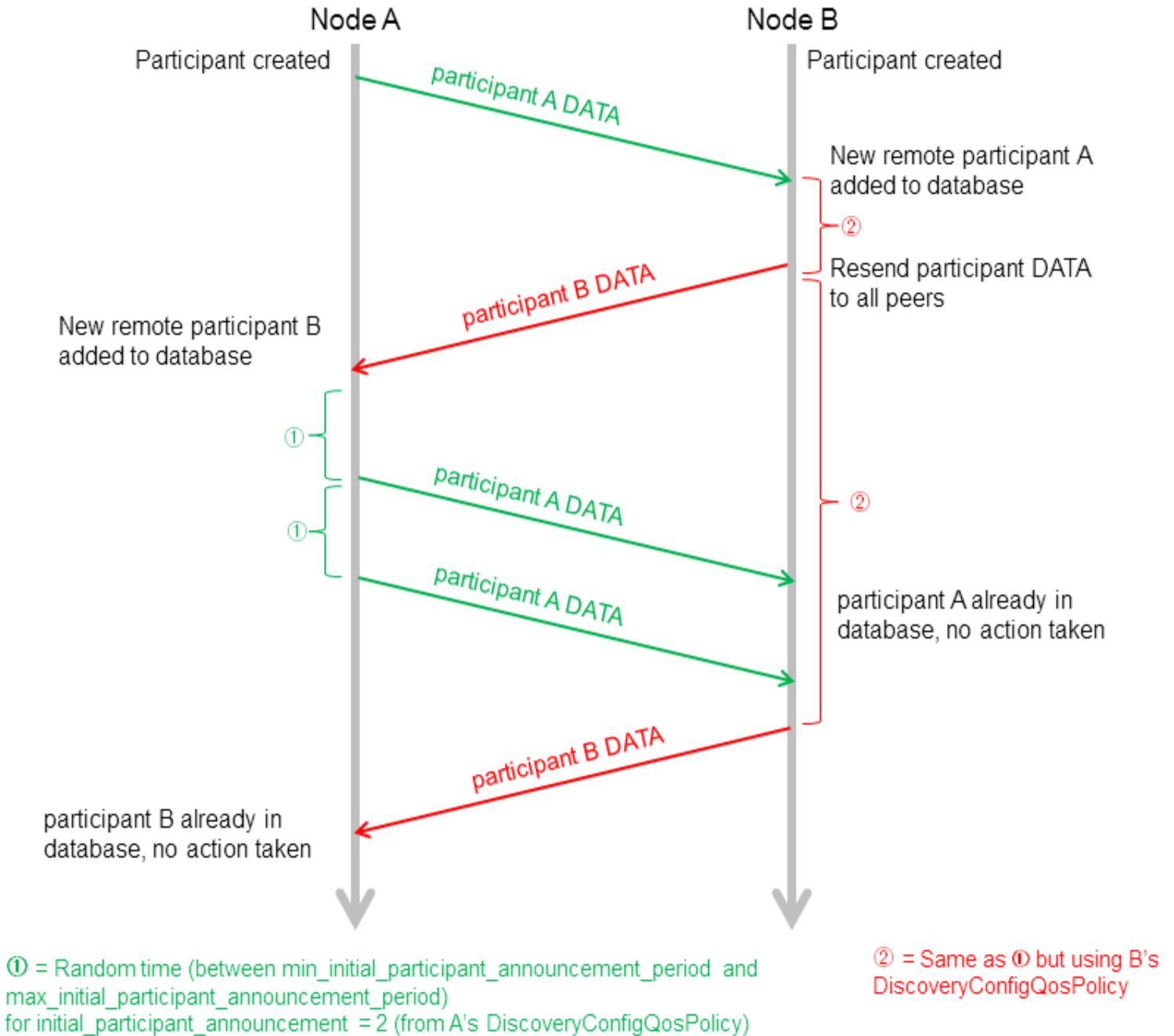
(Periodic resends of 'participant B DATA' from B to A are omitted from this diagram for simplicity. Initial repeat messages from A to B are also omitted from this diagram—these messages are sent at a random time between `min_initial_participant_announcement_period` and `max_initial_participant_announcement_period`, see [Figure 14.3 Periodic 'participant DATAs'](#) on the previous page.)

14.3.1.1 Refresh Mechanism

To ensure that a late-joining participant does not need to wait until the next refresh of the remote *participant DATA* to discover the remote participant, there is a resend mechanism. If the received *participant DATA* is from a never-before-seen remote participant, and it is in the local participant's peers list, the application will resend its own *participant DATA* to *all its peers*. This resend can potentially be done multiple times, with a random sleep time in between. [Figure 14.5 Resending 'participant DATA' to a Late-Joiner on the facing page](#) illustrates this scenario.

The number of retries and the random amount of sleep between them are controlled by each participant's [DISCOVERY_CONFIG QosPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#) (see [Figure 14.5 Resending 'participant DATA' to a Late-Joiner on the facing page](#)).

Figure 14.5 Resending 'participant DATA' to a Late-Joiner

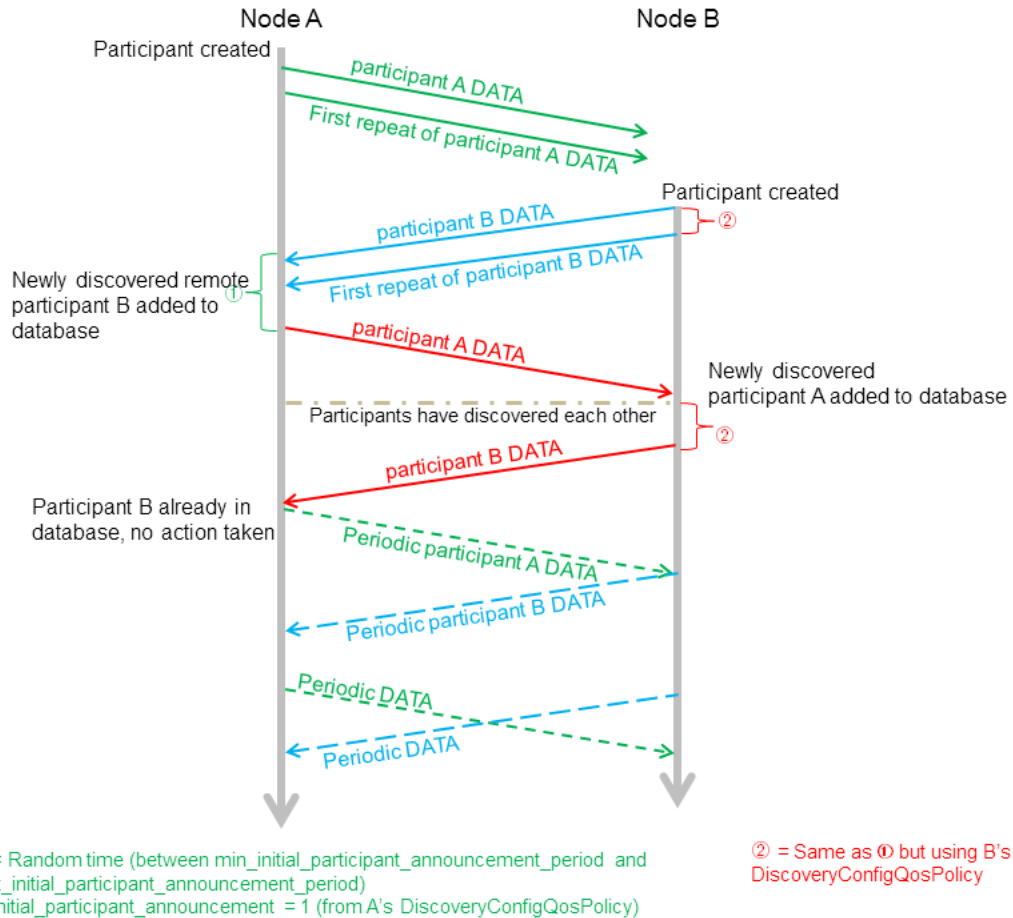


Participant A has Participant B in its peers list. Participant B does not have Participant A in its peers list, but [DiscoveryQosPolicy.accept_unknown_peers] is set to DDS_BOOLEAN_TRUE. Participant A joins the system after B has sent its initial announcement. After B discovers A, it waits for time \hat{A} , then resends its participant DATA.

(Initial repeat messages are omitted from this diagram for simplicity, see [Figure 14.3 Periodic 'participant DATAs' on page 720.](#))

Figure 14.6 Participant Discovery Summary below provides a summary of the messages sent during the participant discovery phase.

Figure 14.6 Participant Discovery Summary



Participants A and B both have each other in their peers lists. Participant A is created first.

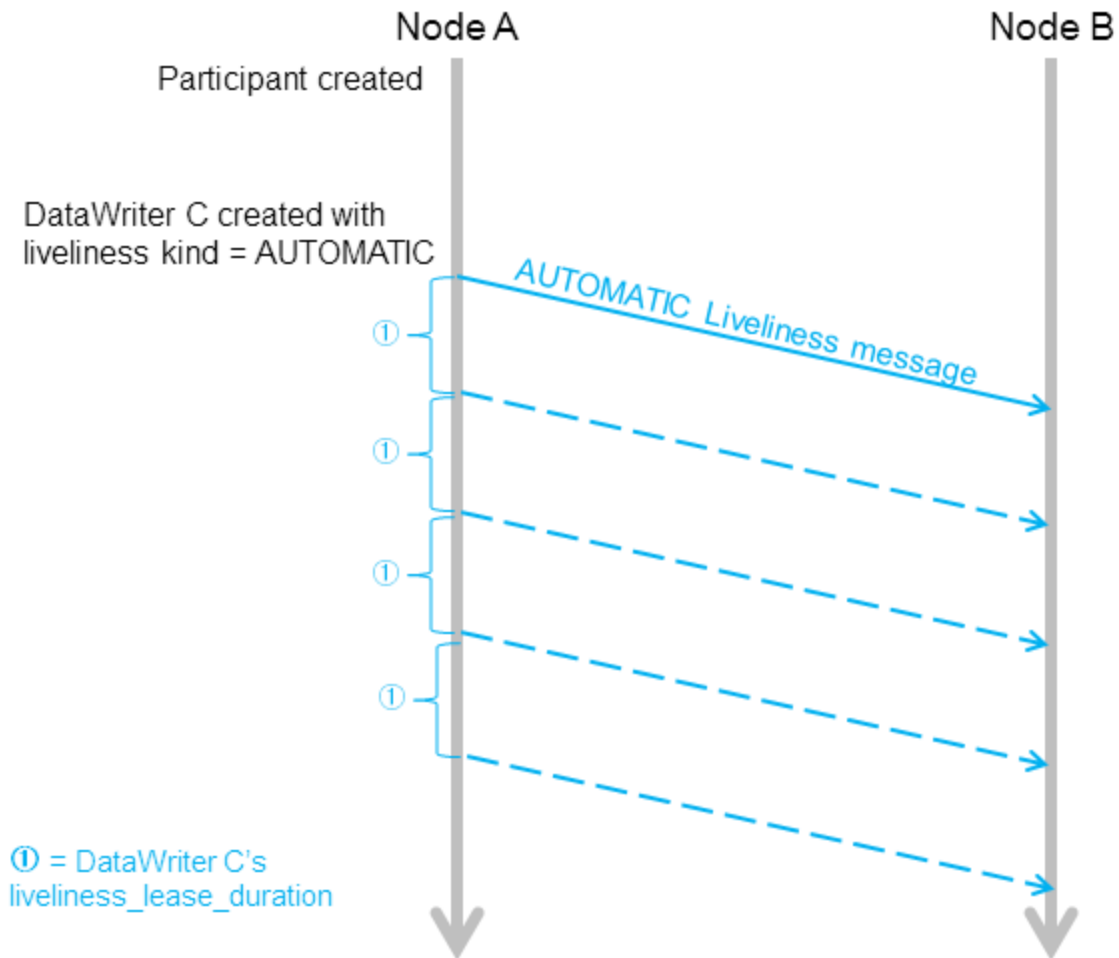
14.3.1.2 Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT

To maintain the liveliness of *DataWriters* that have a `LIVELINESS QosPolicy` (Section 6.5.13 on page 382) `kind` field set to `AUTOMATIC` or `MANUAL_BY_PARTICIPANT`, Connex DDS uses a built-in *DataWriter* and *DataReader* pair, referred to as the *inter-participant reader* and *inter-participant writer*.

If the *DomainParticipant* has any *DataWriters* with Liveliness QosPolicy `kind` set to `AUTOMATIC`, the inter-participant writer will reliably broadcast an `AUTOMATIC` liveliness message at a period equal to

the shortest **lease_duration** of these *DataWriters*. (The **lease_duration** is a field in the [LIVELINESS QoSPolicy](#) (Section 6.5.13 on page 382).) Figure 14.7 DataWriter with AUTOMATIC Liveliness below illustrates this scenario.

Figure 14.7 DataWriter with AUTOMATIC Liveliness



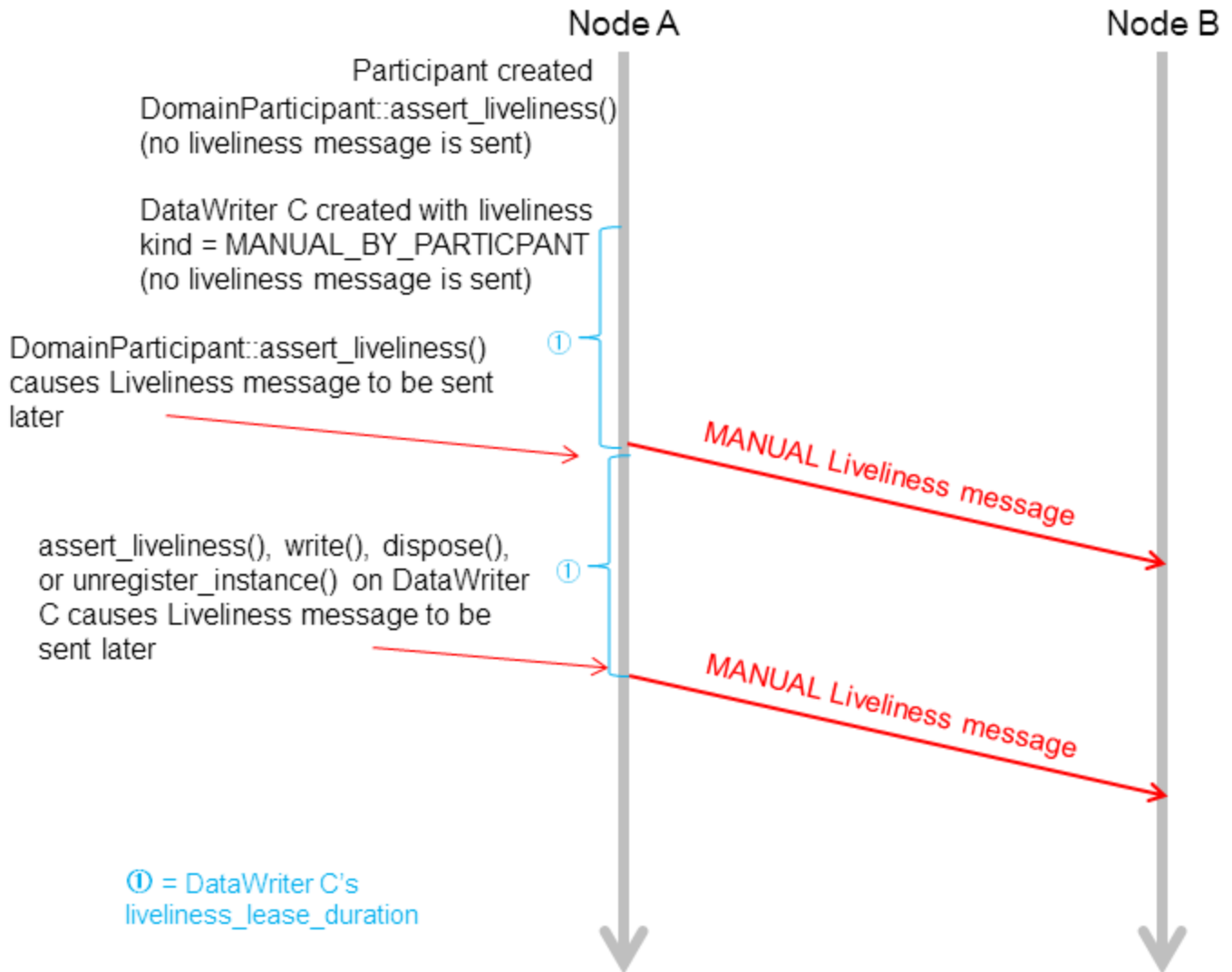
A liveliness message is sent automatically when a DataWriter with AUTOMATIC Liveliness kind is created, and then periodically, every `DDS_DataWriterQoS.liveliness.lease_duration`.

If the *DomainParticipant* has any *DataWriters* with Liveliness QoSPolicy **kind** set to **MANUAL_BY_PARTICIPANT**, Connex DDS will periodically check to see if any of them have called **write()**, **assert_liveliness()**, **dispose()** or **unregister()**. The rate of this check is every X seconds, where X is the smallest **lease_duration** among all the *DomainParticipant's* **MANUAL_BY_PARTICIPANT** *DataWriters*. (The **lease_duration** is a field in the [LIVELINESS QoSPolicy](#) (Section 6.5.13 on page

382).) If any of the **MANUAL_BY_PARTICIPANT** *DataWriters* have called any of those operations, the inter-participant writer will reliably broadcast a **MANUAL** liveliness message.

If a *DomainParticipant's* **assert_liveliness()** operation is called, and that *DomainParticipant* has any **MANUAL_BY_PARTICIPANT** *DataWriters*, the inter-participant writer will reliably broadcast a **MANUAL** liveliness message within the above-defined X time period. These **MANUAL** liveliness messages are used to update the liveliness of all the *DomainParticipant's* **MANUAL_BY_PARTICIPANT** *DataWriters*, as well as the liveliness of the *DomainParticipant* itself. [Figure 14.8 DataWriter with MANUAL_BY_PARTICIPANT Liveliness on the facing page](#) shows an example sequence.

Figure 14.8 DataWriter with MANUAL_BY_PARTICIPANT Liveliness



Once a `MANUAL_BY_PARTICIPANT` DataWriter is created, subsequent calls to `assert_liveliness`, `write`, `dispose`, or `unregister_instance` will trigger Liveliness messages, which update the liveliness status of all the participant's DataWriters, and the participant itself.

The inter-participant reader receives data from remote inter-participant writers and asserts the liveliness of remote `DomainParticipant` endpoints accordingly.

If the `DomainParticipant` has no DataWriters with `LIVELINESS` QoS Policy (Section 6.5.13 on page 382) kind set to `AUTOMATIC` or `MANUAL_BY_PARTICIPANT`, then no liveliness messages are ever sent from the inter-participant writer.

14.3.2 Endpoint Discovery

As we saw in [Built-in Writers and Readers for Discovery \(Section Figure 14.2 on page 718\)](#), reliable *DataReaders* and *Datawriters* are automatically created to exchange publication/subscription information for each *DomainParticipant*. We will refer to these as ‘discovery endpoint readers and writers.’ However, nothing is sent through the network using these entities until they have been ‘matched’ with their remote counterparts. This ‘matching’ is triggered by the Participant Discovery phase. The goal of the Endpoint Discovery phase is to add the remote endpoint to the local database, so that user-created endpoints (your application’s *DataWriters/DataReaders*) can communicate with each other.

When a new remote *DomainParticipant* is discovered and added to a participant’s database, Connex DDS assumes that the remote *DomainParticipant* is implemented in the same way and therefore is creating the appropriate counterpart entities. Therefore, Connex DDS will automatically add two remote discovery endpoint readers and two remote discovery endpoint writers for that remote *DomainParticipant* into the local database. Once that is done, there is now a match with the local discovery endpoint writers and readers, and *publication DATAs* and *subscription DATAs* can then be sent between the discovery endpoint readers/writers of the two *DomainParticipant*.

When you create a *DataWriter/DataReader* for your user data, a *publication/subscription DATA* describing the newly created object is sent from the local discovery endpoint writer to the remote discovery endpoint readers of the remote *DomainParticipants* that are currently in the local database.

If your application changes any of the following QoS Policies for a local user-data *DataWriter/DataReader*, a modified *subscription/publication DATA* is sent to propagate the QoS change to other *DomainParticipants*:

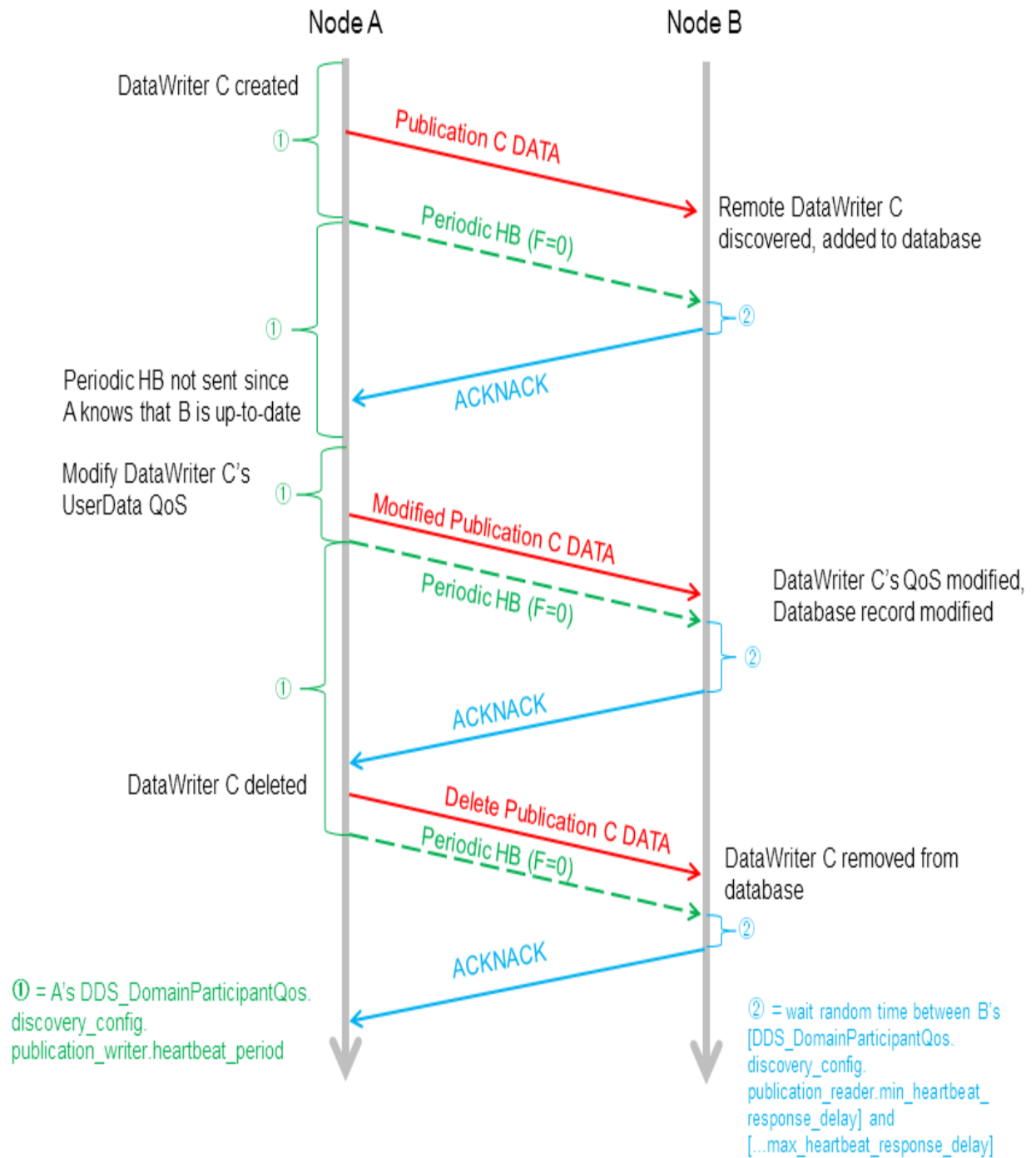
- [TOPIC_DATA QoS Policy \(Section 5.2.1 on page 209\)](#)
- [GROUP_DATA QoS Policy \(Section 6.4.4 on page 320\)](#)
- [USER_DATA QoS Policy \(Section 6.5.26 on page 417\)](#)
- [OWNERSHIP_STRENGTH QoS Policy \(Section 6.5.16 on page 393\)](#)
- [PARTITION QoS Policy \(Section 6.4.5 on page 323\)](#)
- [TIME_BASED_FILTER QoS Policy \(Section 7.6.4 on page 526\)](#)
- [LIFESPAN QoS Policy \(Section 6.5.12 on page 381\)](#)

What the above QoS Policies have in common is that they are all changeable and part of the built-in data (see [Built-In Topics \(Section Chapter 16 on page 772\)](#)).

Similarly, if the application deletes any user-data writers/readers, the discovery endpoint writer/readers send *delete publication/subscription DATAs*. In addition to sending *publication/subscription DATAs*, the discovery endpoint writer will check periodically to see if the remote discovery endpoint reader is up-to-date. (The rate for this check is the `publication_writer.heartbeat_period` or `subscription_writer.heartbeat_period` in the [DISCOVERY_CONFIG QoS Policy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#). If the

discovery endpoint writer has not been acknowledged by the remote discovery endpoint reader regarding receipt of the latest DATA, the discovery endpoint writer will send a special Heartbeat (HB) message with the Final bit set to 0 (F=0) to request acknowledgement from the remote discovery endpoint reader, as seen in [Figure 14.9 Endpoint Discovery Summary on the next page](#).

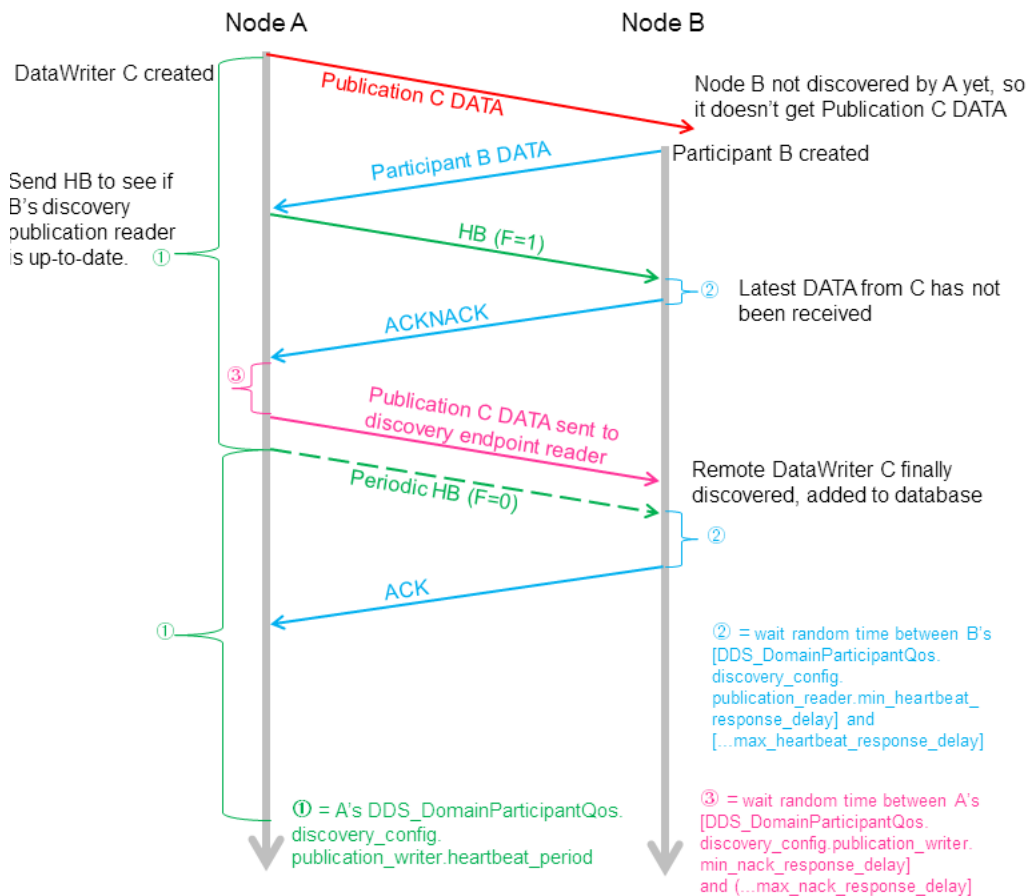
Figure 14.9 Endpoint Discovery Summary



Assume participants A and B have been discovered on both sides. A's `DiscoveryConfigQosPolicy.publication_writer_heartbeats_per_max_samples = 0`, so no HB is piggybacked with the publication DATA. A HB with $F=0$ is a request for an ACK/NACK. The periodic and initial repeat participant DATAs are omitted from the diagram.

Discovery endpoint writers and readers have their `HISTORY QosPolicy` (Section 6.5.10 on page 376) set to `KEEP_LAST`, and their `DURABILITY QosPolicy` (Section 6.5.7 on page 368) set to `TRANSIENT_LOCAL`. Therefore, even if the remote `DomainParticipant` has not yet been discovered at the time the local user's `DataWriter/DataReader` is created, the remote `DomainParticipant` will still be informed about the previously created `DataWriter/DataReader`. This is achieved by the HB and ACK/NACK that are immediately sent by the built-in endpoint writer and built-in endpoint reader respectively when a new remote participant is discovered. Figure 14.10 `DataWriter Discovered by Late-Joiner, Triggered by HB` below and Figure 14.11 `DataWriter Discovered by Late-Joiner, Triggered by ACKNACK` on the next page illustrate this sequence for HB and ACK/NACK triggers, respectively.

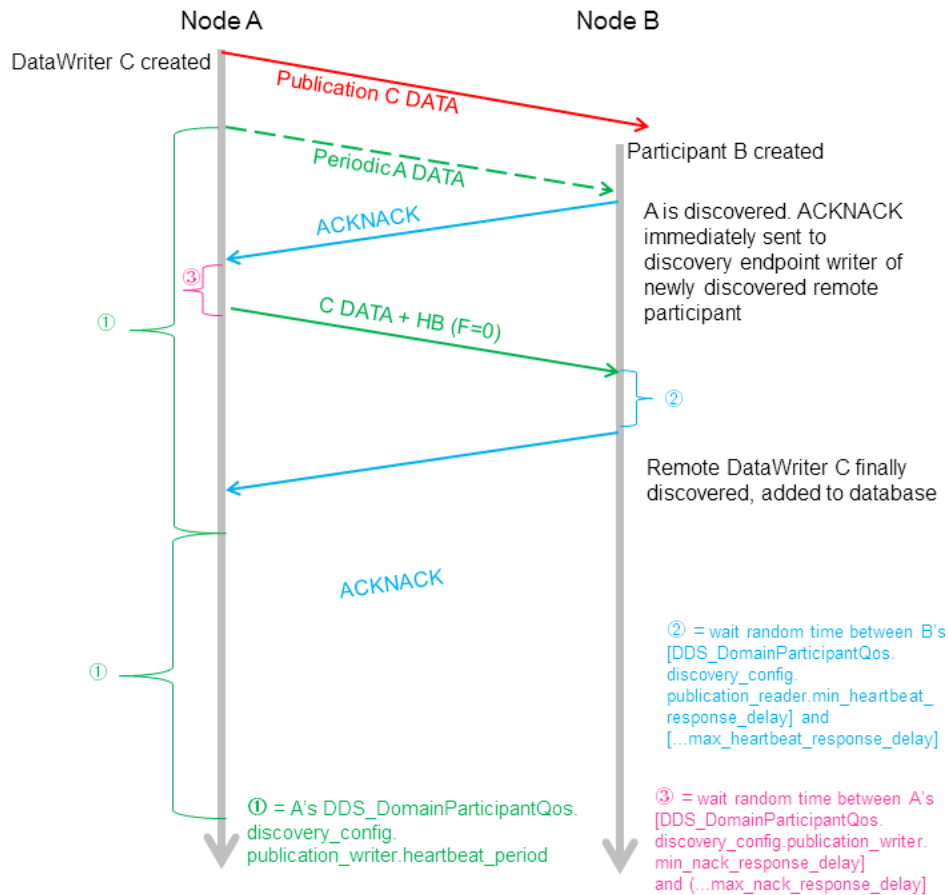
Figure 14.10 `DataWriter Discovered by Late-Joiner, Triggered by HB`



Writer C is created on Participant A before Participant A discovers Participant B. Assuming `DiscoveryConfigQosPolicy.publication_writer.heartbeats_per_max_samples = 0`, no HB is piggybacked with the publication DATA. Participant B has A in its peer list, but not vice versa. `Accept_unknown_locators` is true. On A, in response to receiving the new participant B DATA message, a participant A DATA message is sent to B. The discovery endpoint

reader on A will also send an ACK/NACK to the discovery endpoint writer on B. (Initial repeat participant messages and periodic participant messages are omitted from this diagram for simplicity, see [Figure 14.3 Periodic 'participant DATAs' on page 720 in Participant Discovery \(Section 14.3.1 on page 718\).](#))

Figure 14.11 DataWriter Discovered by Late-Joiner, Triggered by ACKNACK



Writer C is created on Participant A before Participant A discovers Participant B. Assuming `DiscoveryConfigQosPolicy.publication_writer.heartbeats_per_max_samples = 0`, no HB is piggybacked with the publication DATA message. Participant A has B in its peer list, but not vice versa. `Accept_unknown_locators` is true. In response to receiving the new Participant A DATA message on node B, a participant B DATA message will be sent to A. The discovery endpoint writer on Node B will also send a HB to the discovery endpoint reader on Node A. These are omitted in the diagram for simplicity. (Initial repeat participant messages and periodic participant messages are omitted from this diagram, see [Figure 14.3 Periodic 'participant DATAs' on page 720 in Participant Discovery \(Section 14.3.1 on page 718\).](#))

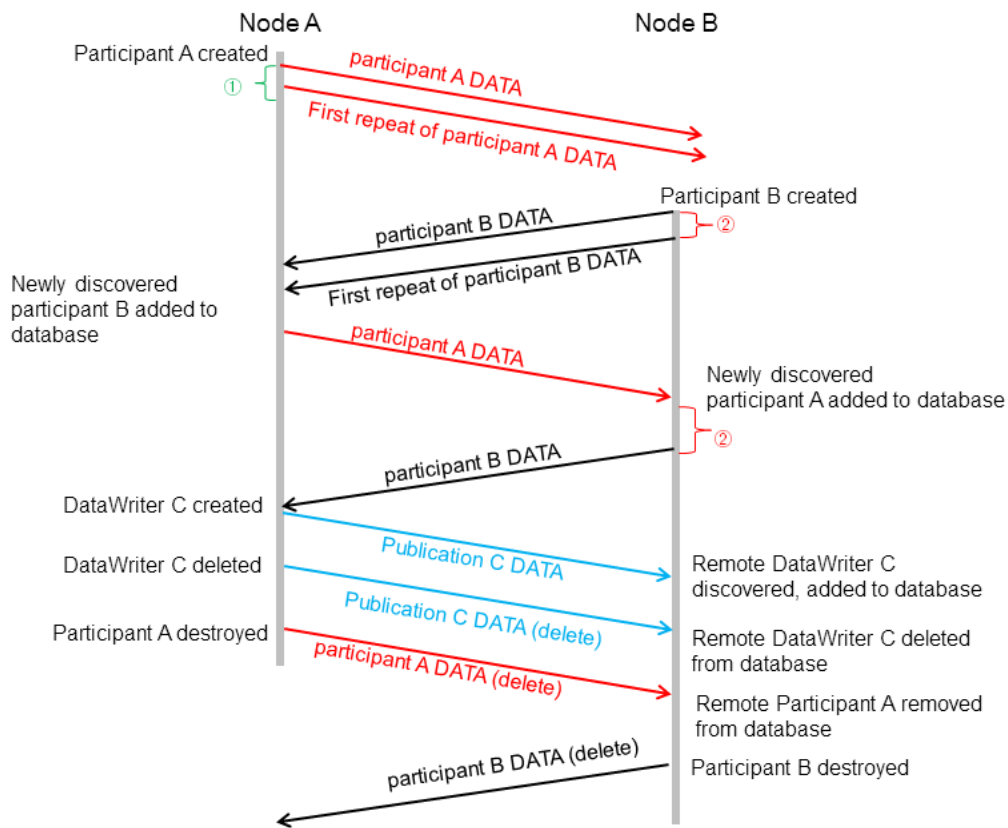
Endpoint discovery latency is determined by the following members of the `DomainParticipant's DISCOVERY_CONFIG QosPolicy (DDS Extension)` (Section 8.5.3 on page 585):

- `publication_writer`
- `subscription_writer`
- `publication_reader`
- `subscription_reader`

When a remote entity record is added, removed, or changed in the database, matching is performed with all the local entities. Only after there is a successful match on both ends can an application’s user-created *DataReaders* and *DataWriters* communicate with each other.

For more information about reliable communication, see [Reliable Communications \(Section Chapter 10 on page 629\)](#).

14.3.3 Discovery Traffic Summary



① = Random time (between `min_initial_participant_announcement_period` and `max_initial_participant_announcement_period`) for `initial_participant_announcement = 1` (from A's `DiscoveryConfigQosPolicy`)

② = Same as ① but using B's `DiscoveryConfigQosPolicy`

This diagram shows both phases of the discovery process. Participant A is created first, followed by Participant B. Each has the other in its peers list. After they have discovered each other, a DataWriter is created on Participant A. Periodic participant DATAs, HBs and ACK/NACKs are omitted from this diagram.

14.3.4 Discovery-Related QoS

Each *DomainParticipant* needs to be uniquely identified in the DDS domain and specify which other *DomainParticipants* it is interested in communicating with. The [WIRE_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 8.5.9 on page 610\)](#) uniquely identifies a *DomainParticipant* in the DDS domain. The [DISCOVERY QoSPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#) specifies the peer participants it is interested in communicating with.

There is a trade-off between the amount of traffic on the network for the purposes of discovery and the delay in reaching steady state when the *DomainParticipant* is first created.

For example, if the [DISCOVERY QoSPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#)'s `participant_liveliness_assert_period` and `participant_liveliness_lease_duration` fields are set to small values, the discovery of stale remote *DomainParticipants* will occur faster, but more discovery traffic will be sent over the network. Setting the participant's `heartbeat_period`¹ to a small value can cause late-joining *DomainParticipants* to discover remote user-data *DataWriters* and *DataReaders* at a faster rate, but Connext DDS might send HBs to other nodes more often. This timing can be controlled by the following *DomainParticipant* QoS Policies:

- [DISCOVERY QoSPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#) — specifies how other *DomainParticipants* in the network can communicate with this *DomainParticipant*, and which other *DomainParticipants* in the network this *DomainParticipant* is interested in communicating with. See also: [Ports Used for Discovery \(Section 14.5 on page 738\)](#).
- [DISCOVERY_CONFIG QoSPolicy \(DDS Extension\) \(Section 8.5.3 on page 585\)](#) — specifies the QoS of the discovery readers and writers (parameters that control the HB and ACK rates of discovery endpoint readers/writers, and periodic refreshing of *participant DATA* from discovery participant readers/writers). It also allow you to configure asynchronous writers in order to send data with a larger size than the transport message size.
- [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QoSPolicy \(DDS Extension\) \(Section 8.5.4 on page 593\)](#) — specifies the number of local and remote entities expected in the system.
- [WIRE_PROTOCOL QoSPolicy \(DDS Extension\) \(Section 8.5.9 on page 610\)](#) — specifies the `rtps_app_id` and `rtps_host_id` that uniquely identify the participant in the DDS domain.

The other important parameter is the domain ID: *DomainParticipants* can only discover each other if they belong to the same DDS domain. The domain ID is a parameter passed to the `create_participant()` operation (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#)).

¹`heartbeat_period` is part of the `DDS_RtpsReliableWriterProtocol_t` structure used in the [DISCOVERY QoSPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#)'s `publication_writer` and `subscription_writer` fields.

14.4 Debugging Discovery

To understand the flow of messages during discovery, you can increase the verbosity of the messages logged by Connex DDS so that you will see whenever a new entity is discovered, and whenever there is a match between a local entity and a remote entity.

This can be achieved with the logging API:

```
NDDConfigLogger::get_instance()->set_verbosity_by_category (NDDS_CONFIG_LOG_CATEGORY_ENTITIES, NDDS_CONFIG_LOG_VERBOSITY_STATUS_REMOTE);
```

Using the scenario in the summary diagram in [Discovery Traffic Summary \(Section 14.3.3 on page 733\)](#), these are the messages as seen on DomainParticipant A:

```
[D0049|ENABLE]DISCPluginManager_onAfterLocalParticipantEnabled:announcing new local participant: 0XA0A01A1,0X5522,0X1,0X1C1
```

```
[D0049|ENABLE]DISCPluginManager_onAfterLocalParticipantEnabled:at {46c614d9,0C43B2DC}
```

(The above messages mean: First participant A DATA sent out when participant A is enabled.)

```
DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:discovered new participant: host=0x0A0A01A1, app=0x0000552B, instance=0x00000001
```

```
DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:at {46c614dd,8FA13C1F}
```

```
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin discovered/updated remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
```

```
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at {46c614dd,8FACE677}
```

```
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin accepted new remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
```

```
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at {46c614dd,8FACE677}
```

(The above messages mean: Received participant B DATA.)

```
DISCSimpleParticipantDiscoveryPlugin_remoteParticipantDiscovered:re-announcing participant self: 0XA0A01A1,0X5522,0X1,0X1C1
```

```
DISCSimpleParticipantDiscoveryPlugin_remoteParticipantDiscovered:at {46c614dd,8FC02AF7}
```

(The above messages mean: Resending participant A DATA to the newly discovered remote participant.)

```
PRESPEsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X200C2, local 0x000200C7 in reliable reader service
```

```
PRESPEsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X200C7, local 0x000200C2 in reliable writer service
```

```

PRESpsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X4C7, local 0x000004C2 in
reliable writer service

PRESpsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X3C7, local 0x000003C2 in
reliable writer service

PRESpsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X4C2, local 0x000004C7 in
reliable reader service

PRESpsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X3C2, local 0x000003C7 in
reliable reader service

PRESpsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X100C2, local 0x000100C7
in best effort reader service

```

(The above messages mean: Automatic matching of the discovery readers and writers. A built-in remote endpoint's object ID always ends with Cx.)

```

DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:discovered modified
participant: host=0x0A0A01A1, app=0x0000552B, instance=0x00000001

DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin discovered/updated remote
participant: 0XA0A01A1,0X552B,0X1,0X1C1

DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at {46c614dd,904D876C}

```

(The above messages mean: Received participant B DATA.)

```

DISCPluginManager_onAfterLocalEndpointEnabled:announcing new local publication:
0XA0A01A1,0X5522,0X1,0X80000003

DISCPluginManager_onAfterLocalEndpointEnabled:at {46c614d9,1013B9F0}

DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLocalWriterEnabled:announcing new
publication: 0XA0A01A1,0X5522,0X1,0X80000003

DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLocalWriterEnabled:at {46c614d9,101615EB}

```

(The above messages mean: Publication C DATA has been sent.)

```

DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAvailable:discovered
subscription: 0XA0A01A1,0X552B,0X1,0X80000004

DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAvailable:at
{46c614dd,94FAEFEF}

DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:plugin discovered/updated remote endpoint:
0XA0A01A1,0X552B,0X1,0X80000004

DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:at {46c614dd,950203DF}

```

(The above messages mean: Receiving subscription D DATA from Node B.)

```
PRESPEsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X80000004, local
0x80000003 in best effort writer service
```

(The above message means: User-created DataWriter C and DataReader D are matched.)

```
[D0049|DELETE_CONTAINED]DISCPluginManager_onAfterLocalEndpointDeleted:announcing disposed
local publication: 0XA0A01A1,0X5522,0X1,0X80000003

[D0049|DELETE_CONTAINED]DISCPluginManager_onAfterLocalEndpointDeleted:at {46c61501,288051C8}

[D0049|DELETE_CONTAINED]DISCSimpleEndpointDiscoveryPluginPDFListener_
onAfterLocalWriterDeleted:announcing disposed publication: 0XA0A01A1,0X5522,0X1,0X80000003

[D0049|DELETE_CONTAINED]DISCSimpleEndpointDiscoveryPluginPDFListener_
onAfterLocalWriterDeleted:at {46c61501,28840E15}
```

(The above messages mean: Publication C DATA(delete) has been sent.)

```
DISCPluginManager_onBeforeLocalParticipantDeleted:announcing before disposed local
participant: 0XA0A01A1,0X5522,0X1,0X1C1

DISCPluginManager_onBeforeLocalParticipantDeleted:at {46c61501,28A11663}
```

(The above messages mean: Participant A DATA(delete) has been sent.)

```
DISCParticipantDiscoveryPlugin_removeRemoteParticipantsByCookie:plugin removing 3 remote
entities by cookie

DISCParticipantDiscoveryPlugin_removeRemoteParticipantsByCookie:at {46c61501,28E38A7C}

DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:plugin discovered disposed remote
participant: 0XA0A01A1,0X552B,0X1,0X1C1

DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:at {46c61501,28E68E3D}

DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:remote entity removed from database:
0XA0A01A1,0X552B,0X1,0X1C1

DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:at {46c61501,28E68E3D}
```

(The above messages mean: Removing discovered entities from local database, before shutting down.)

As you can see, the messages are encoded, since they are primarily used by RTI support personnel.

For more information on the message logging API, see [Controlling Messages from Connex DDS \(Section 21.2 on page 865\)](#).

If you notice that a remote entity is not being discovered, check the QoS related to discovery (see [Discovery-Related QoS \(Section 14.3.4 on page 734\)](#)).

If a remote entity is discovered, but does not match with a local entity as expected, check the QoS of both the remote and local entity.

14.5 Ports Used for Discovery

There are two kinds of traffic in a Connex DDS application: discovery (meta) traffic, and user traffic. Meta-traffic is for data (declarations) that is sent between the automatically-created discovery writers and readers; user traffic is for data that is sent between user-created *DataWriters* and *DataReaders*. To keep the two kinds of traffic separate, Connex DDS uses different ports, as described below.

Note: The ports described in this section are used for *incoming* data. Connex DDS uses ephemeral ports for outbound data.

Connex DDS uses the RTPS wire protocol. The discovery protocols defined by RTPS rely on well-known ports to initiate discovery. These well-known ports define the multicast and unicast ports on which a Participant will listen for meta-traffic from other Participants. The meta-traffic contains the information required by Connex DDS to establish the presence of remote *Entities* in the network.

The well-known incoming ports are defined by RTPS in terms of port mapping expressions with several tunable parameters. This allows you to customize what network ports are used for receiving data by Connex DDS. These parameters are shown in [Table 14.2 WireProtocol QosPolicy's rtps_well_known_ports \(DDS_RtpsWellKnownPorts_t\)](#). (For defaults and valid ranges, please see the API Reference HTML documentation.)

Table 14.2 WireProtocol QosPolicy's rtps_well_known_ports (DDS_RtpsWellKnownPorts_t)

Type	Field Name	Description
DDS_ Long	port_base	The base port offset. All mapped well-known ports are offset by this value. Resulting ports must be within the range imposed by the underlying transport.
	domain_id_gain	Tunable gain parameters. See Tuning domain_id_gain and participant_id_gain (Section 14.5.4 on page 740).
	participant_id_gain	
	builtin_multicast_port_offset	Additional offset for meta-traffic port. See Inbound Ports for Meta-Traffic (Section 14.5.1 on the facing page).
	builtin_unicast_port_offset	
	user_multicast_port_offset	Additional offset for user traffic port. See Inbound Ports for User Traffic (Section 14.5.2 on page 740).
	user_unicast_port_offset	

In order for all Participants in a system to correctly discover each other, it is important that they all use the same port mapping expressions.

In addition to the parameters listed in [Table 14.2 WireProtocol QosPolicy's rtps_well_known_ports \(DDS_RtpsWellKnownPorts_t\)](#), the port formulas described below depend on:

- The domain ID specified when the *DomainParticipant* is created (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#)). The domain ID ensures no port conflicts exist between Participants belonging to different domains. This also means that discovery traffic in one DDS domain is not visible to *DomainParticipants* in other DDS domains.
- The **participant_id** is a field in the [WIRE_PROTOCOL QosPolicy \(DDS Extension\) \(Section 8.5.9 on page 610\)](#), see [Choosing Participant IDs \(Section 8.5.9.1 on page 611\)](#). The **participant_id** ensures that unique unicast port numbers are assigned to *DomainParticipants* belonging to the same DDS domain on a given host.

Backwards Compatibility: Connex DDS supports the standard DDS Interoperability Wire Protocol based on the Real-time Publish-Subscribe (RTPS) protocol. This protocol is not compatible with the one used by earlier releases (4.2c or lower). Therefore, applications built with 4.2d or higher will not interoperate with applications built with 4.2c or lower. The default port mapping from domainID and participant index has also been changed according to the new interoperability specification. The message types and formats used by RTPS have also changed.

Port Aliasing: When modifying the port mapping parameters, *avoid port aliasing*. This would result in undefined discovery behavior. The chosen parameter values will also determine the maximum possible number of DDS domains in the system and the maximum number of participants per DDS domain. Additionally, any resulting mapped port number must be within the range imposed by the underlying transport. For example, for UDPv4, this range typically equals [1024 - 65535].

14.5.1 Inbound Ports for Meta-Traffic

The Wire Protocol QosPolicy's `rtps_well_known_ports.metatraffic_unicast_port` determines the port used for receiving meta-traffic using unicast:

```
metatraffic_unicast_port = port_base +
    (domain_id_gain * Domain ID) +
    (participant_id_gain * participant_id) +
    builtin_unicast_port_offset
```

Similarly, `rtps_well_known_ports.metatraffic_multicast_port` determines the port used for receiving meta-traffic using multicast. The corresponding multicast group addresses are specified via **multicast_receive_addresses** (see [Configuring Multicast Receive Addresses \(Section 8.5.2.4 on page 582\)](#)).

```
metatraffic_multicast_port = port_base +
    (domain_id_gain * Domain ID) +
    builtin_multicast_port_offset
```


Note: Multicast is only used for meta-traffic if a multicast address is specified in the `NDDS_DISCOVERY_PEERS` environment variable or file or if the `multicast_receive_addresses` field of the `DISCOVERY_CONFIG QosPolicy (DDS Extension)` (Section 8.5.3 on page 585) is set.

14.5.2 Inbound Ports for User Traffic

RTPS also defines the default multicast and unicast ports on which *DataReaders* and *DataWriters* receive user traffic. These default ports can be overridden using the *DataReader's* `TRANSPORT_MULTICAST QosPolicy (DDS Extension)` (Section 7.6.5 on page 529) and `TRANSPORT_UNICAST QosPolicy (DDS Extension)` (Section 6.5.24 on page 412), or the *DataWriter's* `TRANSPORT_UNICAST QosPolicy (DDS Extension)` (Section 6.5.24 on page 412).

The WireProtocol QosPolicy's `rtps_well_known_ports.usertraffic_unicast_port` determines the port used for receiving user data using unicast:

```
usertraffic_unicast_port =
    port_base +
    (domain_id_gain * Domain ID) +
    (participant_id_gain * participant_id) +
    user_unicast_port_offset
```

Similarly, `rtps_well_known_ports.usertraffic_multicast_port` determines the port used for receiving user data using multicast. The corresponding multicast group addresses can be configured using the `TRANSPORT_UNICAST QosPolicy (DDS Extension)` (Section 6.5.24 on page 412).

```
usertraffic_multicast_port =
    port_base +
    (domain_id_gain * Domain ID) +
    user_multicast_port_offset
```

14.5.3 Automatic Selection of participant_id and Port Reservation

The `WIRE_PROTOCOL QosPolicy (DDS Extension)` (Section 8.5.9 on page 610) `rtps_reserved_ports_mask` field determines what type of ports are reserved when the *DomainParticipant* is enabled. See [Choosing Participant IDs \(Section 8.5.9.1 on page 611\)](#).

14.5.4 Tuning domain_id_gain and participant_id_gain

The `domain_id_gain` is used as a multiplier of the domain ID. Together with `participant_id_gain` ([Tuning domain_id_gain and participant_id_gain \(Section 14.5.4 above\)](#)), these values determine the highest domain ID and `participant_id` allowed on this network.

In general, there are two ways to set up the `domain_id_gain` and `participant_id_gain` parameters.

- If `domain_id_gain > participant_id_gain`, it results in a port mapping layout where all *DomainParticipants* in a DDS domain occupy a consecutive range of `domain_id_gain` ports.

Precisely, all ports occupied by the DDS domain fall within:

```
(port_base + (domain_id_gain * Domain ID))
```

and:

```
(port_base + (domain_id_gain * (Domain ID + 1)) - 1)
```

In this case, the highest domain ID is limited only by the underlying transport's maximum port. The highest **participant_id**, however, must satisfy:

```
max_participant_id < (domain_id_gain / participant_id_gain)
```

- Or if **domain_id_gain** <= **participant_id_gain**, it results in a port mapping layout where a given DDS domain's *DomainParticipant* instances occupy ports spanned across the entire valid port range allowed by the underlying transport. For instance, it results in the following potential mapping:

Mapped Port	Domain ID	Participant ID
higher port number	1	2
	0	
	1	1
	0	
lower port number	1	0
	0	

In this case, the highest **participant_id** is limited only by the underlying transport's maximum port. The highest **domain_id**, however, must satisfy:

```
max_domain_id < (participant_id_gain / domain_id_gain)
```

The **domain_id_gain** also determines the range of the port-specific offsets:

```
domain_id_gain >
abs(builtin_multicast_port_offset - user_multicast_port_offset)
```

and

```
domain_id_gain >
abs(builtin_unicast_port_offset - user_unicast_port_offset)
```

Violating this may result in port aliasing and undefined discovery behavior.

The **participant_id_gain** also determines the range of **builtin_unicast_port_offset** and **user_unicast_port_offset**.

```
participant_id_gain >  
abs(builtin_unicast_port_offset - user_unicast_port_offset)
```

In all cases, the resulting ports must be within the range imposed by the underlying transport.

Chapter 15 Transport Plugins

Connex DDS has a pluggable-transport architecture. The core of Connex DDS is transport agnostic—it does not make any assumptions about the actual transports used to send and receive messages. Instead, Connex DDS uses an abstract "transport API" to interact with the transport plugins that implement that API. A transport plugin implements the abstract transport API, and performs the actual work of sending and receiving messages over a physical transport.

There are essentially three categories of transport plugins:

- **Builtin Transport Plugins** Connex DDS comes with a set of commonly used transport plugins. These 'builtin' plugins include UDPv4, UDPv6, and shared memory. So that Connex DDS applications can work out-of-the-box, some of these are enabled by default (see [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\) \(Section 8.5.7 on page 606\)](#)).
- **Extension Transport Plugins** RTI offers extension transports, including *RTI Secure WAN Transport* (see [Part 5: RTI Secure WAN Transport \(Section on page 900\)](#)) and *RTI TCP Transport* (see [Part 8: RTI TCP Transport \(Section on page 987\)](#)).
- **Custom-developed Transport Plugins** RTI supports the use of custom transport plugins. This is a powerful capability that distinguishes Connex DDS from competing middleware approaches. If you are interested in developing a custom transport plugin for Connex DDS, please contact your local RTI representative or email sales@rti.com.

15.1 Builtin Transport Plugins

There are two ways in which the builtin transport plugins may be registered:

- **Default builtin Transport Instances:** Builtin transports that are turned "on" in the [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\) \(Section 8.5.7 on page 606\)](#) are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter/DataReader* is created, or (c) you look up a builtin *DataReader* (by calling `lookup_datareader()` on a Subscriber), whichever happens first. The builtin transport plugins have

default properties. If you want to change these properties, do so *before*¹ the transports are registered.

- **Other Transport Instances:** There are two ways to install non-default builtin transport instances:
 - Transport plugins may be explicitly registered by first creating an instance of the transport plugin (by calling `NDDS_Transport_UDPv4_new()`, `NDDS_Transport_UDPv6_new()` or `NDDS_Transport_Shmem_new()`, see [Explicitly Creating Builtin Transport Plugin Instances \(Section 15.4 on page 746\)](#)), then calling `register_transport()` ([Installing Additional Builtin Transport Plugins with register_transport\(\) \(Section 15.7 on page 765\)](#)). (For example, suppose you want an extra instance of a transport.) (Not available for the Java or .NET API.)
 - Additional builtin transport instances can also be installed through the [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#).

To configure the properties of the builtin transports:

- Set properties by calling `set_builtin_transport_property()` (see [Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties\(\) \(Section 15.5 on page 746\)](#))

or

- Specify predefined property strings in the *DomainParticipant's* `PropertyQosPolicy`, as described in [Setting Builtin Transport Properties with the PropertyQosPolicy \(Section 15.6 on page 748\)](#).

For other builtin transport instances:

- If the builtin transport plugin is created with `NDDS_Transport_UDPv4_new()`, `NDDS_Transport_UDPv6_new()` or `NDDS_Transport_Shmem_new()`, properties can be specified during creation time. See [Explicitly Creating Builtin Transport Plugin Instances \(Section 15.4 on page 746\)](#).
- If the additional builtin transport instances are installed through the [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#), the properties of the builtin transport plugins can also be specified through that same `QosPolicy`.

15.2 Extension Transport Plugins

If you want to change the properties for an extension transport plugin, do so *before* the plugin is registered. Any transport property changes made after the plugin is registered will have no effect.

There are two ways to install an extension transport plugin:

¹Any transport property changes made after the plugin is registered will have no effect.

- **Implicit Registration:** Transports can be installed through the predefined strings in the *DomainParticipant's* PropertyQoSPolicy. Once the transport's properties are specified in the PropertyQoSPolicy, the transport will be implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter/DataReader* is created, or (c) you look up a builtin *DataReader* (by calling **lookup_datareader()** on a Subscriber), whichever happens first.

QoS Policies can also be configured from XML resources (files, strings)—with this approach, you can change the QoS without recompiling the application. The QoS settings are automatically loaded by the DomainParticipantFactory when the first *DomainParticipant* is created. For more information, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

- **Explicit Registration:** Transports may be explicitly registered by first creating an instance of the transport plugin (see [Explicitly Creating Builtin Transport Plugin Instances \(Section 15.4 on the next page\)](#)) and then calling **register_transport()** (see [Installing Additional Builtin Transport Plugins with register_transport\(\)](#) (Section 15.7 on page 765)).

15.3 The NDDSTransportSupport Class

The **register_transport()** and **set_builtin_transport_property()** operations are part of the **NDDSTransportSupport** class, which includes the operations listed in [Table 15.1 Transport Support Operations](#).

Table 15.1 Transport Support Operations

Operation	Description	Reference
get_transport_plugin	Retrieves a previously registered transport plugin.	Installing Additional Builtin Transport Plugins with register_transport() (Section 15.7 on page 765)
register_transport	Registers a transport plugin for use with a <i>DomainParticipant</i> .	
get_builtin_transport_property	Gets the properties used to create a builtin transport plugin.	Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties() (Section 15.5 on the next page)
set_builtin_transport_property	Sets the properties used to create a builtin transport plugin.	
add_send_route	Adds a route for outgoing messages.	Adding a Send Route (Section 15.9.1 on page 769)
add_receive_route	Adds a route for incoming messages.	Adding a Receive Route (Section 15.9.2 on page 770)
lookup_transport	Looks up a transport plugin within a <i>DomainParticipant</i> .	Looking Up a Transport Plugin (Section 15.9.3 on page 771)

15.4 Explicitly Creating Builtin Transport Plugin Instances

The builtin transports (UDPv4, UDPv6, and Shared Memory) are implicitly created by default (if they are enabled via the [TRANSPORT_BUILTIN QosPolicy \(DDS Extension\)](#) (Section 8.5.7 on page 606)). Therefore, you only need to explicitly create a new instance if you want an extra instance (suppose you want two UDPv4 transports, one with special settings).

Transport plugins may be explicitly registered by first creating an instance of the transport plugin and then calling `register_transport()` ([Installing Additional Builtin Transport Plugins with `register_transport\(\)`](#) (Section 15.7 on page 765)). (For example, suppose you want an extra instance of a transport.) (Not available for the Java API.)

To create an instance of a builtin transport plugin, use one of the following functions:

```
NDDS_Transport_Plugin* NDDS_Transport_UDPv4_new (
    const struct NDDS_Transport_UDPv4_Property_t * property_in)
NDDS_Transport_Plugin* NDDS_Transport_UDPv6_new (
    const struct NDDS_Transport_UDPv6_Property_t * property_in)
NDDS_Transport_Plugin* NDDS_Transport_Shmem_new (
    const struct NDDS_Transport_Shmem_Property_t * property_in)
```

Where:

property_in Desired behavior of this transport. May be NULL for default properties.

For details on using these functions, please see the API Reference HTML documentation.

Your application may create and register multiple instances of these transport plugins with Connex DDS. This may be done to partition the network interfaces across multiple DDS domains. However, note that the underlying transport, the operating system's IP layer, is still a "singleton." For example, if a unicast transport has already bound to a port, and another unicast transport tries to bind to the same port, the second attempt will fail.

15.5 Setting Builtin Transport Properties of Default Transport Instance—`get/set_builtin_transport_properties()`

Perhaps you want to use one of the builtin transports, but need to modify the properties. (For default values, please see the API Reference HTML documentation.) Used together, the two operations below allow you to customize properties of the builtin transport when it is implicitly registered (see [Builtin Transport Plugins](#) (Section 15.1 on page 743)).

Note: Another way to change the properties is with the Property QosPolicy, see [Setting Builtin Transport Properties with the PropertyQosPolicy](#) (Section 15.6 on page 748). Changing properties with the Property QosPolicy will overwrite the properties set by calling `set_builtin_transport_property()`.

```
DDS_ReturnCode_t
NDDSTransportSupport::get_builtin_transport_property (
```

```

    DDSDomainParticipant * participant_in,
    DDS_TransportBuiltinKind builtin_transport_kind_in,
    struct NDDS_Transport_Property_t
        &builtin_transport_property_inout)
DDS_ReturnCode_t
NDDSTransportSupport::set_builtin_transport_property(
    DDSDomainParticipant * participant_in,
    DDS_TransportBuiltinKind builtin_transport_kind_in,
    const struct NDDS_Transport_Property_t
        &builtin_transport_property_in)

```

Where:

- participant_in** A valid non-NULL *DomainParticipant* that has not been enabled. If the *DomainParticipant* is already enabled when this operation is called, your transport property changes will not be reflected in the transport used by the *DomainParticipant's DataWriters* and *DataReaders*.
- builtin_transport_kind_in** The builtin transport kind for which to specify the properties.
- builtin_transport_property_inout** (Used by the “get” operation only.) The storage area where the retrieved property will be output. The specific type required by the **builtin_transport_kind_in** must be used.
- builtin_transport_property_in** (Used by the “set” operation only.) The new transport property that will be used to create the builtin transport plugin. The specific type required by the **builtin_transport_kind_in** must be used.

In this example, we want to use the builtin UDPv4 transport, but with modified properties.

```

/* Before this point, create a disabled DomainParticipant */
struct NDDS_Transport_UDPv4_Property_t property =
    NDDS_TRANSPORT_UDPv4_PROPERTY_DEFAULT;
if (NDDSTransportSupport::get_builtin_transport_property(
    participant, DDS_TRANSPORTBUILTIN_UDPv4,
    (struct NDDS_Transport_Property_t&)property) !=
    DDS_RETCODE_OK) {
    printf("***Error: get builtin transport property\n");
}
/* Make your desired changes here */
/* For example, to increase the UDPv4 max msg size to 64K: */
property.parent.message_size_max = 65535;
property.recv_socket_buffer_size = 65535;
property.send_socket_buffer_size = 65535;
if (NDDSTransportSupport::set_builtin_transport_property(
    participant, DDS_TRANSPORTBUILTIN_UDPv4,
    (struct NDDS_Transport_Property_t&)property)
    != DDS_RETCODE_OK) {
    printf("***Error: set builtin transport property\n");
}
/* Enable the participant to turn on communications with
   other participants in the DDS domain using the new

```



```

properties for the automatically registered builtin
transport plugins */
if (entity->enable() != DDS_RETCODE_OK) {
    printf("***Error: failed to enable entity\n");
}

```

Note: Builtin transport property changes will have no effect after the builtin transport has been registered. The builtin transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter/DataReader* is created, or (c) you lookup a builtin *DataReader*, whichever happens first.

15.6 Setting Builtin Transport Properties with the PropertyQosPolicy

The [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#) allows you to set name/value pairs of data and attach them to an entity, such as a *DomainParticipant*.

To assign properties, use the `add_property()` operation:

```

DDS_ReturnCode_t DDSPropertyQosPolicyHelper::add_property
    (DDS_PropertyQosPolicy policy,
     const char * name,
     const char * value,
     DDS_Boolean propagate)

```

For more information on `add_property()` and the other operations in the `DDSPropertyQosPolicyHelper` class, please see [Table 6.57 PropertyQoSPolicyHelper Operations](#), as well as the API Reference HTML documentation.

The ‘name’ part of the name/value pairs is a predefined string. The property names for the builtin transports are described in these tables:

- [Table 15.2 Properties for the Builtin UDPv4 Transport](#)
- [Table 15.3 Properties for Builtin UDPv6 Transport](#)
- [Table 15.4 Properties for Builtin Shared-Memory Transport](#)

See also:

- [Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6 \(Section 15.6.1 on page 763\)](#)
- [Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6 \(Section 15.6.1 on page 763\)](#)
- [Formatting Rules for IPv6 ‘Allow’ and ‘Deny’ Address Lists \(Section 15.6.2 on page 765\)](#)

Note:

Changing properties with the [PROPERTY QosPolicy \(DDS Extension\)](#) (Section 6.5.17 on page 394) will overwrite any properties set by calling `set_builtin_transport_property()`.

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
parent.address_bit_count	<p>Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128.</p> <p>For example, for an address range of 0-255, the address_bit_count should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.</p>
parent.properties_bitmap	<p>A bitmap that defines various properties of the transport to the Connex DDS core.</p> <p>Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connex DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.</p>
parent.gather_send_buffer_count_max	<p>Specifies the maximum number of buffers that Connex DDS can pass to the send() method of a transport plugin.</p> <p>The transport plugin send() API supports a gather-send concept, where the send() call can take several discontinuous buffers, assemble and send them in a single message. This enables Connex DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.</p> <p>However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connex DDS from trying to gather too many buffers into a send call for the transport plugin.</p> <p>Connex DDS requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is <code>NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN</code>.</p> <p>See Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6 (Section 15.6.1 on page 763).</p>
parent.message_size_max	<p>The maximum size of a message in bytes that can be sent or received by the transport plugin.</p> <p>This value must be set before the transport plugin is registered, so that Connex DDS can properly use the plugin.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
parent.allow_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>For example, the following are acceptable strings:</p> <p>192.168.1.1 192.168.1.* 192.168.* 192.* ether0</p> <p>If the list is non-empty, this "white" list is applied before the parent.deny_interfaces_list (Section below) list. The <i>DomainParticipant</i> will use the resulting list of interfaces to inform its remote participant(s) about which unicast addresses may be used to contact the <i>DomainParticipant</i>.</p> <p>The resulting list restricts <i>reception</i> to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform).</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>
parent.deny_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of these interfaces.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>For example, the following are acceptable strings:</p> <p>192.168.1.1 192.168.1.* 192.168.* 192.* ether0</p> <p>This "black" list is applied after the parent.allow_interfaces_list (Section above) list and filters out the interfaces that should <i>not</i> be used for receiving data.</p> <p>The resulting list restricts <i>reception</i> to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform).</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
parent. allow_multicast_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, allow the use of multicast only on these interfaces. If the list is empty, allow the use of all the allowed interfaces.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>This list sub-selects from the allowed interfaces that are obtained after applying the parent.allow_interfaces_list (Section on the previous page) "white" list and the parent.deny_interfaces_list (Section on the previous page) "black" list. From that resulting list, parent.deny_multicast_interfaces_list (Section below) is applied. Multicast output will be sent and may be received over the interfaces in the resulting list (if multicast is supported on the platform).</p> <p>If this list is empty, all the allowed interfaces may potentially be used for multicast.</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>
parent. deny_multicast_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of those interfaces for multicast.</p> <p>Interfaces should be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>This "black" list is applied after the parent.allow_multicast_interfaces_list (Section above) list and filters out the interfaces that should <i>not</i> be used for multicast. The final resulting list will be those interfaces that—if multicast is available—will be used for multicast sends.</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>
send_socket_buffer_size	<p>Size in bytes of the send buffer of a socket used for sending. On most operating systems, setsockopt() will be called to set the <code>SEND_BUF</code> to the value of this parameter.</p> <p>This value must be greater than or equal to the property, parent.message_size_max (Section on page 749).</p> <p>The maximum value is operating system-dependent.</p> <p>If <code>NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT</code>, then setsockopt() (or equivalent) will not be called to size the send buffer of the socket.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
recv_socket_buffer_size	<p>Size in bytes of the receive buffer of a socket used for receiving.</p> <p>On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter.</p> <p>This value must be greater than or equal to the property, parent.message_size_max (Section on page 749). The maximum value is operating system-dependent.</p> <p>Default: <code>NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT</code>.</p> <p>If <code>NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT</code>, then setsockopt() (or equivalent) will not be called to size the receive buffer of the socket.</p>
unicast_enabled	<p>Allows the transport plugin to use unicast UDP for sending and receiving. By default, it will be turned on. Also by default, it will use all the allowed network interfaces that it finds up and running when the plugin is instantiated.</p> <p>Can be 1 (enabled) or 0 (disabled).</p>
multicast_enabled	<p>Allows the transport plugin to use multicast for sending and receiving. You can turn multicast on or off for this plugin. The default is that multicast is on and the plugin will use the all network interfaces allowed for multicast that it finds up and running when the plugin is instantiated.</p> <p>Can be 1 (enabled) or 0 (disabled).</p>
multicast_ttl	<p>Value for the time-to-live parameter for all multicast sends using this plugin. This is used to set the TTL of multicast packets sent by this transport plugin.</p>
multicast_loopback_disabled	<p>Prevents the transport plugin from putting multicast packets onto the loopback interface.</p> <p>If disabled, then when sending multicast packets, do not put a copy on the loopback interface. This will prevent other applications on the same node (including itself) from receiving those packets.</p> <p>This is set to 0 by default. So multicast loopback is enabled. Turning off multicast loopback (set to 1) may result in minor performance gains when using multicast.</p> <p>Note: Windows CE does not support multicast loopback. This field is ignored for Windows CE targets.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
ignore_loopback_interface	<p>Prevents the transport plugin from using the IP loopback interface. Three values are allowed:</p> <ul style="list-style-type: none"> • 0: Forces local traffic to be sent over loopback, even if a more efficient transport (such as shared memory) is installed (in which case traffic will be sent over both transports). • 1: Disables local traffic via this plugin. The IP loopback interface will not be used, even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient transport (such as shared memory) instead of the IP loopback. • -1: Automatic. Enables local traffic via this plugin. To avoid redundant traffic, Connex DDS will selectively ignore the loopback destinations that are also reachable through shared memory.
ignore_nonup_interfaces	<p>This property is only supported on Windows platforms with statically configured IP addresses. It allows/disallows the use of interfaces that are not reported as UP (by the operating system) in the UDPv4 transport. Two values are allowed:</p> <ul style="list-style-type: none"> • 0: Allow interfaces that are reported as DOWN. <p>Setting this value to 0 supports communication scenarios in which interfaces are enabled after the participant is created. Once the interfaces are enabled, discovery will not occur until the participant sends the next periodic announcement (controlled by the parameter participant_qos.discovery_config.participant_liveliness_assert_period).</p> <p>To reduce discovery time, you may want to decrease the value of participant_liveliness_assert_period.</p> <p>For the above scenario, there is one caveat: non-UP interfaces must have a static IP assigned.</p> <ul style="list-style-type: none"> • 1 (default): Do not allow interfaces that are reported as DOWN.
interface_poll_period	<p>If ignore_nonup_interfaces is 0, the UDPv4 transport creates a new thread to query the status of the interfaces. The interface_poll_period specifies the polling period in milliseconds for performing this query.</p> <p>This property's value is ignored if ignore_nonup_interfaces is 1.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
ignore_nonrunning_interfaces	<p>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.</p> <p>The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed:</p> <ul style="list-style-type: none"> • 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. • 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.
no_zero_copy	<p>Prevents the transport plugin from doing a zero copy.</p> <p>By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application.</p> <p>The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the hardware, device driver, or the OS to allow the zero-copy feature to work for your application, you may have no choice but to turn off zero-copy.</p> <p>By default this is set to 0, so Connex DDS will use the zero-copy API if offered by the OS.</p>
send_blocking	<p>Controls the blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS. Currently two values are defined:</p> <p>NDDS_TRANSPORT_UDPv4_BLOCKING_ALWAYS: Sockets are blocking (default socket options for operating system).</p> <p>NDDS_TRANSPORT_UDPv4_BLOCKING_NEVER: Sockets are modified to make them non-blocking. This is not a supported configuration and may cause significant performance problems.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
transport_priority_mask	<p>Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low (Section below) and transport_priority_mapping_high (Section below) to define the mapping from the <code>TRANSPORT_PRIORITY</code> QosPolicy (Section 6.5.22 on page 409) to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket.</p> <p>For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high.</p> <p>If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.</p>
transport_priority_mapping_low	<p>Sets the low and high values of the output range to IPv4 TOS.</p>
transport_priority_mapping_high	<p>These values are used in conjunction with transport_priority_mask (Section above) to define the mapping from the <code>TRANSPORT_PRIORITY</code> QosPolicy (Section 6.5.22 on page 409) to the IPv4 TOS field. Defines the low and high values of the output range for scaling.</p> <p>Note that IPv4 TOS is generally an 8-bit value.</p>
reuse_multicast_receive_resource	<p>Controls whether or not to reuse receive resources. Setting this to 0 (FALSE) prevents multicast crosstalk by uniquely configuring a port and creating a receive thread for each multicast group address.</p> <p>Affects Linux systems only; ignored for non-Linux systems.</p>
protocol_overhead_max	<p>Maximum size in bytes of protocol overhead, including headers.</p> <p>This value is the maximum size, in bytes, of protocol-related overhead. Normally, the overhead accounts for UDP and IP headers. The default value is set to accommodate the most common UDP/IP header size.</p> <p>Note that when parent.message_size_max (Section on page 749) plus this overhead is larger than the UDPv4 maximum message size (65535 bytes), the middleware will automatically reduce the effective <code>message_size_max</code> to 65535 minus this overhead.</p>

Table 15.2 Properties for the Builtin UDPv4 Transport

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
public_address	<p>Public IP address associated with the transport instantiation.</p> <p>Setting the public IP address is only necessary to support communication over WAN that involves Network Address Translation (NAT).</p> <p>Typically, the address is the public address of the IP NAT router that provides access to the WAN.</p> <p>By default, the <i>DomainParticipant</i> creating the transport will announce the IP addresses obtained from the NICs to other <i>DomainParticipants</i> in the system.</p> <p>When this property is set, the <i>DomainParticipant</i> will announce the IP address corresponding to the property value instead of the LAN IP addresses associated with the NICs.</p> <p>Notes:</p> <p>Setting this property is necessary, but is not a sufficient condition for sending and receiving data over the WAN. You must also configure the IP NAT router to allow UDP traffic and to map the public IP address specified by this property to the <i>DomainParticipant's</i> private LAN IP address. This is typically done with one of these mechanisms:</p> <p>Port Forwarding: You must map the private ports used to receive discovery and user data traffic to the corresponding public ports (see Table 8.20 DDS_RtpsWellKnownPorts_t). Public and private ports must be the same since the transport does not allow you to change the mapping.</p> <p>1:1 NAT: You must add a 1:1 NAT entry that maps the public IP address specified in this property to the private LAN IP address of the <i>DomainParticipant</i>.</p> <p>By setting this property, the <i>DomainParticipant</i> only announces its public IP address to other <i>DomainParticipants</i>. Therefore, communication with <i>DomainParticipants</i> within the LAN that are running on different nodes will not work unless the NAT router is configured to enable NAT reflection (hairpin NAT).</p> <p>There is another way to achieve simultaneous communication with <i>DomainParticipants</i> running in the LAN and WAN, that does not require hairpin NAT. This way uses a gateway application such as <i>RTI Routing Service</i> to provide access to the WAN.</p>

Table 15.3 Properties for Builtin UDPv6 Transport

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
parent.address_bit_count	<p>Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128.</p> <p>For example, for an address range of 0-255, this address_bit_count should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.</p>
parent.properties_bitmap	<p>A bitmap that defines various properties of the transport to the Connex DDS core.</p> <p>Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connex DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.</p>
parent.gather_send_buffer_count_max	<p>Specifies the maximum number of buffers that Connex DDS can pass to the send() method of a transport plugin.</p> <p>The transport plugin send() API supports a gather-send concept, where the send() call can take several discontinuous buffers, assemble and send them in a single message. This enables Connex DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.</p> <p>However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connex DDS from trying to gather too many buffers into a send call for the transport plugin.</p> <p>Connex DDS requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.</p>
parent.message_size_max	<p>The maximum size of a message in bytes that can be sent or received by the transport plugin.</p> <p>This value must be set before the transport plugin is registered, so that Connex DDS can properly use the plugin.</p>
parent.allow_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See Formatting Rules for IPv6 'Allow' and 'Deny' Address Lists (Section 15.6.2 on page 765).</p> <p>If the list is non-empty, this "white" list is applied before the parent.deny_interfaces_list (Section on the next page) list. The <i>DomainParticipant</i> will use the resulting list of interfaces to inform its remote participant(s) about which unicast addresses may be used to contact the <i>DomainParticipant</i>.</p> <p>The resulting list restricts <i>reception</i> to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform).</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>

Table 15.3 Properties for Builtin UDPv6 Transport

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
parent.deny_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of these interfaces.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See Formatting Rules for IPv6 'Allow' and 'Deny' Address Lists (Section 15.6.2 on page 765).</p> <p>This "black" list is applied after the parent.allow_interfaces_list (Section on the previous page) list and filters out the interfaces that should <i>not</i> be used.</p> <p>The resulting list restricts <i>reception</i> to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform).</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>
parent.allow_multicast_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, allow the use of multicast only these interfaces; otherwise allow the use of all the allowed interfaces.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See Formatting Rules for IPv6 'Allow' and 'Deny' Address Lists (Section 15.6.2 on page 765).</p> <p>This list sub-selects from the allowed interfaces that are obtained after applying the parent.allow_interfaces_list (Section on the previous page) "white" list <i>and</i> the parent.deny_interfaces_list (Section above) "black" list. Finally, the parent.deny_multicast_interfaces_list (Section below) is applied. Multicast output will be sent and may be received over the interfaces in the resulting list (if multicast is supported on the platform).</p> <p>If this list is empty, all the allowed interfaces may potentially be used for multicast.</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>
parent.deny_multicast_interfaces_list	<p>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of those interfaces for multicast.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See Formatting Rules for IPv6 'Allow' and 'Deny' Address Lists (Section 15.6.2 on page 765).</p> <p>This "black" list is applied after the parent.allow_multicast_interfaces_list (Section above) list and filters out the interfaces that should <i>not</i> be used for multicast. Multicast output will be sent and may be received over the interfaces in the resulting list (if multicast is supported on the platform).</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is deleted.</p>

Table 15.3 Properties for Builtin UDPv6 Transport

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
send_socket_buffer_size	<p>Size in bytes of the send buffer of a socket used for sending.</p> <p>On most operating systems, setsockopt() will be called to set the SENDBUF to the value of this parameter.</p> <p>This value must be greater than or equal to parent.message_size_max. The maximum value is operating system-dependent.</p> <p>If <code>NDDS_TRANSPORT_UDPv6_SOCKET_BUFFER_SIZE_OS_DEFAULT</code>, then setsockopt() (or equivalent) will not be called to size the send buffer of the socket.</p>
recv_socket_buffer_size	<p>Size in bytes of the receive buffer of a socket used for receiving.</p> <p>On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter.</p> <p>This value must be greater than or equal to parent.message_size_max. The maximum value is operating system-dependent.</p> <p>If <code>NDDS_TRANSPORT_UDPv6_SOCKET_BUFFER_SIZE_OS_DEFAULT</code>, then setsockopt() (or equivalent) will not be called to size the receive buffer of the socket.</p>
unicast_enabled	<p>Allows the transport plugin to use unicast UDP for sending and receiving. By default, it will be turned on (1). Also by default, it will use all the allowed network interfaces that it finds up and running when the plugin is instanced.</p> <p>Can be 1 (enabled) or 0 (disabled).</p>
multicast_enabled	<p>Allows the transport plugin to use multicast for sending and receiving.</p> <p>You can turn multicast UDP on or off for this plugin. By default, it will be turned on (1). Also by default, it will use the all network interfaces allowed for multicast that it finds up and running when the plugin is instanced.</p> <p>Can be 1 (enabled) or 0 (disabled).</p>
multicast_ttl	<p>Value for the time-to-live parameter for all multicast sends using this plugin.</p> <p>This is used to set the TTL of multicast packets sent by this transport plugin</p>
multicast_loopback_disabled	<p>Prevents the transport plugin from putting multicast packets onto the loopback interface.</p> <p>If disabled, then when sending multicast packets, Connex DDS will not put a copy on the loopback interface. This will prevent applications on the same node (including itself) from receiving those packets.</p> <p>This is set to 0 by default, meaning multicast loopback is enabled. Disabling multicast loopback off (setting this value to 1) may result in minor performance gains when using multicast.</p>

Table 15.3 Properties for Builtin UDPv6 Transport

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
ignore_loopback_interface	<p>Prevents the transport plugin from using the IP loopback interface. Three values are allowed:</p> <ul style="list-style-type: none"> • 0: Enable local traffic via this plugin. This plugin will only use and report the IP loopback interface if there are no other network interfaces (NICs) up on the system. • 1: Disable local traffic via this plugin. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plugin like Shared Memory instead of the IP loopback. • -1: Automatic. Enables local traffic via this plugin. To avoid redundant traffic, Connex DDS will selectively ignore the loopback destinations that are also reachable through shared memory.
ignore_nonrunning_interfaces	<p>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.</p> <p>The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed:</p> <ul style="list-style-type: none"> • 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. • 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.
no_zero_copy	<p>Prevents the transport plugin from doing a zero copy.</p> <p>By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application.</p> <p>The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the H/W, device driver, or the OS to allow the zero-copy feature to work for your application, you may have no choice but to turn off zero-copy.</p> <p>By default this is set to 0, so Connex DDS will use the zero-copy API if offered by the OS.</p>

Table 15.3 Properties for Builtin UDPv6 Transport

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
send_blocking	<p>Controls the blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS. Currently two values are defined:</p> <ul style="list-style-type: none"> • <code>NDDS_TRANSPORT_UDPv4_BLOCKING_ALWAYS</code>: Sockets are blocking (default socket options for Operating System). • <code>NDDS_TRANSPORT_UDPv4_BLOCKING_NEVER</code>: Sockets are modified to make them non-blocking. This is not a supported configuration and may cause significant performance problems.
enable_v4mapped	<p>Specifies whether the UDPv6 transport will process IPv4 addresses.</p> <p>Set this to 1 to turn on processing of IPv4 addresses. Note that this may make it incompatible with use of the UDPv4 transport within the same <i>DomainParticipant</i>.</p>
transport_priority_mask	<p>Sets a mask for use of transport priority field.</p> <p>If transport priority mapping is supported on the platform¹, this mask is used in conjunction with transport_priority_mapping_low (Section below) and transport_priority_mapping_high (Section below) to define the mapping from the DDS transport priority <code>TRANSPORT_PRIORITY</code> QosPolicy (Section 6.5.22 on page 409) to the IPv6 TCLASS field.</p> <p>Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv6 TCLASS field on an outgoing socket.</p> <p>For example, the value <code>0x0000ff00</code> causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (<code>0x0000 - 0xff00</code> in this case) to the range specified by low and high.</p> <p>If the mask is set to zero, then the transport will not set IPv6 TCLASS for send sockets.</p>
transport_priority_mapping_low	<p>Sets the low and high values of the output range to IPv6 TCLASS.</p>
transport_priority_mapping_high	<p>These values are used in conjunction with transport_priority_mask (Section above) to define the mapping from DDS transport priority to the IPv6 TCLASS field. Defines the low and high values of the output range for scaling.</p> <p>Note that IPv6 TCLASS is generally an 8-bit value.</p>

¹See the *Platform Notes* to find out if the transport priority is supported on a specific platform.

Table 15.4 Properties for Builtin Shared-Memory Transport

Property Name (prefix with 'dds.transport.shmem.builtin.')	Property Value Description
parent.address_bit_count	<p>Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128.</p> <p>For example, for an address range of 0-255, this address_bit_count should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.</p>
parent.properties_bitmap	<p>A bitmap that defines various properties of the transport to the Connex DDS core.</p> <p>Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connex DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.</p>
parent.gather_send_buffer_count_max	<p>Specifies the maximum number of buffers that Connex DDS can pass to the send() method of a transport plugin.</p> <p>The transport plugin send() API supports a gather-send concept, where the send() call can take several discontinuous buffers, assemble and send them in a single message. This enables Connex DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.</p> <p>However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connex DDS from trying to gather too many buffers into a send call for the transport plugin.</p> <p>Connex DDS requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum is <code>NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN</code>.</p>
parent.message_size_max	<p>The maximum size of a message in bytes that can be sent or received by the transport plugin.</p> <p>This value must be set before the transport plugin is registered, so that Connex DDS can properly use the plugin.</p>
parent.allow_interfaces_list	<p>Not applicable to the Shared-Memory Transport</p>
parent.deny_interfaces_list	
parent.allow_multicast_interfaces_list	
parent.deny_multicast_interfaces_list	

Table 15.4 Properties for Builtin Shared-Memory Transport

Property Name (prefix with 'dds.transport.shmem.builtin.')	Property Value Description
received_message_count_max	<p>Number of messages that can be buffered in the receive queue. This is the maximum number of messages that can be buffered in a RecvResource of the Transport Plugin. This does not guarantee that the Transport-Plugin will actually be able to buffer received_message_count_max messages of the maximum size set in parent.message_size_max (Section on the previous page).</p> <p>The total number of bytes that can be buffered for a RecvResource is actually controlled by receive_buffer_size (Section below).</p>
receive_buffer_size	<p>The total number of bytes that can be buffered in the receive queue.</p> <p>This number controls how much memory is allocated by the plugin for the receive queue (on a per RecvResource basis). The actual number of bytes allocated is:</p> $\text{size} = \text{receive_buffer_size} + \text{message_size_max} + \text{received_message_count_max} * \text{fixedOverhead}$ <p>where <i>fixedOverhead</i> is some small number of bytes used by the queue data structure.</p> <p>If receive_buffer_size < (message_size_max * received_message_count_max), the transport plugin will not be able to store received_message_count_max messages of size message_size_max.</p> <p>If receive_buffer_size > (message_size_max * received_message_count_max), then there will be memory allocated that cannot be used by the plugin and thus wasted.</p> <p>To optimize memory usage, specify a receive queue size less than that required to hold the maximum number of messages which are all of the maximum size.</p> <p>In most situations, the average message size may be far less than the maximum message size. So for example, if the maximum message size is 64K bytes, and you configure the plugin to buffer at least 10 messages, then 640K bytes of memory would be needed if all messages were 64K bytes. Should this be desired, then receive_buffer_size should be set to 640K bytes.</p> <p>However, if the average message size is only 10K bytes, then you could set the receive_buffer_size to 100K bytes. This allows you to optimize the memory usage of the plugin for the average case and yet allow the plugin to handle the extreme case.</p> <p>The queue will always be able to hold 1 message of message_size_max bytes, regardless of the value of receive_buffer_size.</p>

15.6.1 Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6

To minimize memory copies, Connex DDS uses the "gather send" API that may be available on the transport.

Some operating systems limit the number of gather buffers that can be given to the gather-send function. This limits Connex DDS's ability to concatenate multiple DDS samples into a single network message.

An example is the UDP transport's **sendmsg()** call, which on some OSs (such as Solaris) can only take 16 gather buffers, limiting the number of DDS samples that can be concatenated to five or six.

To match this limitation, Connex DDS sets the UDPv4 and UDPv6 transport plug-ins' **gather_send_buffer_count_max** to 16 by default for all operating systems. This field is part of the **NDDS_Transport_Property_t** structure.

- On VxWorks 5.5 operating systems, **gather_send_buffer_count_max** can be set as high as 63.
- On Windows and INTEGRITY operating systems, **gather_send_buffer_count_max** can be set as high as 128.
- On most other operating systems, **gather_send_buffer_count_max** can be set as high as 16.

If you are using an OS that allows more than 16 gather buffers for a **sendmsg()** call, you may increase the UDPv4 or UDPv6 transport plug-in's **gather_send_buffer_count_max** from the default up to your OS's limit (but no higher than 128).

For example, if your OS imposes a limit of 64 gather buffers, you may increase the **gather_send_buffer_count_max** up to 64. However, if your OS's gather-buffer limit is 1024, you may only increase the **gather_send_buffer_count_max** up to 128.

By changing **gather_send_buffer_count_max**, you can increase performance in the following situations:

- When a *DataWriter* is sending multiple packets to a *DataReader* either because the *DataReader* is a late-joiner and needs to catch up, or because several packets were dropped and need to be resent. Changing the setting will help when the *DataWriter* needs to send or resend more than five or six packets at a time.
- If your application has more than five or six *DataWriters* or *DataReaders* in a participant. (In this case, the change will make the discovery process more efficient.)
- When using an asynchronous *DataWriter*, DDS samples are sent asynchronously by a separate thread. DDS samples may not be sent immediately, but may be queued instead, depending on the settings of the associated FlowController. If multiple DDS samples in the queue must be sent to the same destination, they will be coalesced into as few network packets as possible. The number of DDS samples that can be put in a single message is directly proportional to **gather_send_buffer_count_max**. Therefore, by maximizing **gather_send_buffer_count_max**, you can minimize the number of packets on the wire.

The **register_transport()** operation registers a transport plugin for use with a *DomainParticipant* and assigns it a network address. (Note: this operation is only available in the APIs other than Java or .NET. If you are using Java or .NET, use the Property QosPolicy to install additional transport plugins.)

```
NDDS_Transport_Handle_t NDDSTransportSupport::register_transport(
    DDSDomainParticipant * participant_in,
    NDDS_Transport_Plugin * transport_in,
    const DDS_StringSeq & aliases_in,
    const NDDS_Transport_Address_t & network_address_in)
```

Where:

participant_in A non-NULL, disabled DomainParticipant.

transport_in A non-NULL transport plugin that is currently not registered with another DomainParticipant.

aliases_in A non-NULL sequence of strings used as aliases to refer to the transport plugin symbolically. The transport plugin will be "available for use" to an Entity contained in the DomainParticipant, if the transport alias list associated with the Entity contains one of these transport aliases. An empty alias list represents a WILDCARD and matches ALL aliases. See [Transport Aliases \(Section 15.7.2 on the facing page\)](#).

network_address_in The network address at which to register this transport plugin. The least significant transport_in.property.address_bit_count will be truncated. The remaining bits are the network address of the transport plugin. See [Transport Network Addresses \(Section 15.7.3 on page 768\)](#).

Note: You must ensure that the transport plugin instance is only used by one *DomainParticipant* at a time. See [Transport Lifecycles \(Section 15.7.1 below\)](#).

Upon success, a valid non-NIL transport handle is returned, representing the association between the *DomainParticipant* and the transport plugin. If the transport cannot be registered, NDDS_TRANSPORT_HANDLE_NIL is returned.

Note that a transport plugin's class name is automatically registered as an implicit alias for the plugin. Thus, a class name can be used to refer to all the transport plugin instances of that class.

The C and C++ APIs also have a operation to retrieve a registered transport plugin, **get_transport_plugin()**.

```
NDDS_Transport_Plugin* get_transport_plugin(
    DDSDomainParticipant* participant_in,
    const char* alias_in);
```

15.7.1 Transport Lifecycles

If you create and register a transport plugin with a *DomainParticipant*, you are responsible for deleting it by calling its destructor. Builtin transport plugins are automatically managed by Connex DDS if they are implicitly registered through the TransportBuiltinQosPolicy.

User-created transport plugins must not be deleted while they are still in use by a *DomainParticipant*. This generally means that a user-created transport plugin instance can only be deleted after the *DomainParticipant* with which it was registered is deleted. Note that a transport plugin cannot be "unregistered" from a *DomainParticipant*.

A transport plugin instance cannot be registered with more than one *DomainParticipant* at a time. This requirement is necessary to guarantee the multi-threaded safety of the transport API.

Thus, if the same physical transport resources are to be used with multiple *DomainParticipants* in the same address space, the transport plugin should be written in such a way so that it can be instantiated multiple times—once for each *DomainParticipant* in the address space. Note that it is always possible to write the transport plugin so that multiple transport plugin instances share the same underlying resources; however the burden (if any) of guaranteeing multi-threaded safety to access shared resource shifts to the transport plugin developer.

15.7.2 Transport Aliases

In order to use a transport plugin instance in a Connex DDS application, it must be registered with a *DomainParticipant* using the **register_transport()** operation ([Installing Additional Builtin Transport Plugins with register_transport\(\)](#) (Section 15.7 on page 765)). **register_transport()** takes a pointer to the transport plugin instance, and in addition allows you to specify a sequence of "alias" strings to symbolically refer to the transport plugin. The same alias strings can be used to register more than one transport plugin.

Multiple transport plugins can be registered with a *DomainParticipant*. An alias symbolically refers to one or more transport plugins registered with the *DomainParticipant*. Pre-configured builtin transport plugin instances can be referred to using preconfigured aliases.

A transport plugin's class name is automatically used as an implicit alias. It can be used to refer to all the transport plugin instance of that class.

You can use aliases to refer to transport plugins in order to specify:

- Transport plugins to use for discovery (see **enabled_transports** in [DISCOVERY QosPolicy \(DDS Extension\)](#) (Section 8.5.2 on page 580)), and for *DataWriters* and *DataReaders* (see [TRANSPORT_SELECTION QosPolicy \(DDS Extension\)](#) (Section 6.5.23 on page 411)).
- Multicast addresses on which to receive discovery messages (see **multicast_receive_addresses** in [DISCOVERY QosPolicy \(DDS Extension\)](#) (Section 8.5.2 on page 580)), and the multicast addresses and ports on which to receive user data (`DDS_DataReaderQos::multicast`).
- Unicast ports used for user data (see [TRANSPORT_UNICAST QosPolicy \(DDS Extension\)](#) (Section 6.5.24 on page 412)) on both *DataWriters* and *DataReaders*.
- Transport plugins used to parse an address string in a locator.

A *DomainParticipant* (and its contained entities) will start using a transport plugin after the *DomainParticipant* is enabled (see [Enabling DDS Entities \(Section 4.1.2 on page 154\)](#)). An entity will use all the transport plugins that match the specified transport QoS policy. All transport plugins are treated uniformly, regardless of how they were created or registered; there is no notion of some transports being more "special" than others.

15.7.3 Transport Network Addresses

The address bits *not* used by the transport plugin for its internal addressing constitute its network address bits.

In order for Connex DDS to properly route the messages, each unicast interface in the DDS domain must have a unique address.

You specify the network address when installing a transport plugin via the **register_transport()** operation ([Installing Additional Builtin Transport Plugins with register_transport\(\) \(Section 15.7 on page 765\)](#)). Choose the network address for a transport plugin so that the resulting fully qualified 128-bit address will be unique in the DDS domain.

If two instances of a transport plugin are registered with a *DomainParticipant*, they need different network addresses so that their unicast interfaces will have unique, fully qualified 128-bit addresses.

While it is possible to create multiple transports with the same network address (this can be useful for certain situations), this requires special entity configuration for most transports to avoid clashes in resource use (e.g., sockets for UDPv4 transport).

15.8 Installing Additional Builtin Transport Plugins with PropertyQosPolicy

Similar to default builtin transport instances, additional builtin transport instances can also be configured through [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#).

To install additional instances of builtin transport, the Properties listed in [Table 15.6 Properties for Dynamically Loading and Registering Additional Builtin Transport Plugins](#) are required.

Table 15.6 Properties for Dynamically Loading and Registering Additional Builtin Transport Plugins

Property Name	Description
dds.transport.load_plugins	Comma-separated list of <TRANSPORT_PREFIX>. Up to 8 entries may be specified.

Table 15.6 Properties for Dynamically Loading and Registering Additional Builtin Transport Plugins

Property Name	Description
<TRANSPORT_PREFIX>	<p>Indicates the additional builtin transport instances to be installed, and must be in one of the following form, where <STRING> can be any string other than “builtin”:</p> <pre>dds.transport.shmem.<STRING></pre> <pre>dds.transport.UDPv4.<STRING></pre> <pre>dds.transport.UDPv6.<STRING></pre> <p>In the following examples in this table, <TRANSPORT_PREFIX> is used to indicate one element of this string that is used as a prefix in the property names for all the settings that are related to the plugin.</p>
<TRANSPORT_PREFIX>. aliases	<p>Optional.</p> <p>Aliases used to register the transport to the <i>DomainParticipant</i>. Refer to the aliases_in parameter in register_transport() (see Installing Additional Builtin Transport Plugins with register_transport() (Section 15.7 on page 765)). Aliases should be specified as a comma separated string, with each comma delimiting an alias. If it is not specified, <TRANSPORT_PREFIX> is used as the default alias for the plugin.</p>
<TRANSPORT_PREFIX>. network_address	<p>Optional.</p> <p>Network address used to register the transport to the <i>DomainParticipant</i>. Refer to network_address_in parameter in register_transport() (see Installing Additional Builtin Transport Plugins with register_transport() (Section 15.7 on page 765)). If it is not specified, the network_address_out output parameter from NDDS_Transport_create_plugin is used. The default value is a zeroed out network address.</p>
<TRANSPORT_PREFIX>. <property_name>	<p>Optional.</p> <p>Property for creating the transport plugin. More than one <TRANSPORT_PREFIX>.<property_name> can be specified. See Table 15.2 Properties for the Builtin UDPv4 Transport through Table 15.4 Properties for Builtin Shared-Memory Transport for the property names that can be used to configure the additional builtin transport instances. The only difference is that the property name will be prefixed by dds.transport.<builtin_transport_name>.<instance_name>, where <instance_name> is configured through the dds.transport.load_plugins property instead of dds.transport.<builtin_transport_name>.builtin.</p>

15.9 Other Transport Support Operations

15.9.1 Adding a Send Route

By default, a transport plugin will send outgoing messages using the network address range at which the plugin was registered.

The **add_send_route()** operation allows you to control the routing of outgoing messages, so that a transport plugin will only send messages to certain ranges of destination addresses.

Before using this operation, the *DomainParticipant* to which the transport is registered must be disabled.

```
DDS_ReturnCode_t NDDSTransportSupport::add_send_route(
    const NDDS_Transport_Handle_t & transport_handle_in,
```

```
const NDDS_Transport_Address_t & address_range_in,
DDS_Long address_range_bit_count_in)
```

Where:

transport_handle_in A valid non-NIL transport handle as a result of a call to **register_transport()** ([Installing Additional Builtin Transport Plugins with register_transport\(\)](#) (Section 15.7 on page 765)).

address_range_in The outgoing address range for which to use this transport plugin.

address_range_bit_count_in The number of most significant bits used to specify the address range.

It returns one of the standard return codes or DDS_RETCODE_PRECONDITION_NOT_MET.

The method can be called multiple times for a transport plugin, with different address ranges. You can set up a routing table to restrict the use of a transport plugin to send messages to selected addresses ranges.

Outgoing Address Range 1	->	Transport Plugin
...	->	...
Outgoing Address Range K	->	Transport Plugin

15.9.2 Adding a Receive Route

By default, a transport plugin will receive incoming messages using the network address range at which the plugin was registered.

The **add_receive_route()** operation allows you to configure a transport plugin so that it will only receive messages on certain ranges of addresses.

Before using this operation, the *DomainParticipant* to which the transport is registered must be disabled.

```
DDS_ReturnCode_t NDDSTransportSupport::add_receive_route(
    const NDDS_Transport_Handle_t & transport_handle_in,

    const NDDS_Transport_Address_t & address_range_in,

    DDS_Long address_range_bit_count_in)
```

Where:

transport_handle_in A valid non-NIL transport handle as a result of a call to **register_transport()** ([Installing Additional Builtin Transport Plugins with register_transport\(\)](#) (Section 15.7 on page 765)).

address_range_in The incoming address range for which to use this transport plugin.

address_range_bit_count_in The number of most significant bits used to specify the address range.

It returns one of the standard return codes or `DDS_RETCODE_PRECONDITION_NOT_MET`.

The method can be called multiple times for a transport plugin, with different address ranges.

Transport Plugin	<-	Incoming Address Range 1
...	<-	...
Transport Plugin	<-	Incoming Address Range M

You can set up a routing table to restrict the use of a transport plugin to receive messages from selected ranges. For example, you may restrict a transport plugin to:

Receive messages from a certain multicast address range.

Receive messages only on certain unicast interfaces (when multiple unicast interfaces are available on the transport plugin).

15.9.3 Looking Up a Transport Plugin

If you need to get the handle associated with a transport plugin that is registered with a *DomainParticipant*, use the **lookup_transport()** operation.

```
NDDS_Transport_Handle_t NDDSTransportSupport::lookup_transport(
    DDSDomainParticipant * participant_in,
    DDS_StringSeq & aliases_out,
    NDDS_Transport_Address_t & network_address_out,
    NDDS_Transport_Plugin * transport_in )
```

Where:

participant_in A non-NULL *DomainParticipant*.

aliases_out A sequence of strings where the aliases used to refer to the transport plugin symbolically will be returned. NULL if not interested.

network_address_out The network address at which to register the transport plugin will be returned here. NULL if not interested.

transport_in A non-NULL transport plugin that is already registered with the *DomainParticipant*.

If successful, this operation returns a valid non-NIL transport handle, representing the association between the *DomainParticipant* and the transport plugin; otherwise it returns a `NDDS_TRANSPORT_HANDLE_NIL` upon failure.

Chapter 16 Built-In Topics

This chapter discusses how to use Built-in Topics.

Connex DDS must discover and keep track of remote entities, such as new participants in the DDS domain. This information may also be important to the application itself, which may want to react to this discovery or access it on demand. To support these needs, Connex DDS provides *built-in Topics* (“DCPSParticipant”, “DCPSPublication”, “DCPSSubscription” in [Built-in Writers and Readers for Discovery \(Section Figure 14.2 on page 718\)](#)) and the corresponding built-in *DataReaders* that you can use to access this discovery information.

The discovery information is accessed just as if it is normal application data. This allows the application to know (either via listeners or by polling) when there are any changes in those values. Note that only entities that belong to a *different DomainParticipant* are being discovered and can be accessed through the built-in readers. Entities that are created within the local *DomainParticipant* are not included as part of the data that can be accessed by the built-in readers.

Built-in topics contain information about the remote entities, including their QoS policies. These QoS policies appear as normal fields inside the topic’s data, which can be read by means of the built-in Topic. Additional information is provided to identify the entity and facilitate the application logic.

16.1 Listeners for Built-in Entities

Built-in entities have default listener settings:

- The built-in *Subscriber* and its built-in topics have 'nil' listeners—all status bits are set in the listener masks, but the listener is NULL. This effectively creates a NO-OP listener that does not reset communication status.
- Built-in *DataReaders* have null listeners with no status bits set in their masks.

This approach prevents callbacks to the built-in *DataReader* listeners from invoking your *DomainParticipant*’s listeners, and at the same time ensures that the status changed flag is not

reset. For more information, see [Table 4.4 Effect of Different Combinations of Listeners and Status Bit Masks](#) and [Hierarchical Processing of Listeners \(Section 4.4.4 on page 180\)](#).

16.2 Built-in DataReaders

Built-in *DataReaders* belong to a built-in *Subscriber*, which can be retrieved by using the *DomainParticipant*'s `get_builtin_subscriber()` operation. You can retrieve the built-in *DataReaders* by using the *Subscriber*'s `lookup_datareader()` operation, which takes the Topic name as a parameter. The built-in *DataReader* is created when `lookup_datareader()` is called on a built-in topic for the first time.

To conserve memory, built-in *Subscribers* and *DataReaders* are created only if and when you look them up. Therefore, if you do not want to miss any built-in data, you should look up the built-in readers before the *DomainParticipant* is enabled.

The following tables describe the built-in topics and their data types. The [USER_DATA QosPolicy \(Section 6.5.26 on page 417\)](#), [TOPIC_DATA QosPolicy \(Section 5.2.1 on page 209\)](#) and [GROUP_DATA QosPolicy \(Section 6.4.4 on page 320\)](#) are included as part of the built-in data type and are not used by Connex DDS. Therefore, you can use them to send application-specific information.

Built-in topics can be used in conjunction with the `ignore_*` operations to ignore certain entities (see [Restricting Communication—Ignoring Entities \(Section 16.4 on page 784\)](#)).

Table 16.1 Participant Built-in Topic's Data Type (DDS_ParticipantBuiltinTopicData)

Type	Field	Description
DDS_BuiltinTopicKey	key	Key to distinguish the discovered <i>DomainParticipant</i>
DDS_UserDataQosPolicy	user_data	Data that can be set when the related <i>DomainParticipant</i> is created (via the USER_DATA QosPolicy (Section 6.5.26 on page 417)) and that the application may use as it wishes (e.g., to perform some security checking).
DDS_PropertyQosPolicy	property	Pairs of names/values to be stored with the <i>DomainParticipant</i> . See PROPERTY QosPolicy (DDS Extension) (Section 6.5.17 on page 394) . The usage is strictly application-dependent.
DDS_ProtocolVersion_t	rtps_protocol_version	Version number of the RTPS wire protocol used.
DDS_VendorId_t	rtps_vendor_id	ID of vendor implementing the RTPS wire protocol.
DDS_UnsignedLong	dds_builtin_endpoints	Bitmap set by the discovery plugins. Each bit in this field indicates a built-in endpoint present for discovery.

Table 16.1 Participant Built-in Topic's Data Type (DDS_ParticipantBuiltinTopicData)

Type	Field	Description
DDS_LocatorSeq	default_unicast_locators	If the TransportUnicastQosPolicy is not specified when a <i>DataWriter/DataReader</i> is created, the unicast_locators in the corresponding Publication/Subscription built-in topic data will be empty. When the unicast_locators in the Publication/SubscriptionBuiltinTopicData is empty, the default_unicast_locators in the corresponding Participant Builtin Topic Data is assumed. If default_unicast_locators is empty, it defaults to DomainParticipantQos.default_unicast.
DDS_ProductVersion_t	product_version	Vendor-specific parameter. The current version of Connex DDS.
DDS_EntityNameQosPolicy	participant_name	Name and role_name assigned to the <i>DomainParticipant</i> . See ENTITY_NAME QosPolicy (DDS Extension) (Section 6.5.9 on page 374).
DDS_DomainId_t	domain_id	Domain ID associated with the discovered participant.
DDS_TransportInfoSeq	transport_info	A sequence of DDS_TransportInfo_t containing information about each of the installed transports of the discovered <i>DomainParticipant</i> . A DDS_TransportInfo_t structure contains the class_id and message_size_max for a single transport. The maximum length of this sequence is controlled by the DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4 on page 593) transport_info_list_max_length (see Table 8.12 DDS_DomainParticipantResourceLimitsQosPolicy). Connex DDS uses the transport information propagated via discovery to detect potential misconfigurations in a Connex DDS distributed system. If two <i>DomainParticipants</i> that discover each other have one common transport with different values for message_size_max, Connex DDS prints a warning message about that condition.

Table 16.2 Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)

Type	Field	Description
DDS_BuiltinTopicKey_t	key	Key to distinguish the discovered <i>DataWriter</i>
DDS_BuiltinTopicKey_t	participant_key	Key to distinguish the participant to which the discovered <i>DataWriter</i> belongs
DDS_String	topic_name	Topic name of the discovered <i>DataWriter</i>
DDS_String	type_name	Type name attached to the topic of the discovered <i>DataWriter</i>

Table 16.2 Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)

Type	Field	Description
DDS_DurabilityQosPolicy	durability	QosPolicies of the discovered <i>DataWriter</i>
DDS_DurabilityService-QosPolicy	durability_service	
DDS_DeadlineQosPolicy	deadline	
DDS_DestinationOrder-QosPolicy	destination_order	
DDS_LatencyBudget-QosPolicy	latency_budget	
DDS_LivelinessQosPolicy	liveliness	
DDS_ReliabilityQosPolicy	reliability	
DDS_LifespanQosPolicy	lifespan	
DDS_UserDataQosPolicy	user_data	Data that can be set when the <i>DataWriter</i> is created (via the USER_DATA QosPolicy (Section 6.5.26 on page 417)) and that the application may use as it wishes.
DDS_OwnershipQosPolicy	ownership	QosPolicies of the discovered <i>DataWriter</i>
DDS_OwnershipStrength-QosPolicy	ownership_strength	
DDS_DestinationOrder-QosPolicy	destination_order	
DDS_PresentationQosPolicy	presentation	
DDS_PartitionQosPolicy	partition	Name of the partition, set in the PARTITION QosPolicy (Section 6.4.5 on page 323) for the publisher to which the discovered <i>DataWriter</i> belongs

Table 16.2 Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)

Type	Field	Description
DDS_ TopicDataQosPolicy	topic_data	Data that can be set when the <i>Topic</i> (with which the discovered <i>DataWriter</i> is associated) is created (via the TOPIC_DATA QosPolicy (Section 5.2.1 on page 209)) and that the application may use as it wishes.
DDS_ GroupDataQosPolicy	group_data	Data that can be set when the <i>Publisher</i> to which the discovered <i>DataWriter</i> belongs is created (via the GROUP_DATA QosPolicy (Section 6.4.4 on page 320)) and that the application may use as it wishes.
DDS_TypeObject *	type	Describes the type of the remote <i>DataReader</i> . See the API Reference HTML documentation.
DDS_TypeCode *	type_code	Type code information about this <i>Topic</i> . See Using Generated Types without Connex DDS (Standalone) (Section 3.7 on page 139) .
DDS_ BuiltinTopicKey_t	publisher_ key	The key of the <i>Publisher</i> to which the <i>DataWriter</i> belongs.
DDS_ PropertyQosPolicy	property	Properties (pairs of names/values) assigned to the corresponding <i>DataWriter</i> . Usage is strictly application-dependent. See PROPERTY QosPolicy (DDS Extension) (Section 6.5.17 on page 394) .
DDS_LocatorSeq	unicast_ locators	If the TransportUnicastQosPolicy is not specified when a <i>DataWriter/DataReader</i> is created, the unicast_locators in the corresponding Publication/Subscription built-in topic data will be empty. When the unicast_locators in the Publication/SubscriptionBuiltinTopicData is empty, the default_unicast_locators in the corresponding Participant Builtin Topic Data is assumed.
DDS_GUID_t	virtual_guid	Virtual GUID for the corresponding <i>DataWriter</i> . For more information, see Durability and Persistence Based on Virtual GUIDs (Section 12.2 on page 680) .
DDS_ ServiceQosPolicy	service	Service associated with the discovered <i>DataWriter</i> .
DDS_ ProtocolVersion_t	rtps_ protocol_ version	Version number of the RTPS wire protocol in use.
DDS_VendorId_t	rtps_ vendor_id	ID of the vendor implementing the RTPS wire protocol.
DDS_Product_ Version_t	product_ version	Vendor-specific value. For RTI, this is the current version of Connex DDS.

Table 16.2 Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)

Type	Field	Description
DDS_LocatorFilterQosPolicy	locator_filter	<p>When the MULTI_CHANNEL QoS Policy (DDS Extension) (Section 6.5.14 on page 386) is used on the discovered <i>DataWriter</i>, the locator_filter contains the sequence of LocatorFilters in that policy.</p> <p>There is one LocatorFilter per <i>DataWriter</i> channel. A channel is defined by a filter expression and a sequence of multicast locators.</p> <p>See LOCATOR_FILTER QoS Policy (DDS Extension) (Section 16.2.1 on page 782).</p>
DDS_Boolean	disable_positive_acks	Vendor specific parameter. Determines whether matching <i>DataReaders</i> send positive acknowledgements for reliability.
DDS_EntityNameQosPolicy	publication_name	Name and role_name assigned to the <i>DataWriter</i> . See ENTITY_NAME QoS Policy (DDS Extension) (Section 6.5.9 on page 374).

Table 16.3 Subscription Built-in Topic's Data Type (DDS_SubscriptionBuiltinTopicData)

Type	Field	Description
DDS_BuiltinTopicKey_t	key	Key to distinguish the discovered <i>DataReader</i> .
DDS_BuiltinTopicKey_t	participant_key	Key to distinguish the participant to which the discovered <i>DataReader</i> belongs.
char *	topic_name	Topic name of the discovered <i>DataReader</i> .
char *	type_name	Type name attached to the <i>Topic</i> of the discovered <i>DataReader</i> .
DDS_DurabilityQosPolicy	durability	QoS Policies of the discovered <i>DataReader</i>
DDS_DeadlineQosPolicy	deadline	
DDS_LatencyBudget-QoS Policy	latency_budget	
DDS_LivelinessQosPolicy	liveliness	
DDS_ReliabilityQosPolicy	reliability	
DDS_OwnershipQosPolicy	ownership	
DDS_DestinationOrderQosPolicy	destination_order	

Table 16.3 Subscription Built-in Topic's Data Type (DDS_SubscriptionBuiltinTopicData)

Type	Field	Description
DDS_UserDataQosPolicy	user_data	Data that can be set when the <i>DataReader</i> is created (via the USER_DATA QosPolicy (Section 6.5.26 on page 417)) and that the application may use as it wishes.
DDS_TimeBasedFilterQosPolicy	time_based_filter	QosPolicies of the discovered <i>DataReader</i>
DDS_PresentationQosPolicy	presentation	
DDS_PartitionQosPolicy	partition	Name of the partition, set in the PARTITION QosPolicy (Section 6.4.5 on page 323) for the <i>Subscriber</i> to which the discovered <i>DataReader</i> belongs.
DDS_TopicDataQosPolicy	topic_data	Data that can be set when the <i>Topic</i> to which the discovered <i>DataReader</i> belongs is created (via the TOPIC_DATA QosPolicy (Section 5.2.1 on page 209)) and that the application may use as it wishes.
DDS_GroupDataQosPolicy	group_data	Data that can be set when the <i>Publisher</i> to which the discovered <i>DataReader</i> belongs is created (via the GROUP_DATA QosPolicy (Section 6.4.4 on page 320)) and that the application may use as it wishes.
DDS_TypeObject *	type	Describes the type of the remote <i>DataReader</i> . See the API Reference HTML documentation.
DDS_TypeConsistencyEnforcementQosPolicy	type_consistency	Indicates the type-consistency requirements of the remote <i>DataReader</i> . See TYPE_CONSISTENCY_ENFORCEMENT QosPolicy (Section 7.6.6 on page 532) and the RTI Connex DDS Core Libraries Getting Started Guide Addendum for Extensible Types .
DDS_TypeCode *	type_code	Type code information about this <i>Topic</i> . See Using Generated Types without Connex DDS (Standalone) (Section 3.7 on page 139) .
DDS_BuiltinTopicKey_t	subscriber_key	Key of the <i>Subscriber</i> to which the <i>DataReader</i> belongs.
DDS_PropertyQosPolicy	property	Properties (pairs of names/values) assigned to the corresponding <i>DataReader</i> . Usage is strictly application-dependent. See PROPERTY QosPolicy (DDS Extension) (Section 6.5.17 on page 394) .
DDS_LocatorSeq	unicast_locators	If the <i>TransportUnicastQosPolicy</i> is not specified when a <i>DataWriter/DataReader</i> is created, the <i>unicast_locators</i> in the corresponding <i>Publication/Subscription builtin topic data</i> will be empty. When the <i>unicast_locators</i> in the <i>Publication/SubscriptionBuiltinTopicData</i> is empty, the default <i>unicast_locators</i> in the corresponding <i>Participant Builtin Topic Data</i> is assumed.
DDS_LocatorSeq	multicast_locators	Custom multicast locators that the endpoint can specify.

Table 16.3 Subscription Built-in Topic's Data Type (DDS_SubscriptionBuiltinTopicData)

Type	Field	Description
DDS_ContentFilter-Property_t	content_filter_property	Provides all the required information to enable content filtering on the writer side.
DDS_GUID_t	virtual_guid	Virtual GUID for the corresponding <i>DataReader</i> . For more information, see Durability and Persistence Based on Virtual GUIDs (Section 12.2 on page 680) .
DDS_ServiceQosPolicy	service	Service associated with the discovered <i>DataReader</i> .
DDS_ProtocolVersion_t	rtps_protocol_version	Version number of the RTPS wire protocol in use.
DDS_VendorId_t	rtps_vendor_id	ID of the vendor implementing the RTPS wire protocol.
DDS_Product_Version_t	product_version	Vendor-specific value. For RTI, this is the current version of Connex DDS.
DDS_Boolean	disable_positive_acks	Vendor specific parameter. Determines whether matching <i>DataReaders</i> send positive acknowledgements for reliability.
DDS_EntityNameQosPolicy	subscription_name	Name and role_name assigned to the <i>DataReader</i> . See ENTITY_NAME QosPolicy (DDS Extension) (Section 6.5.9 on page 374) .

Table 16.4 Topic Built-in Topic's Data Type (DDS_TopicBuiltinTopicData)

Type	Field	Description
DDS_BuiltinTopicKey_t	key	Key to distinguish the discovered <i>Topic</i>
DDS_String	name	<i>Topic</i> name
DDS_String	type_name	type name attached to the <i>Topic</i>

Table 16.4 Topic Built-in Topic's Data Type (DDS_TopicBuiltinTopicData)

Type	Field	Description
DDS_DurabilityQosPolicy	durability	QosPolicy of the discovered <i>Topic</i>
DDS_DurabilityServiceQosPolicy	durability_service	
DDS_DeadlineQosPolicy	deadline	
DDS_LatencyBudgetQosPolicy	latency_budget	
DDS_LivelinessQosPolicy	liveliness	
DDS_ReliabilityQosPolicy	reliability	
DDS_TransportPriorityQosPolicy	transport_priority	
DDS_LifespanQosPolicy	lifespan	
DDS_DestinationOrderQosPolicy	destination_order	
DDS_HistoryQosPolicy	history	
DDS_ResourceLimitsQosPolicy	resource_limits	
DDS_OwnershipQosPolicy	ownership	
DDS_TopicDataQosPolicy	topic_data	Data that can be set when the <i>Topic</i> to which the discovered <i>DataReader</i> belongs is created (via the TOPIC_DATA QosPolicy (Section 5.2.1 on page 209)) and that the application may use as it wishes.

[Table 16.5 QoS of Built-in Subscriber and DataReader](#) lists the QoS of the built-in *Subscriber* and *DataReader* created for accessing discovery data. These are provided for your reference only; they cannot be changed.

Table 16.5 QoS of Built-in Subscriber and DataReader

QosPolicy	Value
Deadline	period = infinite
DestinationOrder	kind = BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS

Table 16.5 QoS of Built-in Subscriber and DataReader

QosPolicy	Value
Durability	kind = TRANSIENT_LOCAL_DURABILITY_QOS
EntityFactory	autoenable_created_entities = TRUE
GroupData	value = empty sequence
History	kind = KEEP_LAST_HISTORY_QOS depth = 1
LatencyBudget	duration = 0
Liveliness	kind = AUTOMATIC_LIVELINESS_QOS lease_duration = infinite
Ownership	kind = SHARED_OWNERSHIP_QOS
Ownership Strength	value = 0
Presentation	access_scope = TOPIC_PRESENTATION_QOS coherent_access = FALSE ordered_access = FALSE
Partition	name = empty sequence
ReaderDataLifecycle	autopurge_nowriter_samples_delay = infinite
Reliability	kind = RELIABLE_RELIABILITY_QOS max_blocking_time is irrelevant for the <i>DataReader</i>
ResourceLimits	Depends on setting of DomainParticipantResourceLimitsQosPolicy and DiscoveryConfigQosPolicy in DomainParticipantQos: max_samples = domainParticipantQos.discovery_config. [participant/publication/subscription]_reader_resource_limits.max_samples max_instances = domainParticipantQos.resource_limits. [remote_writer/reader/participant]_allocation.max_count max_samples_per_instance = 1
TimeBasedFilter	minimum_separation = 0
TopicData	value = empty sequence
UserData	value = empty sequence

Note:

The DDS_TopicBuiltinTopicData built-in topic (described in [Table 16.4 Topic Built-in Topic's Data Type \(DDS_TopicBuiltinTopicData\)](#)) is meant to convey information about discovered *Topics*. However, this topic's data is not sent separately and therefore a *DataReader* for DDS_TopicBuiltinTopicData will not receive any data. Instead, DDS_TopicBuiltinTopicData data is included in the information carried by the built-in topics for Publications and Subscriptions (DDS_PublicationBuiltinTopicData and DDS_SubscriptionBuiltinTopicData) and can be accessed with their built-in *DataReaders*.

16.2.1 LOCATOR_FILTER QoS Policy (DDS Extension)

The LocatorFilter QoS Policy is only applicable to the built-in topic for a Publication (see [Table 16.2 Publication Built-in Topic's Data Type \(DDS_PublicationBuiltinTopicData\)](#)).

Table 16.6 DDS_LocatorFilterQosPolicy

Type	Field Name	Description
DDS_LocatorFilterSeq	locator_filters	A sequence of locator filters, described in Table 16.7 DDS_LocatorFilter_t . There is one locator filter per <i>DataWriter</i> channel. If the length of the sequence is zero, the <i>DataWriter</i> is not using multi-channel.
char *	filter_name	Name of the filter class used to describe the locator filter expressions. The following two values are supported: <ul style="list-style-type: none"> • DDS_SQLFILTER_NAME • DDS_STRINGMATCHFILTER_NAME

Table 16.7 DDS_LocatorFilter_t

Type	Field Name	Description
DDS_LocatorSeq	locators	A sequence of multicast address locators for the locator filter. See Table 16.8 DDS_Locator_t .
char *	filter_expression	A logical expression used to determine if the data will be published in the channel associated with this locator filter. See SQL Filter Expression Notation (Section 5.4.6 on page 222) and STRINGMATCH Filter Expression Notation (Section 5.4.7 on page 231) for information about the expression syntax.

Table 16.8 DDS_Locator_t

Type	Field Name	Description
DDS_Long	kind	<p>If the locator kind is DDS_LOCATOR_KIND_UDPv4, the address contains an IPv4 address. The leading 12 octets of the address must be zero. The last 4 octets store the IPv4 address.</p> <p>If the locator kind is DDS_LOCATOR_KIND_UDPv6, the address contains an IPv6 address. IPv6 addresses typically use a shorthand hexadecimal notation that maps one-to-one to the 16 octets of the address.</p> <p>In C#, the locator kinds for UDPv4 and UDPv6 addresses are Locator_t.LOCATOR_KIND_UDPv4 and Locator_t.LOCATOR_KIND_UDPv6.</p>
DDS_Octet [16]	address	The locator address.
DDS_UnsignedLong	port	The locator port number.

16.3 Accessing the Built-in Subscriber

Getting the built-in subscriber allows you to retrieve the built-in readers of the built-in topics through the *Subscriber's* lookup_datareader() operation. By accessing the built-in reader, you can access discovery information about remote entities.

```
// Lookup built-in reader

DDSDataReader *builtin_reader =

    builtin_subscriber->lookup_datareader(DDS_PUBLICATION_TOPIC_NAME);

if (builtin_reader == NULL) {

    // ... error

}

// Register listener to built-in reader

MyPublicationBuiltinTopicDataListener builtin_reader_listener =

    new MyPublicationBuiltinTopicDataListener();

if (builtin_reader->set_listener(builtin_reader_listener,

    DDS_DATA_AVAILABLE_STATUS) != DDS_RETCODE_OK) {

    // ... error
}
```

```

}

// enable DomainParticipant

if (participant->enable() != DDS_RETCODE_OK) {

    // ... error

}

```

For example, you can call the *DomainParticipant*'s `get_builtin_subscriber()` operation, which will provide you with a built-in Subscriber. Then you can use that built-in Subscriber to call the *Subscriber*'s `lookup_datareader()` operation; this will retrieve the built-in reader. Another option is to register a *Listener* on the built-in subscriber instead, or poll for the status of the built-in subscriber to see if any of the built-in data readers have received data.

16.4 Restricting Communication—Ignoring Entities

The `ignore_participant()` operation allows an application to ignore all communication from a specific *DomainParticipant*. Or for even finer control you can use the `ignore_publication()`, `ignore_subscription()`, and `ignore_topic()` operations. These operations are described below.

```

DDS_ReturnCode_t ignore_participant (const DDS_InstanceHandle_t &handle)

DDS_ReturnCode_t ignore_publication (const DDS_InstanceHandle_t &handle)

DDS_ReturnCode_t ignore_subscription (const DDS_InstanceHandle_t &handle)

DDS_ReturnCode_t ignore_topic (const DDS_InstanceHandle_t &handle)

```

The entity to ignore is identified by the *handle* argument. It may be a local or remote entity. For **`ignore_publication()`**, the handle will be that of a local *DataWriter* or a discovered remote *DataWriter*. For **`ignore_subscription()`**, that handle will be that of a local *DataReader* or a discovered remote *DataReader*.

The safest approach for ignoring an entity is to call the ignore operation within the *Listener* callback of the built-in reader, or before any local entities are enabled. This will guarantee that the local entities (entities that are created by the local *DomainParticipant*) will never have a chance to establish communication with the remote entities (entities that are created by another *DomainParticipant*) that are going to be ignored.

If the above is not possible and a remote entity is to be ignored after the communication channel has been established, the remote entity will still be removed from the database of the local application as if it never existed. However, since the remote application is not aware that the entity is being ignored, it may potentially be expecting to receive messages or continuing to send messages. Depending on the QoS of the remote entity, this may affect the behavior of the remote application and may potentially stop the remote application from communicating with other entities.

You can use this operation in conjunction with the `ParticipantBuiltinTopicData` to implement access control. You can pass application data associated with a *DomainParticipant* in the `USER_DATA QoSPolicy` (Section 6.5.26 on page 417). This application data is propagated as a field in the built-in topic. Your application can use the data to implement an access control policy.

Ignore operations, in conjunction with the Built-in Topic Data, can be used to implement access control. You can pass data associated with an entity in the `USER_DATA QoSPolicy` (Section 6.5.26 on page 417), `GROUP_DATA QoSPolicy` (Section 6.4.4 on page 320) or `TOPIC_DATA QoSPolicy` (Section 5.2.1 on page 209). This data is propagated as a field in the built-in topic. When data for a built-in topic is received, the application can check the `user_data`, `group_data` or `topic_data` field of the remote entity, determine if it meets the security requirement, and ignore the remote entity if necessary.

See also: [Discovery \(Section Chapter 14 on page 709\)](#).

16.4.1 Ignoring Specific Remote DomainParticipants

The `ignore_participant()` operation is used to instruct Connex DDS to locally ignore a remote *DomainParticipant*. It causes Connex DDS to locally behave as if the remote *DomainParticipant* does not exist.

```
DDS_ReturnCode_t ignore_participant (const DDS_InstanceHandle_t & handle)
```

After invoking this operation, Connex DDS will locally ignore any *Topic*, *publication*, or *subscription* that originates on that *DomainParticipant*. (If you only want to ignore specific publications or subscriptions, see [Ignoring Publications and Subscriptions \(Section 16.4.2 on the facing page\)](#) instead.) [Ignoring Participants \(Section Figure 16.1 below\)](#) provides an example.

By default, the maximum number of participants that can be ignored is limited by `ignored_entity_allocation.max_count` in the `DOMAIN_PARTICIPANT_RESOURCE_LIMITS QoSPolicy (DDS Extension)` (Section 8.5.4 on page 593). However, that behavior can be changed by using `ignore_entity_replacement_kind` in the same QoS policy.

See also: [Resource Limits Considerations for Ignored Entities \(Section 16.4.4 on page 788\)](#).

Caution: There is no way to reverse this operation. You can add to the peer list, however—see [Adding and Removing Peers List Entries \(Section 8.5.2.3 on page 581\)](#).

Figure 16.1 Ignoring Participants

```
class MyParticipantBuiltinTopicDataListener :
public DDSDataReaderListener {
    public:
        virtual void on_data_available(DDSDataReader *reader);
        // .....
};
void MyParticipantBuiltinTopicdataListener::on_data_available(
```

```

DDSDataReader *reader) {
    DDSParticipantBuiltinTopicDataDataReader
        *builtinTopicDataReader =
            DDSParticipantBuiltinTopicDataDataReader *) reader;
    DDS_ParticipantBuiltinTopicDataSeq data_seq;
    DDS_SampleInfoSeq info_seq;
    int = 0;
    if (builtinTopicDataReader->take(data_seq, info_seq,
        DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
        DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE) !=
        DDS_RETCODE_OK) {
        // ... error
    }
    for (i = 0; i < data_seq.length(); ++i) {
        if (info_seq[i].valid_data) {
            // check user_data for access control
            if (data_seq[i].user_data[0] != 0x9) {
                if (builtinTopicDataReader->get_subscriber()
                    ->get_participant()
                    ->ignore_participant(
                        info_seq[i].instance_handle)
                    != DDS_RETCODE_OK) {
                    // ... error
                }
            }
        }
    }
    if (builtinTopicDataReader->return_loan(
        data_seq, info_seq) != DDS_RETCODE_OK) {
        // ... error
    }
}

```

16.4.2 Ignoring Publications and Subscriptions

You can instruct Connex DDS to locally ignore a publication or subscription. A publication/subscription is defined by the association of a *Topic* name, user data and partition set on the *Publisher/Subscriber*. After this call, any data written related to associated *DataWriter/DataReader* will be ignored.

The entity to ignore is identified by the *handle* argument. For **ignore_publication()**, the handle will be that of a *DataWriter*. For **ignore_subscription()**, that handle will be that of a *DataReader*.

This operation can be used to ignore local *and* remote entities:

- For local entities, you can obtain the handle argument by calling the **get_instance_handle()** operation for that particular entity.
- For remote entities, you can obtain the handle argument from the `DDS_SampleInfo` structure retrieved when reading DDS data samples available for the entity's built-in *DataReader*.

```
DDS_ReturnCode_t ignore_publication (const DDS_InstanceHandle_t & handle)
DDS_ReturnCode_t ignore_subscription (const DDS_InstanceHandle_t & handle)
```

Caution: There is no way to reverse these operations.

Figure 16.2 Ignoring Publications below provides an example.

Figure 16.2 Ignoring Publications

```
class MyPublicationBuiltinTopicDataListener : public DDSDataReaderListener
{
public:
virtual void on_data_available(DDSDataReader *reader);
// .....
};
void MyPublicationBuiltinTopicdataListener::on_data_available(
DDSDataReader *reader) {
DDSPublicationBuiltinTopicDataReader *builtinTopicDataReader =
(DDS_PublicationBuiltinTopicDataReader *)reader;
DDS_PublicationBuiltinTopicDataSeq data_seq;
DDS_SampleInfoSeq info_seq;
int = 0;
if (builtinTopicDataReader->take(data_seq, info_seq,
DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE)
!= DDS_RETCODE_OK)
{
// ... error
}
for (i = 0; i < data_seq.length(); ++i) {
if (info_seq[i].valid_data) {
// check user_data for access control
if (data_seq[i].user_data[0] != 0x9) {
if (builtinTopicDataReader->get_subscriber()
->get_participant()
->ignore_publication(
info_seq[i].instance_handle)
!= DDS_RETCODE_OK) {
// ... error
}
}
}
}
}
if (builtinTopicDataReader->return_loan(data_seq, info_seq) !=
DDS_RETCODE_OK) {
...
}
```


16.4.3 Ignoring Topics

The `ignore_topic()` operation instructs Connex DDS to locally ignore a *Topic*. This means it will locally ignore any publication or subscription to the *Topic*.

```
DDS_ReturnCode_t ignore_topic (const DDS_InstanceHandle_t & handle)
```

Caution: There is no way to reverse this operation.

If you know that your application will never publish or subscribe to data under certain topics, you can use this operation to save local resources.

The *Topic* to ignore is identified by the handle argument. This handle is the one that appears in the `DDS_SampleInfo` retrieved when reading the DDS data samples from the built-in `DataReader` to the *Topic*.

16.4.4 Resource Limits Considerations for Ignored Entities

When an entity is ignored, Connex DDS adds it to an internal ‘ignore’ table whose resource limits are configured using the `ignored_entity_allocation.max_count` in the [DOMAIN_PARTICIPANT_RESOURCE_LIMITS QoS Policy \(DDS Extension\)](#) (Section 8.5.4). The behavior of Connex DDS when this limit is exceeded can be modified by using the `ignored_entity_replacement_kind` in the same QoS policy.

The default value for `ignored_entity_replacement_kind` is `DDS_NO_REPLACEMENT_IGNORED_ENTITY_REPLACEMENT`, meaning that a call to the `DomainParticipant’s ignore_participant()`, `ignore_publication()`, or `ignore_subscription()` will fail if the *DomainParticipant* has ignored more entities than the limit set in `ignored_entity_allocation.max_count` entities.

When `ignored_entity_replacement_kind` is set to `DDS_NOT_ALIVE_FIRST_IGNORED_ENTITY_REPLACEMENT`, a call to `ignore_participant()` will not fail when `ignored_entity_allocation.max_count` is exceeded, as long as there is one *DomainParticipant* already ignored. Instead, the call will replace one of the existing *DomainParticipants* in the internal table. The remote *DomainParticipant* that will be replaced is the one for which the local *DomainParticipant* had not received any message for the longest time.

When a remote *DomainParticipant* is replaced in the ‘ignore’ table, it becomes un-ignored. Thus, the local *DomainParticipant* would have to call `ignore_participant()` again to re-ignore the replaced entity.

Note: In this release, ignored publications and subscriptions are never replaced in the ‘ignore’ table. Since this table also contains the ignored *DomainParticipants*, a call to `ignore_participant()` will fail if `ignored_entity_allocation.max_count` is reached and none of the ignored entities is a *DomainParticipant*.

16.4.5 Supervising Endpoint Discovery

It is possible to control for which *DomainParticipants* endpoint discovery may occur. You can configure this behavior with the `enable_endpoint_discovery` field in the [DISCOVERY QoS Policy \(DDS](#)

Extension) (Section 8.5.2 on page 580):

- When set to TRUE (the default value), endpoint discovery will automatically occur for every discovered *DomainParticipant*. This is the normal operation of the discovery process.
- When set to FALSE, endpoint discovery will be disabled for every discovered *DomainParticipant*. Then applications will have to manually enable endpoint discovery (described below) for the *DomainParticipants* they are interested in communicating with. By disabling endpoint discovery, the *DomainParticipant* will not store any state about remote endpoints and will not send local endpoint information to remote *DomainParticipants*.

When **enable_endpoint_discovery** is set to FALSE, you have two options after a remote *DomainParticipant* is discovered:

- Call the *DomainParticipant*'s **resume_endpoint_discovery()** operation to enable endpoint discovery. After invoking this operation, the *DomainParticipant* will start to exchange endpoint information so that matching and communication can occur with the remote *DomainParticipant*.

```
DDS_ReturnCode_t resume_endpoint_discovery(
    const DDS_InstanceHandle_t & remote_participant_handle)
```

Or

- Call the *DomainParticipant*'s **ignore_participant()** operation to permanently ignore endpoint discovery with the remote *DomainParticipant*.

Setting **enable_endpoint_discovery** to FALSE enables application-level authentication use cases, in which a *DomainParticipant* will resume endpoint discovery with a remote *DomainParticipant* after successful authentication at the application level. The following example shows how to provide access control using this feature:

```
class MyParticipantBuiltinTopicDataListener :
    public DDSDataReaderListener {
public:
    virtual void on_data_available(DDSDataReader *reader);
    // ...
};
void MyParticipantBuiltinTopicDataListener::on_data_available(
    DDSDataReader *reader) {
    DDSParticipantBuiltinTopicDataDataReader
    *builtinTopicDataReader =
        DDSParticipantBuiltinTopicDataDataReader *) reader;
    DDS_ParticipantBuiltinTopicDataSeq data_seq;
    DDS_SampleInfoSeq info_seq;
    int = 0;
    if (builtinTopicDataReader->take(
```

```
    data_seq, info_seq,
    DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE) != DDS_RETCODE_OK) {
    // ... error
}
for (i = 0; i < data_seq.length(); ++i) {
    if (info_seq[i].valid_data) {
        DDSDomainParticipant * localParticipant =
            builtinTopicDataReader->
            get_subscriber()->get_participant();
        DDS_ReturnCode_t retCode;
        // check user_data for access control
        if (data_seq[i].user_data[0] != 0x9) {
            retCode = localParticipant->
                ignore_participant(
                    info_seq[i].instance_handle);
        }else {
            retCode = localParticipant->
                resume_endpoint_discovery(
                    info_seq[i].instance_handle)
        }
    }
}
if (builtinTopicDataReader->return_loan(
    data_seq, info_seq)
    != DDS_RETCODE_OK) {
    // ... error }
}
```

Chapter 17 Configuring QoS with XML

Connex DDS entities are configured by means of Quality of Service (QoS) policies, which may be set programmatically in one of the following ways:

- Directly when the entity is created as an additional argument to the `create_<entity>()` operation (or the Entity's constructor in the Modern C++ API).
- Directly via the `set_qos()` operation on the entity.
- Indirectly as a default QoS on the factory for the entity (`set_default_<entity>_qos()` operations on *Publisher*, *Subscriber*, *DomainParticipant*, *DomainParticipantFactory*)

Entities can also be configured from an XML file or XML string. With this feature, you can change QoS configurations simply by changing the XML file or string—you do not have to recompile the application. This chapter describes how to configure Connex DDS entities using XML:

17.1 Example XML File

The QoS configuration of a *Entity* can be loaded from an XML file or string.

The file contents must follow an important hierarchy: the file contains one or more libraries; each library contains one or more profiles; each profile contains QoS settings.

Let's look at a very basic configuration file, just to get an idea of its contents. You will learn the meaning of each line as you read the rest of this chapter:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A XML configuration file -->
<dds version = 5.0.0>
  <qos_library name="RTTLibrary">
    <!-- A QoS Profile is a set of related QoS -->
    <qos_profile name="StrictReliableCommunicationProfile">
      <datawriter_qos>
        <history>
```

```

        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
        <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
</datawriter_qos>
<datareader_qos>
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
        <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
</datareader_qos>
</qos_profile>
<!-- Individual QoS are shortcuts for QoS Profiles with 1 QoS-->
<datawriter_qos name="KeepAllWriter">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>
</qos_library>
</dds>

```

See `<NDDSHOME>/resource/xml/NDDS_QOS_PROFILES.example.xml` for another example; this file contains the default QoS values for all entity kinds.

17.2 QoS Libraries

A QoS Library is a named set of QoS profiles.

One configuration file may have several QoS libraries, each one defining its own QoS profiles.

All QoS libraries must be declared within `<dds>` and `</dds>` tags. For example:

```

<dds>
  <qos_library name="RTILibrary">
    <!-- Individual QoSs are shortcuts
         for QoS Profiles with 1 QoS -->
    <datawriter_qos name="KeepAllWriter">
      <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
      </history>
    </datawriter_qos>
    <!-- Qos Profile -->
    <qos_profile name=
      "StrictReliableCommunicationProfile">
      <datawriter_qos>
        <history>
          <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
          <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>

```

```

    </datawriter_qos>
    <datareader_qos>
      <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
      </history>
      <reliability>
        <kind>RELIABLE_RELIABILITY_QOS</kind>
      </reliability>
    </datareader_qos>
  </qos_profile>
</qos_library>
</dds>

```

A QoS library can be reopened within the same configuration file or across different configuration files. For example:

```

<dds>
  <qos_library name="RTILibrary">
    ...
  </qos_library>
  ...
  <qos_library name="RTILibrary">
    ...
  </qos_library>
</dds>

```

17.3 QoS Profiles

A QoS *profile* groups a set of related QoS, usually one per entity, identified by a name. For example:

```

<qos_profile name="StrictReliableCommunicationProfile">
  <datawriter_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datawriter_qos>
  <datareader_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datareader_qos>
</qos_profile>

```

Duplicate QoS profiles are not allowed. To overwrite a QoS profile, use [QoS Profile Inheritance \(Section 17.3.3 on page 797\)](#).

There are functions that allow you to create *Entities* using profiles, such as `create_participant_with_profile()` (Creating a DomainParticipant (Section 8.3.1 on page 556)), `create_topic_with_profile()` (Creating Topics (Section 5.1.1 on page 202)), etc.

If you create an entity using a profile without a QoS definition or an inherited QoS definition (see [QoS Profile Inheritance \(Section 17.3.3 on page 797\)](#)) for that class of entity, Connex DDS uses the default QoS.

Example 1:

```
<qos_profile name=
"BatchStrictReliableCommunicationProfile"
base_name="StrictReliableCommunicationProfile">
  <datawriter_qos>
    <batch>
      <enable>true</enable>
    </batch>
  </datawriter_qos>
</qos_profile>
```

The *DataReader* QoS value in the profile **BatchStrictReliableCommunicationProfile** is inherited from the profile **StrictReliableCommunicationProfile**.

Example 2:

```
<qos_profile name="BatchProfile">
  <datawriter_qos>
    <batch>
      <enable>true</enable>
    </batch>
  </datawriter_qos>
</qos_profile>
```

The *DataReader* QoS value in the profile **BatchProfile** is the default Connex DDS QoS.

17.3.1 Built-in QoS Profiles

Several QoS profiles are built into the Connex DDS core libraries and can be used as starting points when configuring QoS for your Connex DDS applications. There are two provided libraries, **BuiltinQosLib** and **BuiltinQosLibExp**, and 34 different profiles. You can use any of these profiles as base profiles when creating your own XML configurations or simply use these profiles directly in the **DDS*_create*_with_profile()** APIs.

There are three types of built-in profiles:

- **Baseline.X.X.X** profiles represent the QoS defaults for Connex DDS version X.X.X. The defaults for the latest Connex DDS version can be accessed using the **BuiltinQosLib::Baseline** profile.

- **Generic.X** profiles allow you to easily configure different features and communication use-cases with Connex DDS. For example, there is a **Generic.StrictReliable** profile for use when your application has a requirement for no data loss, regardless of the application domain.
- **Pattern.X** profiles inherit from **Generic.X** profiles and allow you to configure various domain-specific communication use cases. For example, there is a **Pattern.Alarm** profile that can be used to manage the generation and consumption of alarm events.

The **USER_QOS_PROFILES.xml** file generated by *RTI Code Generator* contains a profile that inherits from the **BuiltinQosLibExp::Generic.StrictReliable** profile as an example of how to use these profiles in your own application.

Example use-cases for these profiles:

- To quickly enable *RTI Monitoring Library* by inheriting from the **BuiltinQosLib::Generic.Monitoring.Common** profile. (See note below.)
- To easily revert to the default QoS values from a previous Connex DDS version by inheriting from the correct **BuiltinQosLib::Baseline.X.X.X** profile.
- To set up common use-case configurations and patterns such as strict reliability or large data communication by inheriting from one of the **BuiltinQosLibExp::Generic.X** or **Pattern.X** profiles.

To see the contents of the built-in QoS profiles:

In `<NDDSHOME>/resource/xml`, you will find:

- **BaselineRoot.documentationONLY.xml**—This file contains the root baseline QoS profile corresponding to the default values of Connex DDS 5.0.0.
- **BuiltinProfiles.documentationONLY.xml**—This file contains the rest of the built-in QoS profiles.

Notes:

- The built-in QoS profiles that enable *RTI Monitoring Library* set the property **rti.monitor.create_function**. Consequently, they only work in Connex DDS applications in which the monitoring library can be loaded dynamically. Specifically, the built-in monitoring profiles will not work in these situations:
 - When the Connex DDS application links the monitoring libraries statically
 - When using a VxWorks 6.7 or 6.8 platform with Java¹.

¹VxWorks 6.7 and 6.8 Java platforms require custom supported libraries.

For more information, see [Part 9: RTI Monitoring Library \(Section on page 1022\)](#)).

- Some of the built-in profiles are experimental. All the experimental profiles are contained within the library **BuiltinQosLibExp**.

17.3.2 Overwriting Default QoS Values

There are two ways to overwrite the default QoS used for new entities with values from a profile: programmatically and with an XML attribute.

- You can overwrite the default QoS programmatically with **set_default_<entity>_qos_with_profile()** (where <entity> is participant, topic, publisher, subscriber, datawriter, or datareader)
- You can overwrite the default QoS using the XML attribute **is_default_qos** with the <qos_profile> tag
- Only for the DomainParticipantFactory: You can overwrite the default QoS using the XML attribute **is_default_participant_factory_profile**. This attribute has precedence over **is_default_qos** if both are set.

In the following example, the *DataWriter* and *DataReader* default QoS will be overwritten with the values specified in a profile named ‘**StrictReliableCommunicationProfile**’:

```
<qos_profile name="StrictReliableCommunicationProfile"
  is_default_qos="true">
  <datawriter_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datawriter_qos>
  <datareader_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datareader_qos>
</qos_profile>
```

If multiple profiles are configured to overwrite the default QoS, only the last one parsed applies.

Example:

In this example, the profile used to configure the default QoSs will be **StrictReliableCommunicationProfile**.

```

<qos_profile name="BestEffortCommunicationProfile"
  is_default_qos="true">
  ...
</qos_profile>

<qos_profile name="StrictReliableCommunicationProfile"
  is_default_qos="true">
  ...
</qos_profile>

```

17.3.3 QoS Profile Inheritance

An individual QoS or profile can inherit values from other QoSs or profiles described in the XML file by using the attribute, **base_name**.

Inheriting from other XML Files:

A QoS or QoS Profile may inherit values from other QoSs or QoS Profiles described in different XML files. A QoS or profile can only inherit from other QoS policies or profiles that have already been loaded. The order in which XML resources are loaded is described in [How to Load XML-Specified QoS Settings \(Section 17.5 on page 810\)](#).

The following examples show how to inherit from other profiles:

Example 1:

```

<qos_library name="Library">
  <qos_profile name="BaseProfile">
    <datawriter_qos>
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
  <qos_profile name="DerivedProfile"
    base_name="BaseProfile">
    <datawriter_qos>
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
</qos_library>

```

The **writer_qos** and **reader_qos** in DerivedProfile inherit their values from the corresponding QoS in BaseProfile.

Example 2:

```

<qos_library name="Library">
  <datareader_qos name="BaseProfile">

```

```

    ...
  </datareader_qos>
  <datareader_qos name="DerivedProfile"
    base_name="BaseProfile"
    ...
  </datareader_qos>
</qos_library>

```

The **datareader_qos** in **DerivedProfile** inherits its values from the **datareader_qos** of **BaseProfile**. In this example, the **datareader_qos** definition is a shortcut for a profile definition with a single QoS.

Example 3:

```

<qos_library name="Library">
  <qos_profile name="Profile1">
    <datawriter_qos name="BaseWriterQoS">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
  <qos_profile name="Profile2">
    <datawriter_qos name="DerivedWriterQoS"
      base_name="Profile1::BaseWriterQoS">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
</qos_library>

```

The **datawriter_qos** in **Profile2** inherits its values from the **datawriter_qos** in **Profile1**. The **datareader_qos** in **Profile2** will not inherit the values from the corresponding QoS in **Profile1**.

Example 4:

```

<qos_library name="Library">
  <qos_profile name="Profile1">
    <datawriter_qos>
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
  <qos_profile name="Profile2">
    <datawriter_qos name="BaseWriterQoS">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
</qos_library>

```

```

</qos_profile>
<qos_profile name="Profile3" base_name="Profile1">
  <datawriter_qos name="DerivedWriterQos"
    base_name="Profile2::BaseWriterQos">
    ...
  </datawriter_qos>
  <datareader_qos>
  ...
</datareader_qos>
</qos_profile>
</qos_library></qos_library>

```

The **datawriter_qos** in Profile3 inherits its values from the **datawriter_qos** in Profile2. The **datareader_qos** in Profile3 inherits its values from the **datareader_qos** in Profile1.

Example 5:

```

<qos_library name="Library">
  <datareader_qos name="BaseProfile">
  ...
</datareader_qos>
<profile name="DerivedProfile" base_name="BaseProfile">
  <datareader_qos>
  ...
</datareader_qos>
</profile>
</qos_library>

```

The **datareader_qos** in DerivedProfile inherits its values from the **datareader_qos** in BaseProfile.

17.3.4 Topic Filters

A QoS profile may contain several writer, reader and topic QoSs. Connex DDS will select a QoS based on the evaluation of a filter expression on the topic name. The filter expression is specified as an attribute in the XML QoS definition. For example:

```

<qos_profile name="StrictReliableCommunicationProfile">
<b>datawriter_qos topic_filter="A*"</b>
  <history>
    <kind>KEEP_ALL_HISTORY_QOS</kind>
  </history>
  <reliability>
    <kind>RELIABLE_RELIABILITY_QOS</kind>
  </reliability>
</datawriter_qos>
<b>datawriter_qos topic_filter="B*"</b>
  <history>
    <kind>KEEP_ALL_HISTORY_QOS</kind>
  </history>
  <reliability>
    <kind>RELIABLE_RELIABILITY_QOS</kind>
  </reliability>
  <resource_limits>

```

```

        <max_samples>128</max_samples>
        <max_samples_per_instance>128
        </max_samples_per_instance>
        <initial_samples>128</initial_samples>
        <max_instances>1</max_instances>
        <initial_instances>1</initial_instances>
    </resource_limits>
</datawriter_qos>
    ...
</qos_profile>

```

If **topic_filter** is not specified in a QoS, Connex DDS will assume the filter '*'. The QoSs with an explicit **topic_filter** attribute definition will be evaluated in order; they have precedence over a QoS without a **topic_filter** expression.

The **topic_filter** attribute is only used with the following APIs:

DomainParticipantFactory:

- **get_<entity>_qos_from_profile_w_topic_name()** (where <entity> may be topic, datareader, or datareader; see [Getting QoS Values from a QoS Profile \(Section 8.2.5 on page 547\)](#))

DomainParticipant:

- **create_datawriter_with_profile()** (see [Creating DataWriters \(Section 6.3.1 on page 266\)](#))
- **create_datareader_with_profile()** (see [Creating DataReaders \(Section 7.3.1 on page 463\)](#))
- **create_topic_with_profile()** (see [Creating Topics \(Section 5.1.1 on page 202\)](#))

Publisher:

- **create_datawriter_with_profile()** (see [Creating DataWriters \(Section 6.3.1 on page 266\)](#))

Subscriber:

- **create_datareader_with_profile()** (see [Creating DataReaders \(Section 7.3.1 on page 463\)](#))

Topic:

- **set_qos_with_profile()** (see [Setting Topic QoS Policies \(Section 5.1.3 on page 204\)](#))

DataWriter:

- **set_qos_with_profile()** (see [Changing QoS Settings After the Publisher Has Been Created \(Section 6.2.4.3 on page 254\)](#))

DataReader:

- **set_qos_with_profile()** (see [Setting DataReader QoS Policies \(Section 7.3.8 on page 482\)](#))

Other APIs will ignore QoSs with a **topic_filter** value different than "*". A QoS Profile with QoSs using **topic_filter** can also inherit from other QoS Profiles. In this case, inheritance will consider the value of the **topic_filter** expression.

Example 1:

```
<qos_library name="Library">
  <qos_profile name="BaseProfile">
    <datawriter_qos>
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T1*">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T2*">
      ...
    </datawriter_qos>
  </qos_profile>
  <qos_profile name="DerivedProfile" base_name="BaseProfile">
    <datawriter_qos topic_filter="T11">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T21">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T31">
      ...
    </datawriter_qos>
  </qos_profile>
</qos_library>
```

The **datawriter_qos** with **topic_filter** T11 in DerivedProfile will inherit its values from the **datawriter_qos** with **topic_filter** T1* in BaseProfile. The **datawriter_qos** with **topic_filter** T21 in DerivedProfile will inherit its values from the **datawriter_qos** with **topic_filter** T2* in BaseProfile. The **datawriter_qos** with **topic_filter** T31 in DerivedProfile will inherit its values from the **datawriter_qos** without **topic_filter** in BaseProfile.

Example 2:

```
<qos_library name="Library">
  <qos_profile name="BaseProfile">
    <datawriter_qos topic_filter="T1*">
      ...
    </datawriter_qos>
```

```

    <datawriter_qos name="T2DataWriterQoS" topic_filter="T2*">
      ...
    </datawriter_qos>
  </qos_profile>
  <qos_profile name="DerivedProfile" base_name="BaseProfile">
    <datawriter_qos topic_filter="T11"
      base_name="BaseProfile::T2DataWriterQoS">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T21">
      ...
    </datawriter_qos>
  </qos_profile>
</qos_library>

```

Although the **topic_filter** expressions do not match, the **datawriter_qos** with **topic_filter** T11 in DerivedProfile will inherit its values from the **datawriter_qos** with **topic_filter** T2* in BaseProfile. **topic_filter** is not used with inheritance from QoS to QoS. The **datawriter_qos** with **topic_filter** T21 in DerivedProfile will inherit its values from the **datawriter_qos** with **topic_filter** T2* in BaseProfile.

Example 3:

```

<qos_library name="Library">
  <datawriter_qos name="BaseQoS" topic_filter="T1">
    ...
  </datawriter_qos>
  <datawriter_qos name="DerivedQoS" base_name="BaseQoS" topic_filter="T2">
    ...
  </datawriter_qos>
</qos_library>

```

In the case of a single QoS profile, although the **topic_filter** expressions do not match, the **datawriter_qos** named DerivedQoS with **topic_filter** T2 will inherit its values from the **datawriter_qos** named BaseQoS with **topic_filter** T1.

17.3.5 QoS Profiles with a Single QoS

The definition of an individual QoS outside a profile is a shortcut for defining a QoS profile with a single QoS. For example:

```

<datawriter_qos name="KeepAllWriter">
  <history>
    <kind>KEEP_ALL_HISTORY_QOS</kind>
  </history>
</datawriter_qos>

```

is equivalent to:

```

<qos_profile name="KeepAllWriter">
  <datawriter_qos>
    <history>

```

```

        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>
</qos_profile>

```

17.4 Configuring QoS with XML

To configure the QoS for an *Entity* using XML, use the following tags:

- **<participant_factory_qos>**

Note: The only QoS policies that can be configured for the DomainParticipantFactory are **<entity_factory>** and **<logging>**.

- **<participant_qos>**
- **<publisher_qos>**
- **<subscriber_qos>**
- **<topic_qos>**
- **<datawriter_qos>** or **<writer_qos>** (writer_qos is valid only with DTD validation)
- **<datareader_qos>** or **<reader_qos>** (reader_qos is valid only with DTD validation)

Each QoS can be identified by a name. The QoS can inherit its values from other QoSs described in the XML file. For example:

```

<datawriter_qos name="DerivedWriterQos" base_name="Lib::BaseWriterQos">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>

```

In the above example, the datawriter_qos named 'DerivedWriterQos' inherits the values from 'BaseWriterQos' in the library 'Lib'. The HistoryQosPolicy **kind** is set to KEEP_ALL_HISTORY_QOS.

Each XML tag with an associated name can be uniquely identified by its fully qualified name in C++ style.

The writer, reader and topic QoSs can also contain an attribute called **topic_filter** that will be used to associate a set of topics to a specific QoS when that QoS is part of a QoS profile. See [Topic Filters \(Section 17.3.4 on page 799\)](#) and [URL Groups \(Section 17.8 on page 814\)](#).

17.4.1 QoS Policies

The fields in a QoS Policy are described in XML using a 1-to-1 mapping with the equivalent C representation. For example, the Reliability QoS Policy is represented with the following C structures:


```

struct DDS_Duration_t {
    DDS_Long sec;
    DDS_UnsignedLong nanosec;
}
struct DDS_ReliabilityQosPolicy {
    DDS_ReliabilityQosPolicyKind kind;
    DDS_Duration_t max_blocking_time;
}

```

The equivalent representation in XML is as follows:

```

<reliability>
  <kind></kind>
  <max_blocking_time>
    <sec></sec>
    <nanosec></nanosec>
  </max_blocking_time>
</reliability>

```

17.4.2 Sequences

In general, sequences in QosPolicies are described with the following XML format:

```

<a_sequence_member_name>
  <element>...</element>
  <element>...</element>
  ...
</a_sequence_member_name>

```

Each element of the sequence is enclosed in an `<element>` tag. For example:

```

<property>
  <value>
    <element>
      <name>my name</name>
      <value>my value</value>
    </element>
    <element>
      <name>my name2</name>
      <value>my value2</value>
    </element>
  </value>
</property>

```

A sequence without elements represents a sequence of length 0. For example:

```

<discovery>
  <!-- initial_peers sequence contains zero elements -->
  <initial_peers/>
</discovery>

```

For sequences that may have a default initialization that is *not empty* (such as the **initial_peers** field in the [DISCOVERY QosPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#)), using the above construct

would result in an empty list and not the default value. So to simply show a sequence for the sake of completeness, but not change its default value, comment it out, as follows:

```
<discovery>
  <!-- initial_peers sequence contains the default value -->
  <!-- <initial_peers/> -->
</discovery>
```

As a general rule, sequences defined in a derived¹ QoS will replace the corresponding sequences in the base QoS. For example, consider the following:

```
<qos_profile name="MyBaseProfile">
  <participant_qos>
    <discovery>
      <initial_peers>
        <element>192.168.1.1</element>
        <element>192.168.1.2</element>
      </initial_peers>
    </discovery>
  </participant_qos>
</qos_profile>
<qos_profile name="MyDerivedProfile" base_name="MyBaseProfile">
  <participant_qos>
    <discovery>
      <initial_peers>
        <element>192.168.1.3</element>
      </initial_peers>
    </discovery>
  </participant_qos>
</qos_profile>
```

The initial peers sequence defined above in the participant QoS of MyDerivedProfile will contain a single element with a value 192.168.1.3. The elements 192.168.1.1 and 192.168.1.2 will not be inherited. However, there is one exception to this behavior. The `<property>` tag provides an attribute called **inherit** that allows you to choose the inheritance behavior for the sequence defined within the tag.

The `<property>` tag provides an attribute called **inherit** that allows you to choose the inheritance behavior for the sequence defined within the tag.

By default, the value of the attribute **inherit** is true. Therefore, the `<property>` tag defined within a derived QoS profile will inherit its elements from the `<property>` tag defined within a base QoS profile.

In the following example, the property sequence defined in the participant QoS of MyDerivedProfile will contain two properties:

- **dds.transport.UDPv4.builtin.send_socket_buffer_size** will be inherited from the base profile and have the value 524288.

¹The concepts of *derived* and *base* QoS are described in [QoS Profile Inheritance \(Section 17.3.3 on page 797\)](#).

- **dds.transport.UDPv4.builtin.recv_socket_buffer_size** will overwrite the value defined in the base QoS profile with 1048576.

```

<qos_profile name="MyBaseProfile">
  <participant_qos>
    <property>
      <value>
        <element>
          <name>
            dds.transport.UDPv4.builtin.send_socket_buffer_size
          </name>
          <value>524288</value>
        </element>
        <element>
          <name>
            dds.transport.UDPv4.builtin.recv_socket_buffer_size
          </name>
          <value>2097152</value>
        </element>
      </value>
    </discovery>
  </property>
</qos_profile>
<qos_profile name="MyDerivedProfile" base_name="MyBaseProfile">
  <participant_qos>
    <property>
      <value>
        <element>
          <name>
            dds.transport.UDPv4.builtin.recv_socket_buffer_size
          </name>
          <value>1048576</value>
        </element>
      </value>
    </discovery>
  </property>
</qos_profile>

```

To discard all the properties defined in the base QoS profile, set **inherit** to false.

In the following example, the property sequence defined in the participant QoS of MyDerivedProfile will contain a single property named **dds.transport.UDPv4.builtin.recv_socket_buffer_size**, with a value of 1048576. The property **dds.transport.UDPv4.builtin.send_socket_buffer_size** will not be inherited.

```

<qos_profile name="MyBaseProfile">
  <participant_qos>
    <property>
      <value>
        <element>
          <name>
            dds.transport.UDPv4.builtin.send_socket_buffer_size
          </name>
          <value>524288</value>
        </element>

```

```

    <element>
      <name>
        dds.transport.UDPv4.builtin.recv_socket_buffer_size
      </name>
      <value>2097152</value>
    </element>
  </value>
</discovery>
</property>
</qos_profile>
<qos_profile name="MyDerivedProfile" base_name="MyBaseProfile"
  <participant_qos>
    <property inherit="false">
      <value>
        <element>
          <name>
            dds.transport.UDPv4.builtin.recv_socket_buffer_size
          </name>
          <value>1048576</value>
        </element>
      </value>
    </discovery>
  </property>
</qos_profile>

```

17.4.3 Arrays

In general, the arrays contained in the QoS Policies are described with the following XML format:

```

<an_array_member_name>
  <element>...</element>
  <element>...</element>
  ...
</an_array_member_name>

```

Each element of the array is enclosed in an `<element>` tag.

As a special case, arrays of octets are represented with a single XML tag enclosing an array of decimal/hexadecimal values between 0..255 separated with commas.

For example:

```

<reader_qos>
  ...
  <protocol>
    <virtual_guid>
      <value>
        1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
      </value>
    </virtual_guid>
  </protocol>
</reader_qos>

```

17.4.4 Enumeration Values

Enumeration values are represented using their C or Java string representation. For example:

```
<history>
  <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
</history>
```

or

```
<history>
  <kind>KEEP_ALL_HISTORY_QOS</kind>
</history>
```

When the XSD document is used for validation during editing (see [XML File Validation During Editing \(Section 17.9.2 on page 816\)](#)), only the Java representation is valid.

17.4.5 Time Values (Durations)

You can use the following special values for fields that require seconds or nanoseconds:

- DURATION_INFINITE_SEC or DDS_DURATION_INFINITE_SEC,
- DURATION_ZERO_SEC or DDS_DURATION_ZERO_SEC,
- DURATION_INFINITE_NSEC or DDS_DURATION_INFINITE_NSEC
- DURATION_ZERO_NSEC or DDS_DURATION_ZERO_NSEC

For example:

```
<deadline>
  <period>
    <sec>DURATION_INFINITE_SEC</sec>
    <nanosec>DURATION_INFINITE_NSEC</nanosec>
  </period>
</deadline>
```

When the XSD document is used for validation during editing (see [XML File Validation During Editing \(Section 17.9.2 on page 816\)](#)), only the values without the DDS prefix are considered valid.

17.4.6 Transport Properties

You can configure transport plugins using the *DomainParticipant's* [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#).

- Properties for the builtin transports are described in [Setting Builtin Transport Properties with the PropertyQosPolicy \(Section 15.6 on page 748\)](#).

- Properties for other transport plugins such as *RTI TCP Transport*¹ are described in their respective chapters in this manual.

For example:

```
<participant_qos>
  <property>
    <value>
      <element>
        <name>
          dds.transport.UDPv4.builtin.parent.message_size_max
        </name>
        <value>65507</value>
      </element>
      <element>
        <name>
          dds.transport.UDPv4.builtin.send_socket_buffer_size
        </name>
        <value>131072</value>
      </element>
      <element>
        <name>
          dds.transport.UDPv4.builtin.recv_socket_buffer_size
        </name>
        <value>131072</value>
      </element>
    </value>
  </property>
</participant_qos>
```

17.4.7 Thread Settings

See [Table 19.1 XML Tags for ThreadSettings_t](#).

17.4.8 Entity Names

The **name** and **role_name** fields in the [ENTITY_NAME QosPolicy \(DDS Extension\)](#) ([Section 6.5.9 on page 374](#)) have three distinct possible values: NULL, an empty string, and a non-empty string. Each of these three states are specified in XML in a different way.

To specify that the **name** or **role_name** of an entity is NULL, use the **xsi:nil** attribute. The **xsi:nil** attribute can be set to either "true" or "false". For example, to set the participant name to NULL:

```
<participant_name>
  <name xsi:nil="true">
</participant_name>
```

To specify the empty string, leave the XML element empty:

¹*RTI TCP Transport* is included with Connex DDS, but is not enabled by default.

```
<participant_name>
  <name/>
</participant_name>
```

To specify a non-empty string:

```
<participant_name>
  <name>"My Participant's Name"</name>
</participant_name>
```

17.5 How to Load XML-Specified QoS Settings

There are several ways to load XML QoS profiles into your application. In C, Traditional C++, Java and .NET, it's the singleton DomainParticipantFactory that loads these profiles. Applications using the Modern C++ API can create any number of instances of `dds::core::QosProvider` with different parameters to load different QoS profiles or, they can use the singleton `QosProvider::Default()`. The profiles configured in the default `QosProvider` are used when creating an Entity without a explicit QoS parameter.

Here are the various approaches, listed in load order:

- `$NDDSHOME/resource/xml/NDDS_QOS_PROFILES.xml`**
 This file is loaded automatically *if it exists* (not the default) *and* `ignore_resource_profile` in the [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#) is FALSE (the default). `NDDS_QOS_PROFILES.xml` does not exist by default. However, `NDDS_QOS_PROFILES.example.xml` is shipped with the host bundle of the product; you can copy it to `NDDS_QOS_PROFILES.xml` and modify it for your own use. The file contains the default QoS values that will be used for all entity kinds. (*First to be loaded*)
- URL Groups in `NDDS_QOS_PROFILES`**
 URL groups (see [URL Groups \(Section 17.8 on page 814\)](#)) separated by semicolons referenced by the environment variable `NDDS_QOS_PROFILES` are loaded automatically if they exist *and* `ignore_environment_profile` in [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#) is FALSE (the default).
- `<working directory>/USER_QOS_PROFILES.xml`**
 This file is loaded automatically if it exists *and* `ignore_user_profile` in [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#) is FALSE (the default).
- URL groups in `url_profile`**
 URL groups (see [URL Groups \(Section 17.8 on page 814\)](#)) referenced by `url_profile` (in [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#)) will be loaded automatically if specified.
- XML strings in `string_profile`**
 The sequence of XML strings referenced by `string_profile` (in [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#)) will be loaded automatically if specified. (*Last to be loaded*)

You may use a combination of the above approaches.

The location of the XML documents (only files and strings are supported) is specified using URL (Uniform Resource Locator) format. For example:

- File Specification: **file:///usr/local/default_dds.xml**
- String Specification: **str://"<dds><qos_library>...</qos_library></dds>"**

If you omit the URL schema name, Connex DDS will assume a file name. For example:

- File Specification: **/usr/local/default_dds.xml**

Duplicate QoS profiles are not allowed. Connex DDS will report an error message in these scenarios. To overwrite a QoS profile, use [QoS Profile Inheritance \(Section 17.3.3 on page 797\)](#).

Several QoS profiles are built into the Connex DDS core libraries and can be used as starting points when configuring QoS for your Connex DDS applications. For details, see [Configuring QoS with XML \(Section 17.4 on page 803\)](#).

17.5.1 Loading, Reloading and Unloading Profiles

You do not have to explicitly call `load_profiles()`. QoS profiles are loaded when any of these DomainParticipantFactory operations are called:

- `create_participant()` (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#))
- `create_participant_with_profile()` (see [Creating a DomainParticipant \(Section 8.3.1 on page 556\)](#))
- `get_<entity>_qos_from_profile()` (where <entity> is **participant**, **topic**, **publisher**, **subscriber**, **datawriter**, or **datareader**) (see [Getting QoS Values from a QoS Profile \(Section 8.2.5 on page 547\)](#))
- `get_<entity>_qos_from_profile_w_topic_name()` (where <entity> is **topic**, **datawriter**, or **datareader**) (see [Getting QoS Values from a QoS Profile \(Section 8.2.5 on page 547\)](#))
- `get_default_participant_qos()` (see [Getting and Setting Default QoS for DomainParticipants \(Section 8.2.2 on page 545\)](#))
- `get_qos_profile_libraries()` (See [Retrieving a List of Available Libraries \(Section 17.10.1 on page 823\)](#))
- `get_qos_profiles()` (See [Configuring QoS with XML \(Section 17.4 on page 803\)](#))
- `load_profiles()`
- `set_default_participant_qos_with_profile()` (see [Getting and Setting Default QoS for DomainParticipants \(Section 8.2.2 on page 545\)](#))

- **set_default_library()** (see [Getting and Setting the Publisher’s Default QoS Profile and Library \(Section 6.2.4.4 on page 255\)](#))
- **set_default_profile()** (see [Getting and Setting the Publisher’s Default QoS Profile and Library \(Section 6.2.4.4 on page 255\)](#))

In the Modern C++ API, the previous operations cause the default QosProvider (QosProvider::Default()) to load the QoS profiles. Any other QosProvider that an application instantiates will load the QoS Profiles it is configured to load in its constructor.

QoS profiles are reloaded when either of these DomainParticipantFactory operations are called:

- **reload_profiles()**
- **set_qos()** (see [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#))

It is important to distinguish between loading and reloading:

- *Loading* only happens when there are no previously loaded profiles. This could be when the profiles are loaded the first time or after a call to **unload_profiles()**.
- *Reloading* replaces all previously loaded profiles. Reloading a profile does not change the QoS of entities that have already been created with previously loaded profiles.

The DomainParticipantFactory also has an **unload_profiles()** operation that frees the resources associated with the XML QoS profiles.

```
DDS_ReturnCode_t unload_profiles()
```

17.6 XML File Syntax

The contents of the XML configuration file must follow an important hierarchy: the file contains one or more libraries; each library contains one or more profiles; each profile contains QoS settings.

In addition, the file must follow these syntax rules:

- The syntax is XML and the character encoding is UTF-8.
- Opening tags are enclosed in `<>`; closing tags are enclosed in `</>`.
- A tag value is a UTF-8 encoded string. Legal values are alphanumeric characters. The middleware’s parser will remove all leading and trailing spaces^afrom the string before it is processed.

^aLeading and trailing spaces in enumeration fields will not be considered valid if you use the distributed XSD document to do validation at run-time with a code editor (see [URL Groups \(Section 17.8 on page 814\)](#)).

- For example, `<tag> value </tag>` is the same as `<tag>value</tag>`.
- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: `<!-- comment -->`.
- The root tag of the configuration file must be `<dds>` and end with `</dds>`.
- The primitive types for tag values are specified in [Table 17.1 Supported Tag Values](#).

Table 17.1 Supported Tag Values

Type	Format	Notes
DDS_Boolean	yes ^a , 1, true, BOOLEAN_TRUE or DDS_BOOLEAN_TRUE: these all mean TRUE	Not case-sensitive
	no, 0, false, BOOLEAN_FALSE or DDS_BOOLEAN_FALSE: these all mean FALSE	
DDS_Enum	A string. Legal values are those listed in the API Reference HTML documentation for the C or Java API.	Must be specified as a string. (Do not use numeric values.)
DDS_Long	-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff or LENGTH_UNLIMITED or DDS_LENGTH_UNLIMITED	A 32-bit signed integer
DDS_UnsignedLong	0 to 4294967296 or 0 to 0xffffffff	A 32-bit unsigned integer
String	UTF-8 character string	All leading and trailing spaces are ignored between two tags

17.6.1 Using Environment Variables in XML

The text within an XML tag and attribute can refer to environment variable. To do so, use the following notation:

```
$ (MY_VARIABLE)
```

For example:

^aThese values will not be considered valid if you use the distributed XSD document to do validation at run-time with a code editor (see [URL Groups \(Section 17.8 on the next page\)](#)).

```
<element attr="The attribute is $(MY_ATTRIBUTE) ">
  <name>The name is $(MY_NAME)</name>
  <value>The value is $(MY_VALUE)</value>
</element>
```

When the Connex DDS XML parser parses the above tags, it will replace the references to environment variables with their actual values.

17.7 XML String Syntax

XML profiles can be described using strings. This configuration is useful for architectures without a file system.

There are two different ways to configure *Entities* via XML strings:

- String URLs are prefixed by the URI schema **str://** and enclosed in double quotes. For example:

```
str://"<dds><qos_library>...</qos_library></dds>"
```

The string URLs can be specified in the environment variable **NDDS_QOS_PROFILES** as well as in the field **url_profile** in [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#).

Each string URL must contain a whole XML document.

- The **string_profile** field in the [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#) allows you to split an XML document into multiple strings. For example:

```
const char * MyXML[4] =
{
    "<dds>",
    "<qos_library name=\"MyLibrary\">",
    "</qos_library>",
    "</dds>"
};
factoryQos.profile.string_profile.from_array(MyXML, 4);
```

Only one XML document can be specified with the **string_profile** field.

17.8 URL Groups

To provide redundancy and fault tolerance, you can specify multiple locations for a single XML document via URL groups. The syntax of a URL group is:

```
[URL1 | URL2 | URL2 | ... | URLn]
```

For example:

```
[file:///usr/local/default_dds.xml | file:///usr/local/alternative_default_dds.xml]
```

Only one of the elements in the group will be loaded by Connex DDS, starting from the left.

Brackets are not required for groups with a single URL.

The **NDDS_QOS_PROFILES** environment variable contains a set of URL groups separated by semi-colons. For example, on Linux and Solaris systems (note: this should be entered in a single command line):

```
setenv NDDS_QOS_PROFILES
[file:///usr/local/default_dds.xml|file:///usr/local/alternative_default_dds.xml];
[str://"<dds><qos_library name="MyQosLibrary"></qos_library></dds>"]
```

The **url_profile** field in the [PROFILE QosPolicy \(DDS Extension\) \(Section 8.4.2 on page 573\)](#) will contain a sequence of URL groups.

17.9 How the XML is Validated

17.9.1 Validation at Run-Time

Connex DDS validates the input XML files using a builtin Document Type Definition (DTD).

You can find a copy of the builtin DTD in `<NDDSHOME>/resource/schema/rti_dds_qos_profiles.dtd`. (This is only a *copy* of what the Connex DDS core uses. Changing this file has no effect unless you specify its path with the `<!DOCTYPE>` tag, described below.)

You can overwrite the builtin DTD by using the XML tag, `<!DOCTYPE>`. For example, the following indicates that Connex DDS must use a DTD file from a user's directory to perform validation:

```
<!DOCTYPE dds SYSTEM "/local/joe/rti/dds/mydds.dtd">
```

- The DTD path can be absolute, or relative to the application's current working directory.
- If the specified file does not exist, you will see the following error:

```
RTIXMLDtdParser_parse:!open DTD file
```

- If you do not specify the `DOCTYPE` tag in the XML file, the builtin DTD is used.
- The XML files used by Connex DDS can be versioned using the attribute `version` in the `<dds>` tag. For example:

```
<dds version="5.x.y">
  ...
</dds>
```

Although the attribute version is not required during the validation process, it helps to detect DTD incompatibility scenarios by providing better error messages.

For example, if an application using Connex DDS 5.x.y tries to load an XML file from Connex DDS 4.5z and there is some incompatibility in the XML content, the following parsing error will be printed:

```
ATTENTION: The version declared in this file (4.5z) is different from the
version of Connex DDS (5.x.y). If these versions are not compatible, that
incompatibility could be the cause of this error.
```

17.9.2 XML File Validation During Editing

Connex DDS provides DTD and XSD files that describe the format of the XML content. We recommend including a reference to one of these documents in the XML file that contains the QoS profiles—this provides helpful features in code editors such as Visual Studio and Eclipse, including validation and auto-completion while you are editing the XML file.

The DTD and XSD definitions of the XML elements are in `<NDDSHOME>/resource/schema/rti_dds_qos_profiles.dtd` and `<NDDSHOME>/resource/schema/rti_dds_qos_profiles.xsd`, respectively. (`<NDDSHOME>` is described in [Paths Mentioned in Documentation \(Section on page xxxviii\)](#).)

To include a reference to the XSD document in your XML file, use the attribute `xsi:noNamespaceSchemaLocation` in the `<dds>` tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
    "<NDDSHOME>/resource/schema/rti_dds_qos_profiles.xsd">
    ...
</dds>
```

To include a reference to the DTD document in your XML file use the `<!DOCTYPE>` tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dds SYSTEM
    "<NDDSHOME>/resource/schema/rti_dds_qos_profiles.dtd">
<dds>
    ...
</dds>
```

We recommend including a reference to the XSD file in the XML documents because it provides stricter validation and better auto-completion than the corresponding DTD file.

17.10 Using QoS Profiles in Your Connex DDS Application

You can use the operations listed in [Table 17.2 Operations for Working with QoS Profiles](#) to refer to and use QoS profiles (see [URL Groups \(Section 17.8 on page 814\)](#)) described in XML files and XML strings.

Table 17.2 Operations for Working with QoS Profiles

Working With ...	Profile-Related Operations	Reference
DataReaders	set_qos_with_profile	Changing QoS Settings After DataReader Has Been Created (Section 7.3.8.3 on page 487)
DataWriters	set_qos_with_profile	Changing QoS Settings After the DataWriter Has Been Created (Section 6.3.15.3 on page 305)

Table 17.2 Operations for Working with QoS Profiles

Working With ...	Profile-Related Operations	Reference
DomainParticipants	create_datareader_with_profile	Creating DataReaders (Section 7.3.1 on page 463)
	create_datawriter_with_profile	Creating DataWriters (Section 6.3.1 on page 266)
	create_publisher_with_profile	Creating Publishers (Section 6.2.2 on page 249)
	create_subscriber_with_profile	Creating Subscribers (Section 7.2.2 on page 445)
	create_topic_with_profile	Creating Topics (Section 5.1.1 on page 202)
	get_default_library	Getting and Setting DomainParticipant's Default QoS Profile and Library (Section 8.3.6.4 on page 567)
	get_default_profile	
	get_default_profile_library	
	set_default_datareader_qos_with_profile	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	set_default_datawriter_qos_with_profile	
	set_default_library	Getting and Setting DomainParticipant's Default QoS Profile and Library (Section 8.3.6.4 on page 567)
	set_default_profile	
	set_default_publisher_qos_with_profile	Getting and Setting Default QoS for Child Entities (Section 8.3.6.5 on page 568)
	set_default_subscriber_qos_with_profile	
	set_default_topic_qos_with_profile	
set_qos_with_profile	Changing QoS Settings After DomainParticipant Has Been Created (Section 8.3.6.3 on page 566)	

Table 17.2 Operations for Working with QoS Profiles

Working With ...	Profile-Related Operations	Reference
DomainParticipantFactory	create_participant_with_profile	Creating a DomainParticipant (Section 8.3.1 on page 556)

Table 17.2 Operations for Working with QoS Profiles

Working With ...	Profile-Related Operations	Reference
	get_datareader_qos_from_profile	Getting QoS Values from a QoS Profile (Section 8.2.5 on page 547)

Table 17.2 Operations for Working with QoS Profiles

Working With ...	Profile-Related Operations	Reference
	get_datawriter_qos_from_profile	
	get_datawriter_qos_from_profile_w_topic_name	
	get_datareader_qos_from_profile_w_topic_name	
	get_default_library	
	get_default_profile	Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library (Section 8.2.1.1 on page 544)
	get_default_profile_library	
	get_participant_qos_from_profile	
	get_publisher_qos_from_profile	Getting QoS Values from a QoS Profile (Section 8.2.5 on page 547)
	get_subscriber_qos_from_profile	
	get_topic_qos_from_profile	
	get_topic_qos_from_profile_w_topic_name	
	get_qos_profiles	Retrieving a List of Available QoS Profiles (Section 17.10.2 on page 823)
	get_qos_profile_libraries	Retrieving a List of Available Libraries (Section 17.10.1 on page 823)
	load_profiles	Loading, Reloading and Unloading Profiles (Section 17.5.1 on page 811)
	reload_profiles	
	set_default_participant_qos_with_profile	Getting and Setting Default QoS for DomainParticipants (Section 8.2.2 on page 545)
	set_default_library	Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library (Section 8.2.1.1 on page 544)
	set_default_profile	
	unload_profiles	Loading, Reloading and Unloading Profiles (Section 17.5.1 on page 811)

Table 17.2 Operations for Working with QoS Profiles

Working With ...	Profile-Related Operations	Reference
Publishers	<code>create_datawriter_with_profile</code>	Creating Publishers (Section 6.2.2 on page 249)
	<code>get_default_library</code>	Getting and Setting the Publisher's Default QoS Profile and Library (Section 6.2.4.4 on page 255)
	<code>get_default_profile</code>	
	<code>get_default_profile_library</code>	
	<code>set_default_datawriter_qos_with_profile</code>	Getting and Setting Default QoS for DataWriters (Section 6.2.4.5 on page 256)
	<code>set_default_library</code>	Getting and Setting the Publisher's Default QoS Profile and Library (Section 6.2.4.4 on page 255)
	<code>set_default_profile</code>	
	<code>set_qos_with_profile</code>	Changing QoS Settings After the Publisher Has Been Created (Section 6.2.4.3 on page 254)
Subscribers	<code>create_datareader_with_profile</code>	Creating DataReaders (Section 7.3.1 on page 463)
	<code>get_default_library</code>	Getting and Settings Subscriber's Default QoS Profile and Library (Section 7.2.4.4 on page 451)
	<code>get_default_profile</code>	
	<code>get_default_profile_library</code>	
	<code>set_default_datareader_qos_with_profile</code>	Getting and Setting Default QoS for DataReaders (Section 7.2.4.5 on page 452)
	<code>set_default_library</code>	Getting and Settings Subscriber's Default QoS Profile and Library (Section 7.2.4.4 on page 451)
	<code>set_default_profile</code>	
	<code>set_qos_with_profile</code>	Changing QoS Settings After Subscriber Has Been Created (Section 7.2.4.3 on page 450)
Topics	<code>set_qos_with_profile</code>	Setting Topic QoS Policies (Section 5.1.3 on page 204)

Note: For the Modern C++ API, please refer to the RTI Connex DDS API Reference HTML documentation, [Configuring QoS Profiles with XML](#).

17.10.1 Retrieving a List of Available Libraries

To get a list of available QoS libraries, call the DomainParticipantFactory's **get_qos_profile_libraries()** operation, which returns the names of all QoS libraries that have been loaded by Connex DDS.

```
DDS_ReturnCode_t get_qos_profile_libraries (struct DDS_StringSeq *profile_names)
```

17.10.2 Retrieving a List of Available QoS Profiles

To get a list of available QoS profiles, call the DomainParticipantFactory's **get_qos_profiles()** operation, which returns the names of all profiles within a specified QoS library. Either the input QoS library name must be specified or the default profile library must have been set prior to calling this function.

```
DDS_ReturnCode_t get_qos_profiles (struct DDS_StringSeq *profile_names,  
                                  const char *library_name)
```

17.11 Configuring Logging Via XML

Logging can be configured via XML using the DomainParticipantFactory's LoggingQosPolicy. See [Configuring Logging via XML \(Section 21.2.2 on page 871\)](#) for additional details.

Chapter 18 Multi-channel DataWriters

In Connex DDS, producers publish data to a *Topic*, identified by a topic name; consumers subscribe to a *Topic* and optionally to specific content by means of a content-filter expression.

A Market Data Example:

A producer can publish data on the Topic "MarketData" which can be defined as a structured record containing fields that identify the exchange (e.g., "NYSE" or "NASDAQ"), the stock symbol (e.g., "APPL" or "JPM"), volume, bid and ask prices, etc.

Similarly, a consumer may want to subscribe to data on the "MarketData" Topic, but only if the exchange is "NYSE" or the symbol starts with the letter "M." Or the consumer may want all the data from the "NYSE" whose volume exceeds a certain threshold, or may want MarketData for a specific stock symbol, regardless of the exchange, and so on.

The middleware's efficient implementation of content-filtering is critical for scenarios such as the above "Market Data" example, where there are large numbers of consumers, large volumes of data, or Topics that transmit information about many data-objects or subjects (e.g., individual stocks).

Traditionally, middleware products use four approaches to implement content filtering: Producer-based, Consumer-based, Server-based, and Network Switch-based.

- **Producer-based approaches** push the burden of filtering to the producer side. The producer knows what each consumer wants and delivers to the consumer only the data that matches the consumer's filter. This approach is suitable when using point-to-point protocols such as TCP—it saves bandwidth and lowers the load on the consumer—but it does not work if data is distributed via multicast. Also, this approach does not scale to large numbers of consumers, because the producer would be overburdened by the need to filter for each individual consumer.

- **Consumer-based approaches** push the burden of filtering to the consumer side. The producer sends all the data to every consumer and the middleware on the consumer side decides whether the application wants it or not, automatically filtering the unwanted data. This approach is simple and fits well in systems that use multicast protocols as a transport. But the approach is not efficient for consumers that want small subsets of the data, since the consumers have to spend a lot of time filtering unwanted data. This approach is also unsuitable for systems with large volumes of data, such as the above Market Data system.
- **Server-based approaches** push the burden of filtering to a third component: a server or broker. This approach has some scalability advantages—the server can be run on a more powerful computer and can be federated to handle a large number of consumers. Some providers also provide hardware-assisted filtering in the server. However, the server-based approach significantly increases latency and jitter. It is also far more expensive to deploy and manage.
- **Network Switch-based approaches** leverage the network hardware, specifically advanced (IGMP snooping) network switches, to offload most of the burden of filtering from the producers and consumers without introducing additional hardware, servers or proxies. This approach preserves the low latency and ease of deployment of the brokerless approaches while still providing most of the offloading and scalability benefits of the broker.

RTI supports the producer-based, consumer-based and network-switch approaches to content filtering:

- RTI automatically uses the producer-based and consumer-based approaches as soon as it detects a consumer that specifies a content filter. The producer-based approach is used if the consumer is receiving data over a point-to-point protocol (i.e., not multicast) and the number of consumers that specify filters is reasonably low (below 32). Otherwise, RTI uses a subscriber-based approach.
- To use the more scalable network-switched based approach, an application must configure the *DataWriter* as a *Multi-channel DataWriter*. This concept is described in the following section.

18.1 What is a Multi-channel DataWriter?

A *Multi-channel DataWriter* is a *DataWriter* that is configured to send data over multiple multicast addresses, according to some filtering criteria applied to the data.

To determine which multicast addresses will be used to send the data, the middleware evaluates a set of filters that are configured for the *DataWriter*. Each filter "guards" a *channel*—a set of multicast addresses. Each time a multi-channel *DataWriter* writes data, the filters are applied. If a filter evaluates to true, the data is sent over that filter's associated channel (set of multicast addresses). We refer to this type of filter as a *Channel Guard filter*.

Figure 18.1 Multi-channel Data Flow

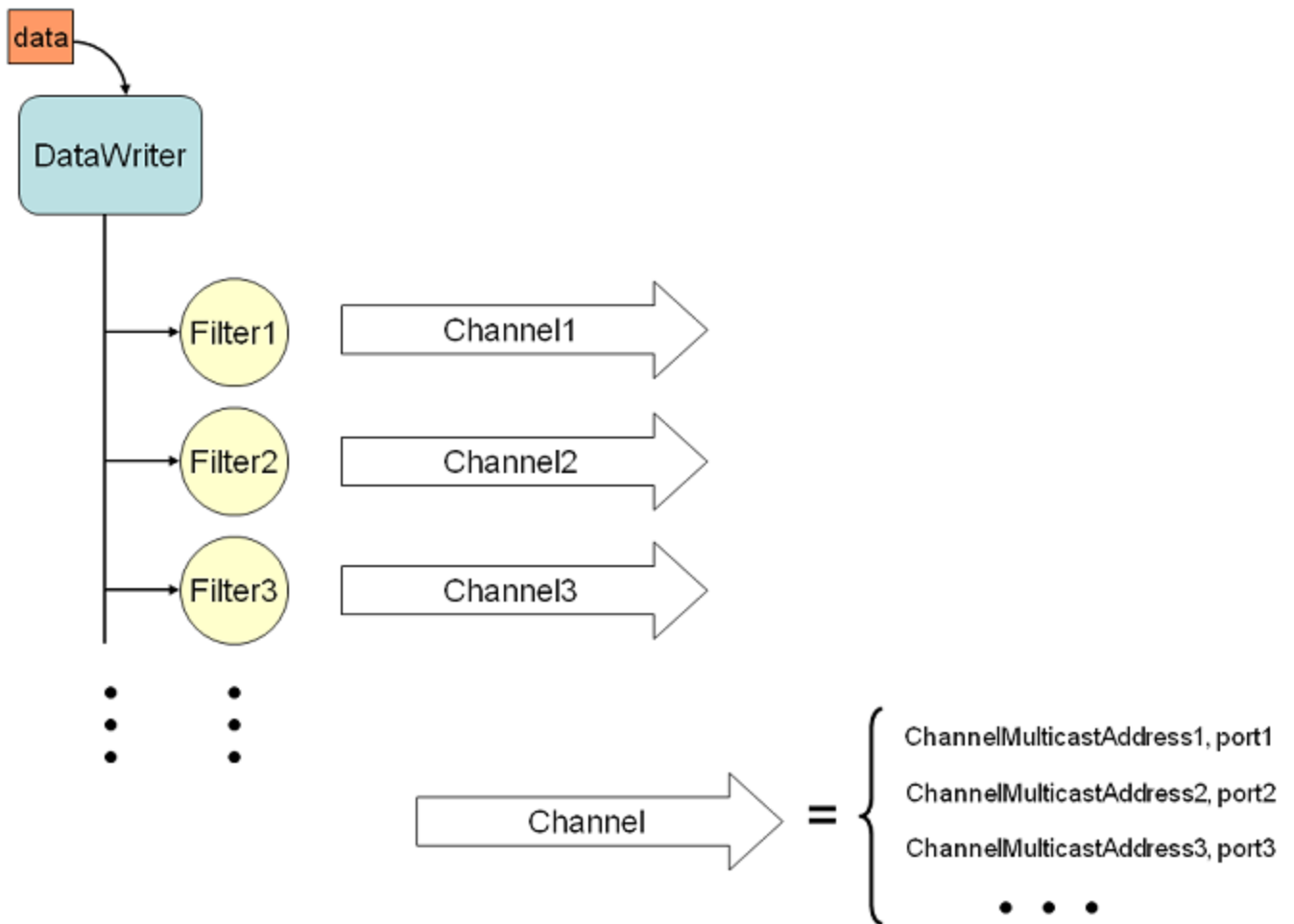
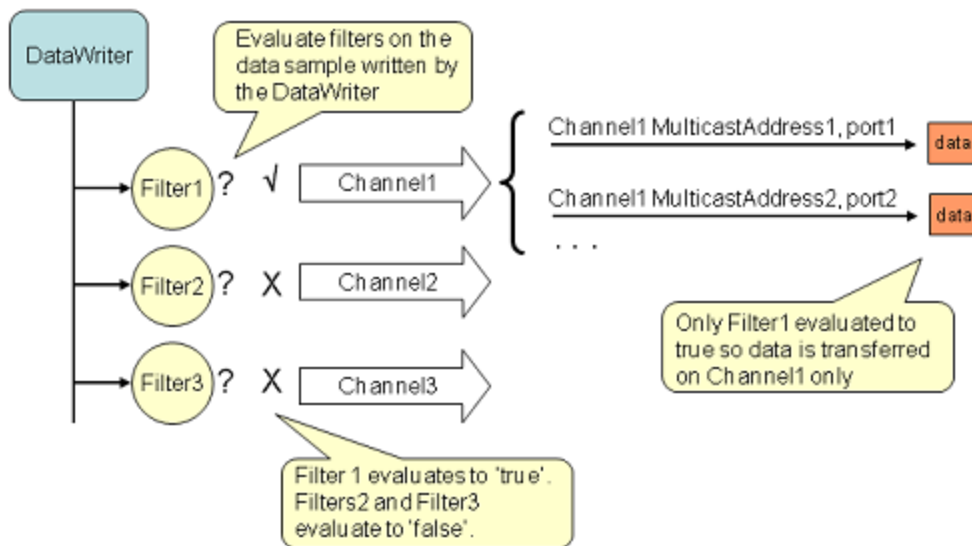


Figure 18.2 Multi-Channel Evaluation



Multi-channel *DataWriters* can be used to trade off network bandwidth with the unnecessary processing of unwanted data for situations where there are multiple *DataReaders* who are interested in different subsets of data that come from the same data stream (Topic). For example, in Financial applications, the data stream may be quotes for different stocks at an exchange. Applications usually only want to receive data (quotes) for only a subset of the stocks being traded. In tracking applications, a data stream may carry information on hundreds or thousands of objects being tracked, but again, applications may only be interested in a subset.

The problem is that the most efficient way to deliver data to multiple applications is to use multicast so that a data value is only sent once on the network for any number of subscribers to the data. However, using multicast, an application will receive all of the data sent and not just the data in which it is interested, thus extra CPU time is wasted to throw away unwanted data. With this QoS, you can analyze the data-usage patterns of your applications and optimize network vs. CPU usage by partitioning the data into multiple multicast streams. While network bandwidth is still being conserved by sending data only once using multicast, most applications will only need to listen to a subset of the multicast addresses and receive a reduced amount of unwanted data.

Note: Your system can gain more of the benefits of using multiple multicast groups if your network uses Layer 2 Ethernet switches. Layer 2 switches can be configured to only route multicast packets to those ports that have added membership to specific multicast groups. Using those switches will ensure that only the multicast packets used by applications on a node are routed to the node; all others are filtered-out by the switch.

18.2 How to Configure a Multi-channel DataWriter

To configure a multi-channel *DataWriter*, simply define a list of all its *channels* in the *DataWriter*'s [MULTI_CHANNEL QosPolicy \(DDS Extension\)](#) (Section 6.5.14 on page 386).

Each *channel* consists of filter criterion to apply to the data and a set of multicast destinations (transport, address, port) that will be used for sending data that matches the filter. You can think of this sequence of channels as a table like the one shown below:

If the Data Matches this Filter...	Send the Data to these Multicast Destinations
Symbol MATCH '[A-K]*'	UDPv4:225.0.0.1:9000
Symbol MATCH '[L-Q]*'	UDPv4:225.0.0.2:9001
Symbol MATCH '[P-Z]*'	UDPv4:225.0.0.3:9002; 225.0.0.4:9003;

The example C++ code in [Using the MULTI_CHANNEL QosPolicy](#) (Section Figure 18.3 on the next page) shows how to configure the channels.

Figure 18.3 Using the MULTI_CHANNEL QosPolicy

```

// initialize writer_qos with default values
publisher->get_default_datawriter_qos(writer_qos);
// Initialize MULTI_CHANNEL Qos Policy
// Assign the filter name
// Possible options: DDS_STRINGMATCHFILTER_NAME, DDS_SQLFILTER_NAME
writer_qos.multi_channel.filter_name =
    (char*) DDS_STRINGMATCHFILTER_NAME;
// Create two channels
writer_qos.multi_channel.channels.ensure_length(2,2);
// First channel
writer_qos.multi_channel.channels[0].filter_expression =
    DDS_String_dup("Symbol MATCH '[A-M]*'");
writer_qos.multi_channel.channels[0].
    multicast_settings.ensure_length(1,1);
writer_qos.multi_channel.channels[0].
    multicast_settings[0].receive_port = 8700;
writer_qos.multi_channel.channels[0].
    multicast_settings[0].receive_address =
    DDS_String_dup("239.255.1.1");
// Second channel
writer_qos.multi_channel.channels[1].
    multicast_settings.ensure_length(1,1);
writer_qos.multi_channel.channels[1].
    multicast_settings[0].receive_port = 8800;
writer_qos.multi_channel.channels[1].
    multicast_settings[0].receive_address =
    DDS_String_dup("239.255.1.2");
writer_qos.multi_channel.channels[1].filter_expression =
    DDS_String_dup("Symbol MATCH '[N-Z]*'");
// Create writer
writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);

```

The MULTI_CHANNEL QosPolicy is propagated along with discovery traffic. The value of this policy is available in the builtin topic for the publication (see the **locator_filter** field in [Table 16.2 Publication Built-in Topic's Data Type \(DDS_PublicationBuiltinTopicData\)](#)).

18.2.1 Limitations

When considering use of a multi-channel DataWriter, please be aware of the following limitations:

- A *DataWriter* that uses the MULTI_CHANNEL QosPolicy will ignore multicast and unicast addresses specified on the reader side through the [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\)](#) (Section 7.6.5 on page 529) and [TRANSPORT_UNICAST QosPolicy \(DDS Extension\)](#) (Section 6.5.24 on page 412). The *DataWriter* will not publish DDS samples on these

locators.

- Multi-channel *DataWriters* cannot be configured to use the Durable Writer History feature (described in [Durable Writer History \(Section 12.3 on page 681\)](#)).
- Multi-channel *DataWriters* do not support fragmentation of large data.
- Multi-channel *DataWriters* cannot be configured for asynchronous publishing (described in [ASYNCHRONOUS_PUBLISHER QosPolicy \(DDS Extension\) \(Section 6.4.1 on page 313\)](#)).
- Multi-channel *DataWriters* rely on the `rtps_object_id` in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\) \(Section 6.5.3 on page 347\)](#) to be `DDS_RTPS_AUTO_ID` (which causes automatic assignment of object IDs to channels).
- To guarantee reliable delivery, a *DataReader's* [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#) must be set to per-instance ordering (`DDS_INSTANCE_PRESENTATION_QOS`, the default value), instead of per-topic ordering (`DDS_TOPIC_PRESENTATION_QOS`), and the matching *DataWriter's* [MULTI_CHANNEL QosPolicy \(DDS Extension\) \(Section 6.5.14 on page 386\)](#) must use expressions that only refer to key fields.

18.3 Multi-Channel Configuration on the Reader Side

No special changes are required in a subscribing application to get data from a multi-channel *DataWriter*.

If you want the *DataReader* to subscribe to only a subset of the channels, use a `ContentFilteredTopic`, as described in [ContentFilteredTopics \(Section 5.4 on page 212\)](#). For example:

```
// Create a content filtered topic
contentFilter =
    participant->create_contentfilteredtopic_with_filter(
        "FilteredTopic",
        topic,
        "symbol MATCH 'NYSE/BAC,NASDAQ/MSFT,NASDAQ/GOOG'",
        parameters,
        DDS_STRINGMATCHFILTER_NAME);
// Create a DataReader that uses the content filtered topic
reader = subscriber->create_datareader(contentFilter,
    DDS_DATAREADER_QOS_DEFAULT,
    NULL, 0);
```

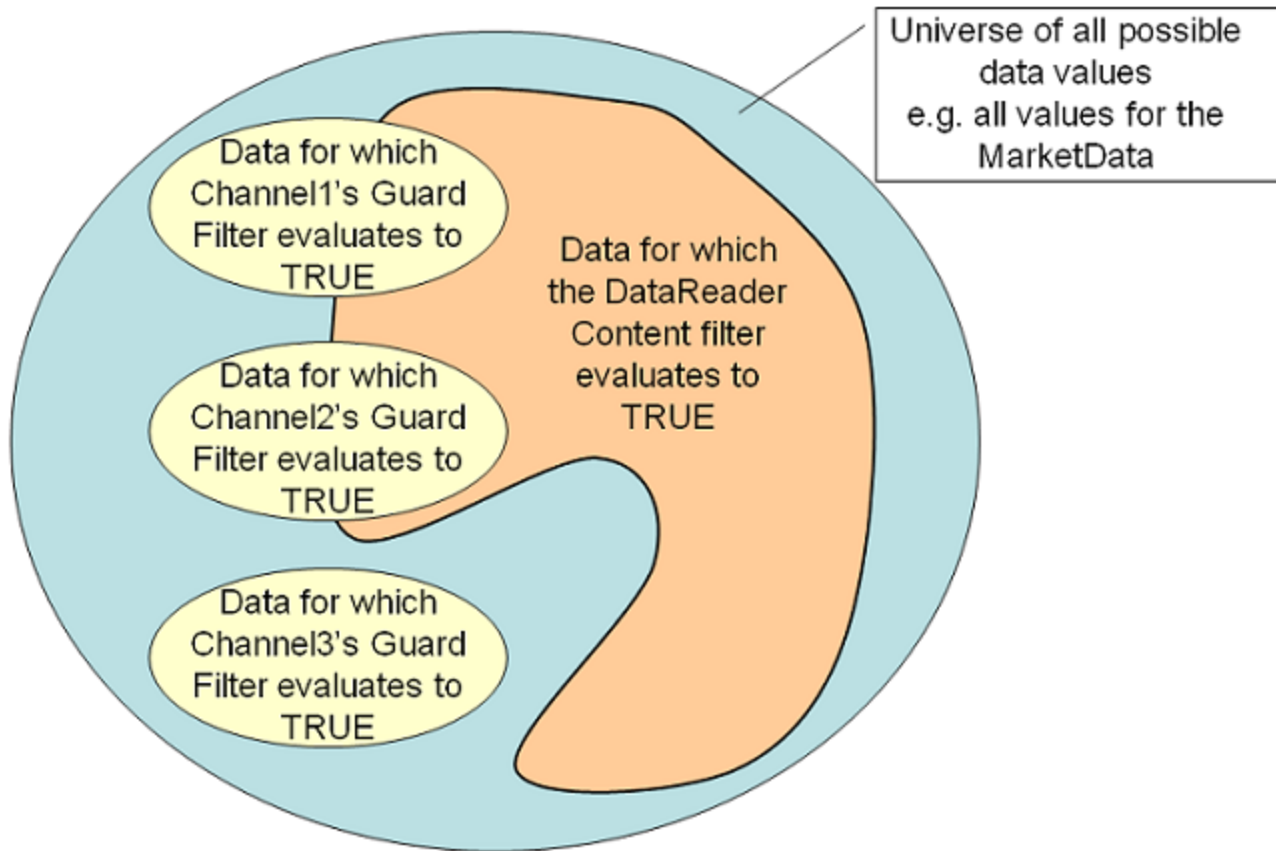
From there, Connex DDS takes care of all the necessary steps:

- The *DataReader* automatically discovers all the *DataWriters*—including multi-channel *DataWriters*—for the Topic it subscribes to.
- When the *DataReader* discovers a multi-channel *DataWriter*, it also discovers the list of channels used by that *DataWriter*.
- When the multi-channel *DataWriter* discovers a *DataReader*, it also discovers the content filters specified by that *DataReader*, if any.

With all this information, Connex DDS automatically determines which channels are of "interest" to the *DataReader*.

A *DataReader* is interested in a channel if and only if the set of data values for which the channel guard filter evaluates to TRUE intersects the set of data values for which the *DataReader's* content filter evaluates to TRUE. If a *DataReader* does not use a content filter, then it is interested in all the channels.

Figure 18.4 Filter Intersection



In this scenario, the DataReader is interested in Channel1 and Channel2, but not Channel3.

Market Data Example, continued:

If the channel guard filter for Channel 1 is 'Symbol MATCH '[A-K]*' then the channel will only transfer data for stocks whose symbol starts with a letter in the A to K range.

That is, it will transfer data on 'APPL', 'GOOG', and 'IBM', but not on 'MSFT', 'ORCL', or 'YHOO'. Channel 1 will be of interest to *DataReaders* whose content filter includes at least one stock whose symbol starts with a letter in the A to K range.

A *DataReader* that specifies a content filter such as "Symbol MATCH 'IBM, YHOO' " will be interested in Channel1.

A *DataReader* that specifies a content filter such as "Symbol MATCH '[G-M]*'" will also be interested in Channel1.

A *DataReader* that specifies a content filter such as "Symbol MATCH '[M-T]*' " will not be interested in Channel1.

18.4 Where Does the Filtering Occur?

If multi-channel *DataWriters* are used, the filtering can occur in three places:

- [Filtering at the DataWriter \(Section 18.4.1 below\)](#)
- [Filtering at the DataReader \(Section 18.4.2 below\)](#)
- [Filtering on the Network Hardware \(Section 18.4.3 on the next page\)](#)

18.4.1 Filtering at the DataWriter

Each time data is written, the *DataWriter* evaluates each of the channel guard filters to determine which channels will transmit the data. This filtering occurs on the *DataWriter*.

Filtering on the *DataWriter* side is scalable because the number of filter evaluations depends only on the number of channels, not on the number of *DataReaders*. Usually, the number of channels is smaller than the number of possible *DataReaders*.

As explained in [Performance Considerations \(Section 18.7 on page 835\)](#), if the channel guard filters are configured to only look at the "key" fields in the data, the channel filtering becomes a very efficient lookup operation.

18.4.2 Filtering at the DataReader

The *DataReader* will listen on the multicast addresses that correspond to the channels of interest (see [Using the MULTI_CHANNEL QosPolicy \(Section Figure 18.3 on page 829\)](#)). When a channel is 'of interest', it means that it is possible for the channel to transmit data that meets the content filter of the *DataReader*, however the channel may also transmit data that does not pass the *DataReader's* content filter. Therefore, the *DataReader* has to filter all incoming data on that channel to determine if it passes its content filter.

Market Data Example, continued:

Channel 1, identified by guard filter "Symbol MATCH '[A-M]*'", will be of interest to *DataReaders* whose content filter includes at least one stock whose symbol starts with a letter in the A to K range.

A *DataReader* with content filter "Symbol MATCH 'GOOG'" will listen on Channel1.

In addition to 'GOOG', the *DataReader* will also receive DDS samples corresponding to stock symbols such as 'MSFT' and 'APPL'. The *DataReader* must filter these DDS samples out.

As explained in [Performance Considerations \(Section 18.7 on page 835\)](#), if the *DataReader's* content filters are configured to only look at the "key" fields in the data, the *DataReader* filtering becomes a very efficient lookup operation.

18.4.3 Filtering on the Network Hardware

DataReaders will only listen to multicast addresses that correspond to the channels of interest. The multicast traffic generated in other channels will be filtered out by the network hardware (routers, switches).

Layer 3 routers will only forward multicast traffic to the actual destination ports. However, by default, layer 2 switches treat multicast traffic as broadcast traffic. To take advantage of network filtering with layer 2 devices, they must be configured with IGMP snooping enabled (see [Network-Switch Filtering \(Section 18.7.1 on page 835\)](#)).

18.5 Fault Tolerance and Redundancy

To achieve fault tolerance and redundancy, configure the *DataWriter's* [MULTI_CHANNEL QosPolicy \(DDS Extension\) \(Section 6.5.14 on page 386\)](#) to publish a DDS sample over multiple channels or over different multicast addresses within a single channel. [Figure 18.5 Using the MULTI_CHANNEL QosPolicy with Overlapping Channels below](#) shows how to use overlapping channels.

If a DDS sample is published to multiple multicast addresses, a *DataReader* may receive multiple copies of the DDS sample. By default, duplicates are discarded by the *DataReader* and not provided to the application. To change this default behavior, use the Durable Reader State property, **dds.data_reader.state.filter_redundant_samples** (see [How To Configure a DataReader for Durable Reader State \(Section 12.4.4 on page 690\)](#)).

Figure 18.5 Using the MULTI_CHANNEL QosPolicy with Overlapping Channels

```
// initialize writer_qos with default values
publisher->get_default_datawriter_qos(writer_qos);
// Initialize MULTI_CHANNEL Qos Policy
// Assign the filter name
// Possible options: DDS_STRINGMATCHFILTER_NAME and DDS_SQLFILTER_NAME
writer_qos.multi_channel.filter_name = (char*) DDS_STRINGMATCHFILTER_NAME;
// Create two channels
writer_qos.multi_channel.channels.ensure_length(2,2);
// First channel
writer_qos.multi_channel.channels[0].filter_expression =
    DDS_String_dup("Symbol MATCH '[A-M]*'");
```

```

writer_qos.multi_channel.channels[0].multicast_settings.ensure_length(2,2);
writer_qos.multi_channel.channels[0].multicast_settings[0].receive_port = 8700;
writer_qos.multi_channel.channels[0].multicast_settings[0].receive_address =
    DDS_String_dup("239.255.1.1");
// Second channel
writer_qos.multi_channel.channels[1].multicast_settings.ensure_length(1,1);
writer_qos.multi_channel.channels[1].multicast_settings[0].receive_port = 8800;
writer_qos.multi_channel.channels[1].multicast_settings[0].receive_address =
    DDS_String_dup("239.255.1.2");
writer_qos.multi_channel.channels[1].filter_expression =
    DDS_String_dup("Symbol MATCH '[C-Z]*'");
// Symbols starting with [C-M] will be published in two different channels
// Create writer
writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);

```

18.6 Reliability with Multi-Channel DataWriters

18.6.1 Reliable Delivery

Reliable delivery is only guaranteed when the **access_scope** in the *Subscriber's* [PRESENTATION QosPolicy \(Section 6.4.6 on page 330\)](#) is set to `DDS_INSTANCE_PRESENTATION_QOS` (default value) and the filters in the *DataWriter's* [MULTI_CHANNEL QosPolicy \(DDS Extension\) \(Section 6.5.14 on page 386\)](#) are keyed-only based.

Market Data Example, continued:

Given the following IDL description for our MarketData topic type:

```

Struct MarketData {
    string<255> Symbol; //@key
    double Price;
}

```

A guard filter "Symbol MATCH 'APPL'" is keyed-only based.

A guard filter "Symbol MATCH 'APPL' and Price < 100" is not keyed-only based.

If any of the guard filters are based on non-key fields, Connex DDS only guarantees reception of the most recent data from the multi-channel *DataWriter*.

18.6.2 Reliable Protocol Considerations

Reliability is maintained on a per-channel basis. Each channel has its own reliability channel send window:

- **low_watermark** and **high_watermark**: The low and high watermarks control the send-window levels (when not using batching, this is a number of DDS samples; when using batching, this is a

number of batches) that determine when to switch between regular and fast heartbeat rates (see [High and Low Watermarks \(Section 6.5.3.1 on page 352\)](#)). With multi-channel *DataWriters*, **high_watermark** and **low_watermark** are computed from the channel with the smaller send-window size and they apply to all the channels. Therefore, because the watermark is determined by the channel with the smallest send-window, periodic heartbeating cannot be controlled on a per-channel basis.

- **heartbeats_per_max_samples**: This field defines the number of piggyback heartbeats per current send-window. For multi-channel *DataWriters*, piggyback heartbeats are sent per channel. The send-window size that is used to calculate the piggyback heartbeat rate is the smallest across all channels..

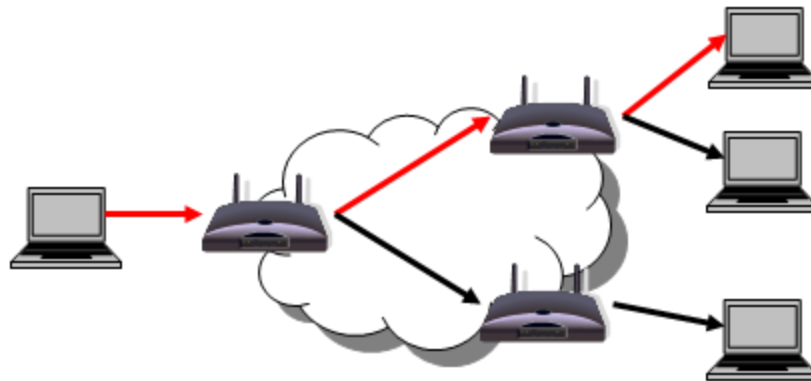
18.7 Performance Considerations

18.7.1 Network-Switch Filtering

By default, multicast traffic is treated as broadcast traffic by layer 2 switches. To avoid flooding the network with broadcast traffic and take full advantage of network filtering, the layer 2 switches should be configured to use IGMP snooping. Refer to your switch's manual for specific instructions.

When IGMP snooping is enabled, a switch can route a multicast packet to just those ports that subscribe to it, as seen in [IGMP Snooping \(Section Figure 18.6 below\)](#).

Figure 18.6 IGMP Snooping



18.7.2 DataWriter and DataReader Filtering

[Where Does the Filtering Occur? \(Section 18.4 on page 832\)](#) describes the three places where filtering can occur with Multi-channel *DataWriters*. To improve performance when filtering occurs on the reader and/or writer sides, use filter expressions that are only based on keys (see [DDS Samples, Instances, and](#)

[Keys \(Section 2.3.1 on page 14\)](#)). Then the results of the filter are cached in a hash table on a per-key basis.

Market Data Example, continued:

The filter expressions in the Market Data example are based on the value of the field, **Symbol**. To make filter operations on this field more efficient, declare **Symbol** as a key. For example:

```
struct {  
    string<MAX_SYMBOL_SIZE> Symbol; //@key  
}
```

You can also improve performance by increasing the number of buckets associated with the hash table. To do so, use the **instance_hash_buckets** field in the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#) on both the writer and reader sides. A higher number of buckets will provide better performance, but requires more resources.

Chapter 19 Connex DDS Threading Model

This chapter describes the internal threads that Connex DDS uses for sending and receiving data, maintaining internal state, and calling user code when events occur such as the arrival of new DDS data samples. It may be important for you to understand how these threads may interact with your application.

A *DomainParticipant* uses three types of threads. The actual number of threads depends on the configuration of various QoS Policies as well as the implementation of the transports used by the *DomainParticipant* to send and receive data.

Through various QoS Policies, the user application can configure the priorities and other properties of the threads created by Connex DDS. In real-time systems, the user often needs to set the priorities of all threads in an application relative to each other for the proper operation of the system.

This chapter includes:

19.1 Database Thread

Connex DDS uses internal data structures to store information about locally-created and remotely-discovered *Entities*. In addition, it will store various objects and data used by Connex DDS for maintaining proper communications between applications. This “database” is created for each *DomainParticipant*.

As *Entities* and objects are created and deleted during the normal operation of the user application, different entries in the database may be created and deleted as well. Because multiple threads may access objects stored in the database simultaneously, the deletion and removal of an object from the database happens in two phases to support thread safety.

When an entry/object in the database is deleted either through the actions of user code or as a result of a change in system state, it is only marked for deletion. It cannot be actually deleted and removed from the database until Connex DDS can be sure that no threads are still accessing the

object. Instead, the actual removal of the object is delegated to an internal thread that Connex DDS spawns to periodically wake up and purge the database of deleted objects.

This thread is known as the Database thread (also referred to as the database cleanup thread).

- Only one Database thread is created for each *DomainParticipant*.

The [DATABASE QosPolicy \(DDS Extension\) \(Section 8.5.1 on page 577\)](#) of the *DomainParticipant* configures both the resources used by the database as well as the properties of the cleanup thread. Specifically, the user may want to use this QosPolicy to set the priority, stack size and thread options of the cleanup thread. You must set these options before the *DomainParticipant* is created, because once the cleanup thread is started as a part of participant creation, these properties cannot be changed.

The period at which the database-cleanup thread wakes up to purge deleted objects is also set in the DATABASE QosPolicy. Typically, this period is set to a long time (on the order of a minute) since there is no need to waste CPU cycles to wake up a thread only to find nothing to do.

However, when a *DomainParticipant* is destroyed, all of the objects created by the *DomainParticipant* will be destroyed as well. Many of these objects are stored in the database, and thus must be destroyed by the cleanup thread. The *DomainParticipant* cannot be destroyed until the database is empty and is destroyed itself. Thus, there is a different parameter in the DATABASE QosPolicy, `shutdown_cleanup_period`, that is used by the database cleanup thread when the *DomainParticipant* is being destroyed. Typically set to be on the order of a second, this parameter reduces the additional time needed to destroy a *DomainParticipant* simply due to waiting for the cleanup thread to wake up and purge the database.

19.2 Event Thread

During operation, Connex DDS must wake up at different intervals to check the condition of many different time-triggered or periodic events. These events are usually to determine if something happened or did not happen within a specified time. Often the condition must be checked periodically as long as the *Entity* for which the condition applies still exists. Also, the *DomainParticipant* may need to do something periodically to maintain connections with remote *Entities*.

For example, the [DEADLINE QosPolicy \(Section 6.5.5 on page 363\)](#) is used to ensure that *DataWriters* have published data or *DataReaders* have received data within a specified time period. Similarly, the [LIVELINESS QosPolicy \(Section 6.5.13 on page 382\)](#) configures Connex DDS both to check periodically to see if a *DataWriter* has sent a liveliness message and to send liveliness messages periodically on the behalf of a *DataWriter*. As a last example, for reliable connections, heartbeats must be sent periodically from the *DataWriter* to the *DataReader* so that the *DataReader* can acknowledge the data that it has received, see [Reliable Communications \(Section Chapter 10 on page 629\)](#).

Connex DDS uses an internal thread, known as the Event thread, to do the following:

- Check whether or not deadlines have been missed
- Invoke user-installed *Listener* callbacks to notify the application of missed deadlines
- Send heartbeats to maintain reliable connections

Note: Only one Event thread is created per *DomainParticipant*.

The [EVENT QosPolicy \(DDS Extension\) \(Section 8.5.5 on page 602\)](#) of the *DomainParticipant* configures both the properties and resources of the Event thread. Specifically, the user may want to use this QosPolicy to set the priority, stack size and thread options of the Event thread. You must set these options before the *DomainParticipant* is created, because once the Event thread is started as a part of participant creation, these properties cannot be changed.

The EVENT QosPolicy also configures the maximum number of events that can be handled by the Event thread. While the Event thread can only service a single event at a time, it must maintain a queue to hold events that are pending. The **initial_count** and **max_count** parameters of the QosPolicy set the initial and maximum size of the queue.

The priority of the Event thread should be carefully set with respect to the priorities of the other threads in a system. While many events can tolerate some amount of latency between the time that the event expires and the time that the Event thread services the event, there may be application-specific events that must be handled as soon as possible.

For example, if an application uses the liveliness of a remote *DataWriter* to infer the correct operation of a remote application, it may be critical for the user code in the *DataReader Listener* callback, `on_liveliness_changed()`, to be called by the Event thread as soon as it can be determined that the remote application has died. The operating system uses the priority of the Event thread to schedule this action.

19.3 Receive Threads

Connex DDS uses internal threads, known as Receive threads, to process the data packets received via underlying network transports. These data packets may contain meta-traffic exchanged by *DomainParticipants* for discovery, or user data (and meta-data to support reliable connections) destined for local *DataReaders*.

As a result of processing packets received by a transport, a Receive thread may respond by sending packets on the network. Discovery packets may be sent to other *DomainParticipants* in response to ones received. ACK/NACK packets are sent in response to heartbeats to support a reliable connection.

When a DDS sample arrives, the Receive thread is responsible for deserializing and storing the data in the receive queue of a *DataReader* as well as invoking the `on_data_available()` *DataReaderListener* callback (see [Setting Up DataReaderListeners \(Section 7.3.4 on page 466\)](#)).

The number of Receive threads that Connex DDS will create for a *DomainParticipant* depends on how you have configured the QosPolicies of *DomainParticipants*, *DataWriters* and *DataReaders* as well as on the implementation of a particular transport. The behavior of the builtin transports is well specified.

However, if a custom transport is installed for a *DomainParticipant*, you will have to understand how the custom transport works to predict how many Receive threads will be created.

The following discussion applies on a per-transport basis. A single Receive thread will only service a single transport.

Connex DDS will try to create receive resources¹ for every port of every transport on which it is configured to receive messages. The [TRANSPORT_UNICAST QosPolicy \(DDS Extension\) \(Section 6.5.24 on page 412\)](#) for *DomainParticipant*, *DataWriters*, and *DataReaders*, the [TRANSPORT_MULTICAST QosPolicy \(DDS Extension\) \(Section 7.6.5 on page 529\)](#) for *DataReaders* and the [DISCOVERY QosPolicy \(DDS Extension\) \(Section 8.5.2 on page 580\)](#) for *DomainParticipants* all configure the number of ports and the number of transports that Connex DDS will try to use for receiving messages.

Generally, transports will require Connex DDS to create a new receive resource for every unique port number. However, this is both dependent on how the underlying physical transport works and the implementation of the transport plug-in used by Connex DDS. Sometimes Connex DDS only needs to create a single receive resource for any number of ports.

When Connex DDS finds that it is configured to receive data on a port for a transport for which it has not already created a receive resource, it will ask the transport if any of the existing receive resources created for the transport can be shared. If so, then Connex DDS will not have to create a new receive resource. If not, then Connex DDS will.

The [TRANSPORT_UNICAST](#), [TRANSPORT_MULTICAST](#), and [DISCOVERY](#) QosPolicies allow you customize ports for receiving user data (on a per-*DataReader* basis) and meta-traffic (*DataWriters* and *DomainParticipants*); ports can be also set differently for unicast and multicast.

How do receive resources relate to Receive threads? Connex DDS will create a Receive thread to service every receive resource that is created. If you use a socket analogy, then for every socket created, Connex DDS will use a separate thread to process the data received on that socket.

So how many thread will Connex DDS create by default—using only the builtin UDPv4 and shared memory transports and without modifying any QosPolicies?

Three Receive threads are created for meta-traffic²:

¹If UDPv4 was the only transport that Connex DDS supports, we would call these receive resources ‘sockets.’

²Meta-traffic refers to traffic internal to Connex DDS related to dynamic discovery (see [Discovery \(Section Chapter 14 on page 709\)](#)).

- 2 for unicast (one for UDPv4, one for shared memory)
- 1 for multicast (for UDPv4)¹

Two Receive threads created for user data:

- 2 for unicast (UDPv4, shared memory)
- 0 for multicast (because user data is not sent via multicast by default)

Therefore, by default, you will have a total of five Receive threads per *DomainParticipant*. By using only a single transport and disabling multicast, a *DomainParticipant* can have as few as 2 Receive threads.

Similar to the Database and Event threads, a Receive thread is configured by the [RECEIVER_POOL QosPolicy \(DDS Extension\) \(Section 8.5.6 on page 604\)](#). However, note that the thread properties in the `RECEIVER_POOL` QosPolicy apply to all Receive threads created for the *DomainParticipant*.

19.4 Exclusive Areas, Connex DDS Threads and User Listeners

Connex DDS Event and Receive threads may invoke user code through the *Listener* callbacks installed on different *Entities* while executing internal Connex DDS code. In turn, user code inside the callbacks may invoke Connex DDS APIs that reenter the internal code space of Connex DDS. For thread safety, Connex DDS allocates and uses mutual exclusion semaphores (mutexes).

As discussed in [Exclusive Areas \(EAs\) \(Section 4.5 on page 182\)](#), when multiple threads and multiple mutexes are mixed together, deadlock may result. To prevent deadlock from occurring, Connex DDS is designed using careful analysis and following rules that force mutexes to be taken in a certain order when a thread must take multiple mutexes simultaneously.

However, because the Event and Receive threads already hold mutexes when invoking user callbacks, and because the Connex DDS APIs that the user code can invoke may try to take other mutexes, deadlock may still result. Thus, to prevent user code to cause internal Connex DDS threads to deadlock, we have created a concept called Exclusive Areas (EA) that follow rules that prevent deadlock. The more EAs that exist in a system, the more concurrency is allowed through Connex DDS code. However, the more EAs that exist, the more restrictions on the Connex DDS APIs that are allowed to be invoked in *Entity Listener* callbacks.

The [EXCLUSIVE_AREA QosPolicy \(DDS Extension\) \(Section 6.4.3 on page 318\)](#) control how many EAs will be created by Connex DDS. For a more detailed discussion on EAs and the restrictions on the use of Connex DDS APIs within *Entity Listener* methods, please see [Exclusive Areas \(EAs\) \(Section 4.5 on page 182\)](#).

¹Multicast is not supported by shared memory transports.

19.5 Controlling CPU Core Affinity for RTI Threads

Two fields in the `DDS_ThreadSettings_t` structure (see [Thread Settings \(Section 17.4.7 on page 809\)](#)) are related to CPU core affinity: `cpu_list` and `cpu_rotation`.

Note: Although `DDS_ThreadSettings_t` is used in the Event, Database, ReceiverPool, and AsynchronousPublisher QoS policies, `cpu_list` and `cpu_rotation` are only relevant in the [RECEIVER_POOL QoS Policy \(DDS Extension\) \(Section 8.5.6 on page 604\)](#).

While most thread-related QoS settings apply to a single thread, the ReceiverPool QoS policy's thread-settings control *every* receive thread created. In this case, there are several schemes to map M threads to N processors; `cpu_rotation` controls which scheme is used.

The `cpu_rotation` determines how `cpu_list` affects processor affinity for thread-related QoS policies that apply to multiple threads. If `cpu_list` is empty, `cpu_rotation` is irrelevant since no affinity adjustment will occur. Suppose instead that `cpu_list` = {0,1} and that the middleware creates three receive threads: {A, B, C}. If `cpu_rotation` is set to `CPU_NO_ROTATION`, threads A, B and C will have the same processor affinities (0-1), and the OS will control thread scheduling within this bound.

CPU affinities are commonly denoted with a bitmask, where set bits represent allowed processors to run on. This mask is printed in hex, so a CPU affinity of 0-1 can be represented by the mask 0x3.

If `cpu_rotation` is `CPU_RR_ROTATION`, each thread will be assigned in round-robin fashion to one of the processors in `cpu_list`; perhaps thread A to 0, B to 1, and C to 0. Note that the order in which internal middleware threads spawn is unspecified.

The [RTI Connex DDS Core Libraries Platform Notes](#) describe which architectures support this feature.

19.6 Configuring Thread Settings with XML

[Table 19.1 XML Tags for ThreadSettings_t](#) describes the XML tags that you can use to configure thread settings. For more information on thread settings, see:

- [Thread Settings \(Section 17.4.7 on page 809\)](#)
- The [RTI Connex DDS Core Libraries Platform Notes](#)
- The API Reference HTML documentation (select **Modules**, **RTI Connex DDS API Reference**, **Infrastructure Module**, **QoS Policies**, **Extended QoS Support**, **Thread Settings**)

Table 19.1 XML Tags for ThreadSettings_t

Tags within <thread>	Description	Number of Tags Allowed
<cpu_list>	<p>Each <element> specifies a processor on which the thread may run.</p> <pre data-bbox="315 485 651 569"><cpu_list> <element>value</element> </cpu_list></pre> <p>Only applies to platforms that support controlling CPU core affinity (see Controlling CPU Core Affinity for RTI Threads (Section 19.5 on the previous page) and the RTI Connex DDS Core Libraries Platform Notes).</p>	0 or 1
<cpu_rotation>	<p>Determines how the CPUs in <cpu_list> will be used by the thread. The value can be either:</p> <p>THREAD_SETTINGS_CPU_NO_ROTATION</p> <p>The thread can run on any listed processor, as determined by OS scheduling.</p> <p>THREAD_SETTINGS_CPU_RR_ROTATION</p> <p>The thread will be assigned a CPU from the list in round-robin order.</p> <p>Only applies to platforms that support controlling CPU core affinity (see the RTI Connex DDS Core Libraries Platform Notes).</p>	0 or 1
<mask>	<p>A collection of flags used to configure threads of execution. Not all of these options may be relevant for all operating systems. May include these bits:</p> <ul data-bbox="342 1140 643 1310" style="list-style-type: none"> • STDIO • FLOATING_POINT • REALTIME_PRIORITY • PRIORITY_ENFORCE <p>It can also be set to a combination of the above bits by using the “or” symbol (), such as STDIO FLOATING_POINT.</p> <p>Default: MASK_DEFAULT</p>	0 or 1

Table 19.1 XML Tags for ThreadSettings_t

Tags within <thread>	Description	Number of Tags Allowed
<priority>	<p>Thread priority. The value can be specified as an unsigned integer or one of the following strings.</p> <ul style="list-style-type: none"> • THREAD_PRIORITY_DEFAULT • THREAD_PRIORITY_HIGH • THREAD_PRIORITY_ABOVE_NORMAL • THREAD_PRIORITY_NORMAL • THREAD_PRIORITY_BELOW_NORMAL • THREAD_PRIORITY_LOW <p>When using an unsigned integer, the allowed range is platform-dependent.</p> <p>When thread priorities are configured using XML, the values are considered native priorities.</p> <p>Example:</p> <pre><thread> <mask>STDIO FLOATING_POINT</mask> <priority>10</priority> <stack_size>THREAD_STACK_SIZE_DEFAULT</stack_size> </thread></pre> <p>When the XML file is loaded using the Java API, the priority is a native priority, not a Java thread priority.</p>	0 or 1
<stack_size>	Thread stack size, specified as an unsigned integer or set to the string THREAD_STACK_SIZE_DEFAULT. The allowed range is platform-dependent.	0 or 1

19.7 User-Managed Threads

In certain scenarios, you may want full control over the internal threads created by your Connex DDS applications. For instance, in memory-constrained systems, applications may want to manage the resources required by internal Connex DDS threads. Also, you may want to use a different thread technology than the one Connex DDS incorporates by default (i.e., pthread on POSIX platforms).

Connex DDS can create the internal threads from the application layer via the abstract factory pattern. You can provide a Connex DDS application with a **ThreadFactory** implementation that *DomainParticipants* will use to create and delete all the threads.

The **ThreadFactory** interface exposes operations for creating and deleting threads. These operations are called on demand as *DomainParticipants* require new threads or need to delete existing ones.

The same **ThreadFactory** instance can be used by multiple *DomainParticipants*. To select which ThreadFactory to use, use the `set_thread_factory()` operation in the *DomainParticipantFactory*:

```
MyThreadFactory myThreadFactory; // Implements DDSThreadFactory
retcode = DDSTheParticipantFactory->set_thread_factory(&myThreadFactory);
```

Then you can create *DomainParticipants* using any of the available APIs (i.e. **create_participant()**, **create_participant_from_config()**, etc). A *DomainParticipant* will use the **ThreadFactory** object that is set in the *DomainParticipantFactory* at the time it is created and throughout its entire lifecycle. If a new *ThreadFactory* is set, existing *DomainParticipants* will not be affected; they will still use the same *ThreadFactory* with which they were created.

This feature is only available for the C/C++ APIs. For further information, please see the API Reference HTML documentation.

Chapter 20 DDS Sample-Data and Instance-Data Memory Management

This chapter describes how Connex DDS manages the memory for the DDS data samples that are sent by *DataWriters* and received by *DataReaders*.

20.1 DDS Sample-Data Memory Management for DataWriters

To configure DDS sample-data memory management on the writer side, use the [PROPERTY QosPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#). [Table 20.1 DDS Sample-Data Memory Management Properties for DataWriters](#) lists the supported memory-management properties for *DataWriters*.

Table 20.1 DDS Sample-Data Memory Management Properties for DataWriters

Property	Description
dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size	<p>If the serialized size of the DDS sample is \leq pool_buffer_max_size: The buffer is obtained from a pre-allocated pool and released when the <i>DataWriter</i> is deleted.</p> <p>If the serialized size of the DDS sample is $>$ pool_buffer_max_size: The buffer is dynamically allocated from the heap and returned to the heap when the DDS sample is removed from the <i>DataWriter</i>'s queue.</p> <p>Default: -1 (UNLIMITED). All DDS sample buffers are obtained from the pre-allocated pool; the buffer size is the maximum serialized size of the DDS samples, as returned by the type plugin get_serialized_sample_max_size() operation.</p> <p>See Memory Management without Batching (Section 20.1.1 on the next page).</p> <p>Note: This property also controls the memory allocation for the serialized key buffer that is stored with every instance. See Instance-Data Memory Management for DataWriters (Section 20.3 on page 861).</p>

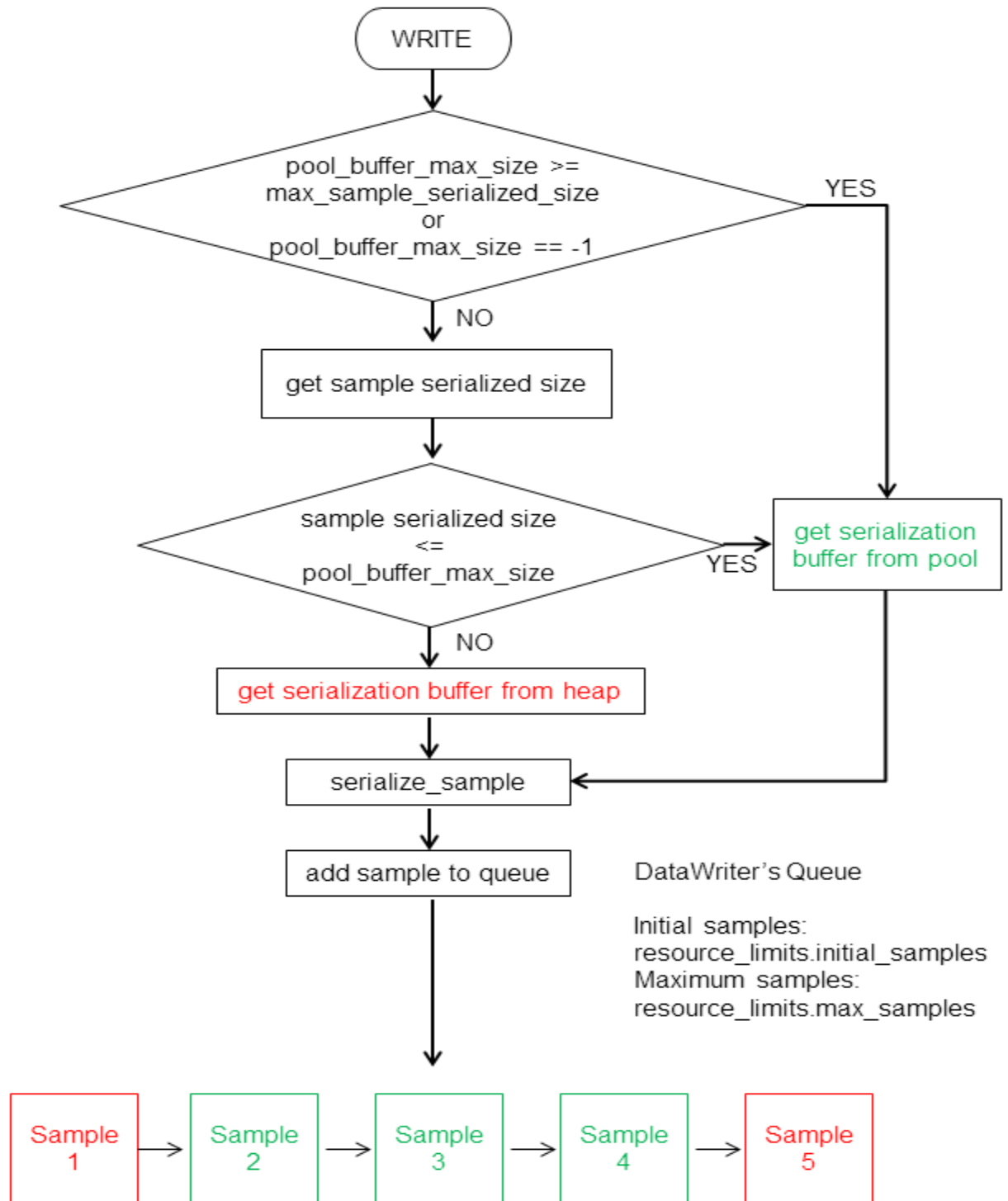
Table 20.1 DDS Sample-Data Memory Management Properties for DataWriters

Property	Description
dds.data_writer. history.memory_ manager. java_ stream.min_size	<p>Only supported when using the Java API.</p> <p>Defines the minimum size of the buffer that will be used to serialize DDS samples.</p> <p>When a <i>DataWriter</i> is created, the Java layer will allocate a buffer of this size and associate it with the <i>DataWriter</i>.</p> <p>Default: -1 (UNLIMITED). This is a sentinel that refers to the maximum serialized size of a DDS sample, as returned by the type plugin <code>get_serialized_sample_max_size()</code> operation</p> <p>See Writer-Side Memory Management when Using Java (Section 20.1.3 on page 851).</p>
dds.data_writer. history.memory_ manager. java_ stream.trim_to_ size	<p>Only supported when using the Java API.</p> <p>A boolean value that controls the growth of the serialization buffer.</p> <p>If set to 0 (default): The buffer will not be reallocated unless the serialized size of a new DDS sample is greater than the current buffer size.</p> <p>If set to 1: The buffer will be reallocated with each new DDS sample to a smaller size in order to just fit the DDS sample serialized size. The new size cannot be smaller than <code>min_size</code>.</p> <p>See Writer-Side Memory Management when Using Java (Section 20.1.3 on page 851).</p>

20.1.1 Memory Management without Batching

When the `write()` operation is called on a *DataWriter* that does not have batching enabled, the *DataWriter* serializes (marshals) the input DDS sample and stores it in the *DataWriter*'s queue (see [Figure 20.1 DataWriter Actions when Batching is Disabled on the facing page](#)). The size of this queue is limited by `initial_samples/max_samples` in the `RESOURCE_LIMITS` QosPolicy (Section 6.5.20 on page 405).

Figure 20.1 DataWriter Actions when Batching is Disabled



Each DDS sample in the queue has an associated serialization buffer in which the *DataWriter* will serialize the DDS sample. This buffer is either obtained from a pre-allocated pool (if the serialized size of the DDS sample is \leq `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`) or the buffer is dynamically allocated from the heap (if the serialized size of the DDS sample is $>$ `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`). See [Table 20.1 DDS Sample-Data Memory Management Properties for DataWriters](#),

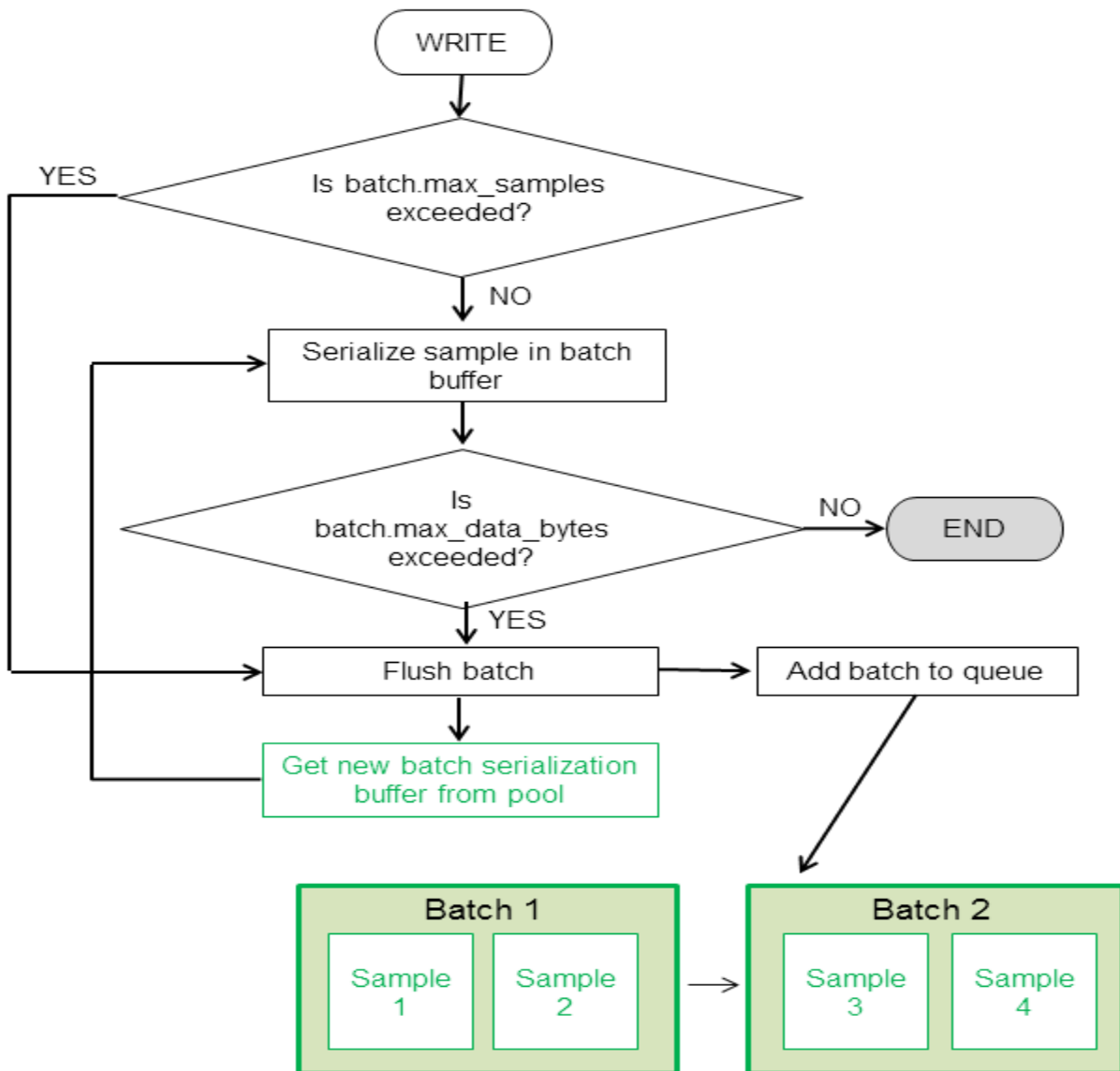
The default value of `pool_buffer_max_size` is -1 (UNLIMITED). In this case, all the DDS samples come from the pre-allocated pool and the size of the buffers is the maximum serialized size of the DDS samples as returned by the type plugin `get_serialized_sample_max_size()` operation. The default value is optimum for real-time applications where determinism and predictability is a must. The trade-off is higher memory usage, especially in cases where the maximum serialized size of a DDS sample is large.

20.1.2 Memory Management with Batching

When the `write()` operation is called on a *DataWriter* for which batching is enabled (see [BATCH QosPolicy \(DDS Extension\) \(Section 6.5.2 on page 341\)](#)), the *DataWriter* serializes (marshals) the input DDS sample into the current batch buffer (see [Figure 20.2 DataWriter Actions when Batching is Enabled on the facing page](#)). When the batch is flushed, it is stored in the *DataWriter*'s queue along with its DDS samples. The *DataWriter* queue can be sized based on:

- The number of DDS samples, using `initial_samples/max_samples` (both set in the [RESOURCE_LIMITS QosPolicy \(Section 6.5.20 on page 405\)](#))
- The number of batches, using `initial_batches/max_batches` (both set in the [DATA_WRITER_RESOURCE_LIMITS QosPolicy \(DDS Extension\) \(Section 6.5.4 on page 359\)](#))
- Or a combination of `max_samples` and `max_batches`

Figure 20.2 DataWriter Actions when Batching is Enabled



DataWriter's Queue

Initial samples:
 resource_limits.initial_samples
 Maximum samples:
 resource_limits.max_samples
 Initial batches:
 writer_resource_limits.initial_batches
 Maximum batches:
 writer_resource_limits.max_batches

When batching is enabled, the memory associated with the batch buffers always comes from a pre-allocated pool. The size of the buffers is determined by the QoS values **max_samples** and **max_data_bytes** (both set in the [BATCH QosPolicy \(DDS Extension\)](#) (Section 6.5.2 on page 341)) as follows:

- If **max_data_bytes** is a finite value, the size of the buffer is the minimum of this value and the maximum serialized size of a DDS sample (**max_sample_serialized_size**) as returned by the type-plugin **get_serialized_sample_max_size()**, since that batch must contain at least one DDS sample).
- Otherwise, the size of the buffer is calculated by **(batch.max_samples * max_sample_serialized_size)**.

Notice that for variable-size DDS samples (for example, DDS samples containing sequences) it is good practice to size the buffer based on **max_data_bytes**, since this leads to more efficient memory usage.

Note: The value of the property **dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size** is ignored by *DataWriters* with batching enabled.

20.1.3 Writer-Side Memory Management when Using Java

When the Java API is used, Connex DDS allocates a Java buffer per *DataWriter*; this buffer is used to serialize the Java DDS samples published by the *DataWriters*. After a DDS sample is serialized into a Java buffer, the result is copied into the underlying native buffer described in [Memory Management without Batching](#) (Section 20.1.1 on page 847) and [Memory Management with Batching](#) (Section 20.1.2 on page 849).

You can use the following two *DataWriter* properties to control memory allocation for the Java buffers that are used for serialization (see [Table 20.1 DDS Sample-Data Memory Management Properties for DataWriters](#)):

- **dds.data_writer.history.memory_manager.java_stream.min_size**
- **dds.data_writer.history.memory_manager.java_stream.trim_to_size**

20.1.4 Writer-Side Memory Management when Working with Large Data

Large DDS samples are DDS samples with a large *maximum* size relative to the memory available to the application. Notice the use of the word *maximum*, as opposed to *actual* size.

As described in [Memory Management without Batching](#) (Section 20.1.1 on page 847), by default, the middleware preallocates the DDS samples in the *DataWriter* queue to their maximum serialized size. This may lead to high memory-usage in *DataWriters* where the maximum serialized size of a DDS sample is large.

For example, let's consider a video conferencing application:

```
struct VideoFrame {
    boolean keyFrame;
```



```
sequence<octet,1024000> data;
};
```

The above IDL definition can be used to work with video streams.

Each frame is transmitted as a sequence of octets with a maximum size of 1 MB. In this example, the video stream has two types of frames: I-Frames (also called key frames) and P-Frames (also called delta frames). I-Frames represent full images and do not require information about the preceding frames in order to be decoded. P-frames require information about the preceding frames in order to be decoded.

A video stream consists of a sequence of frames in which I-Frames are followed by multiple P-frames. The number of P-frames between I-Frames affects the video quality since, in a non-reliable configuration, losing a P-frame will degrade the image quality until the next I-frame is received.

For our use case, let's assume that I-frames may require 1 MB, while P-Frames require less than 32 KB. Also, there are 20 times more P-Frames than I-Frames.

Although the actual size of the frames sent by the Connex DDS application is usually significantly smaller than 1 MB since they are P-Frames, the default memory management will use 1 MB per frame in the *DataWriter* queue. If `resource_limits.max_samples` is 256, the *DataWriter* may end up allocating 256 MB.

Using some domain-specific knowledge, such as the fact that most of the P-Frames have a size smaller than 32 KB, we can optimize memory usage in the *DataWriter's* queue while still maintaining determinism and predictability for the majority of the frames sent on the wire.

The following XML file shows how to optimize the memory usage for the previous example (rather than focusing on efficient usage of the available network bandwidth).

```
<?xml version="1.0"?>
<!-- XML QoS Profile for large data -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- QoS Library containing the QoS profile used for large data -->
  <qos_library name="ReliableLargeDataLibrary">
    <!-- QoS profile to optimize memory usage in DataWriters sending
    large images

    -->
    <qos_profile name="ReliableLargeDataProfile"
    is_default_qos="true">
      <!-- QoS used to configure the DataWriter -->
      <datawriter_qos>
        <resource_limits>
          <max_samples>32</max_samples>
          <!-- No need to pre-allocate 32 images unless
          needed -->
          <initial_samples>1</initial_samples>
        </resource_limits>
        <property>
          <value>
            <!-- For frames with size smaller or
```

```

equal to 33 KB
the serialization buffer is
obtained from a
pre-allocated pool. For sizes
greater than 33 KB,
the DataWriter will use dynamic
memory allocation.
-->
<element>
  <name>
dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size
  </name>
  <value>33792</value>
</element>
<!-- Java will use a 33 KB buffer to
serialize all frames with a
size smaller than or equal to
33 KB.
When an I-frame is published,
Java will reallocate the
serialization buffer to
match the serialized
size of the new frame.

-->
<element>
  <name>
dds.data_writer.history.memory_manager.java_stream.min_size
  </name>
  <value>33792</value>
</element>
<element>
  <name>
dds.data_writer.history.memory_manager.java_stream.trim_to_size
  </name>
  <value>1</value>
</element>
</value>
</property>
</datawriter_qos>
</qos_profile>
</qos_library>
</dds>

```

Working with large data DDS samples will likely require throttling the network traffic generated by single DDS samples. For additional information on shaping network traffic, see [FlowControllers \(DDS Extension\) \(Section 6.6 on page 422\)](#).

20.2 DDS Sample-Data Memory Management for DataReaders

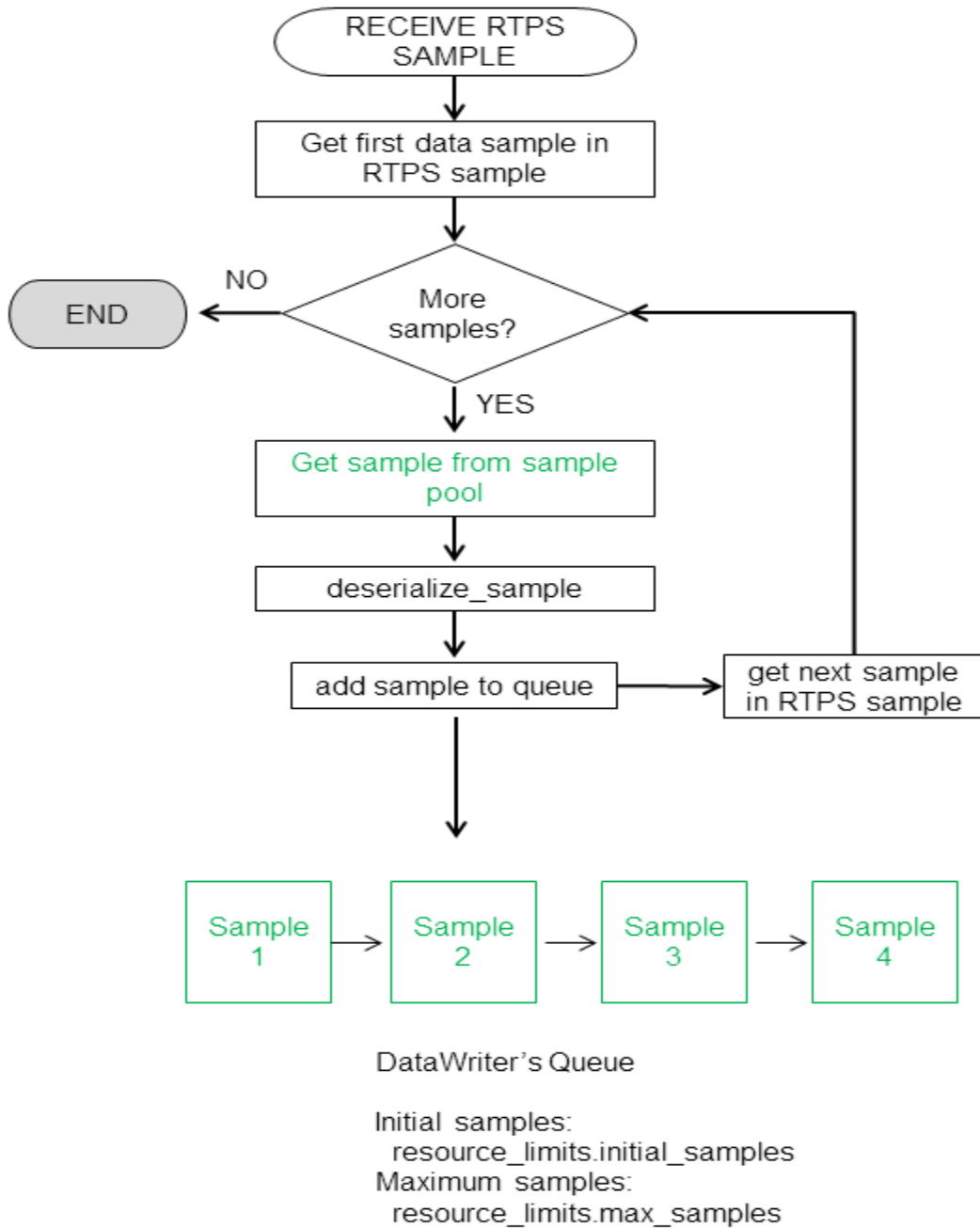
The DDS data samples received by a *DataReader* are deserialized (demarshaled) and stored in the *DataReader's* queue (see [Adding DDS Samples to DataReader's Queue \(Section Figure 20.3 on page](#)

855)). The size of this queue is limited by **initial_samples/max_samples** in the [RESOURCE_LIMITS QosPolicy](#) (Section 6.5.20 on page 405).

20.2.1 Memory Management for DataReaders Using Generated Type-Plugins

[Figure 20.3 Adding DDS Samples to DataReader's Queue on the next page](#) shows how DDS samples are processed and added to the *DataReader's* queue.

Figure 20.3 Adding DDS Samples to DataReader’s Queue



The RTPS DATA DDS samples received by a *DataReader* can be either batch DDS samples or individual DDS samples. The *DataReader* queue does not store batches. Therefore, each one of the DDS samples within a batch will be deserialized and processed individually.

When the *DataReader* processes a new sample, it will deserialize it into a sample obtained from a pre-allocated pool. By default, to provide predictability and determinism, the sample obtained from the pool is

allocated to its maximum size. For example, with the following IDL type, each sample in the *DataReader* queue will consume 1 MB, even if the actual size is smaller.

```
struct VideoFrame {
    boolean keyFrame;
    sequence<octet,1024000> data;
};
```

In the above example, it is possible to reduce the memory consumption in C, C++, and .NET by declaring the data sequence as unbounded and by generating code for the type with the command-line option - **unboundedSupport**. In this case, the middleware will not preallocate 1 MB for the data member. Instead, the generated code will deserialize incoming samples by dynamically allocating and deallocating memory to accommodate the actual size of the data sequence.

20.2.2 Reader-Side Memory Management when Using Java

When the Java API is used with *DataReaders* using generated type-plugins, Connex DDS allocates a Java buffer per *DataReader*; this buffer is used to copy the native serialized data, so that the received DDS samples can be deserialized into the Java objects obtained from the DDS sample pool in [Adding DDS Samples to DataReader's Queue \(Section Figure 20.3 on the previous page\)](#).

You can use the *DataReader* properties in [Table 20.2 DDS Sample-Data Memory Management Properties for DataReaders when Using Java API](#) to control memory allocation for the Java buffer used for deserialization:

Table 20.2 DDS Sample-Data Memory Management Properties for DataReaders when Using Java API

Property	Description
dds.data_reader.history_memory_manager.java_stream.min_size	<p>Only supported when using the Java API.</p> <p>Defines the minimum size of the buffer used for the serialized data.</p> <p>When a <i>DataReader</i> is created, the Java layer will allocate a buffer of this size and associate it with the <i>DataReader</i>.</p> <p>Default: -1 (UNLIMITED) This is a sentinel to refer to the maximum serialized size of a DDS sample, as returned by the type plugin method <code>get_serialized_sample_max_size()</code>.</p>
dds.data_reader.history_memory_manager.java_stream.trim_to_size	<p>Only supported when using the Java API.</p> <p>A Boolean value that controls the growth of the deserialization buffer.</p> <p>If set to 0 (the default), the buffer will not be re-allocated unless the serialized size of a new DDS sample is greater than the current buffer size.</p> <p>If set to 1, the buffer will be re-allocated with each new DDS sample in order to just fit the DDS sample serialized size. The new size cannot be smaller than <code>min_size</code>.</p>

20.2.3 Memory Management for DynamicData DataReaders

Unlike *DataReaders* that use generated type-plugin code, DynamicData *DataReaders* provide configuration mechanisms to optimize the memory usage for use cases involving large data DDS samples.

A DDS DynamicData sample stored in the *DataReader*'s queue has an associated underlying buffer that contains the serialized representation of the DDS sample. The buffer is allocated according to the configuration provided in the **serialization** member of the **DynamicDataProperty_t** used to create the **DynamicDataSupport** (see [Interacting Dynamically with User Data Types \(Section 3.8 on page 141\)](#)).

```
struct DDS_DynamicDataProperty_t {
    ...
    DDS_DynamicDataTypeSerializationProperty_t serialization;
}
struct DDS_DynamicDataTypeSerializationProperty_t {
    ...
    DDS_UnsignedLong max_size_serialized;
    DDS_UnsignedLong min_size_serialized;
    DDS_Boolean trim_to_size;
}
```

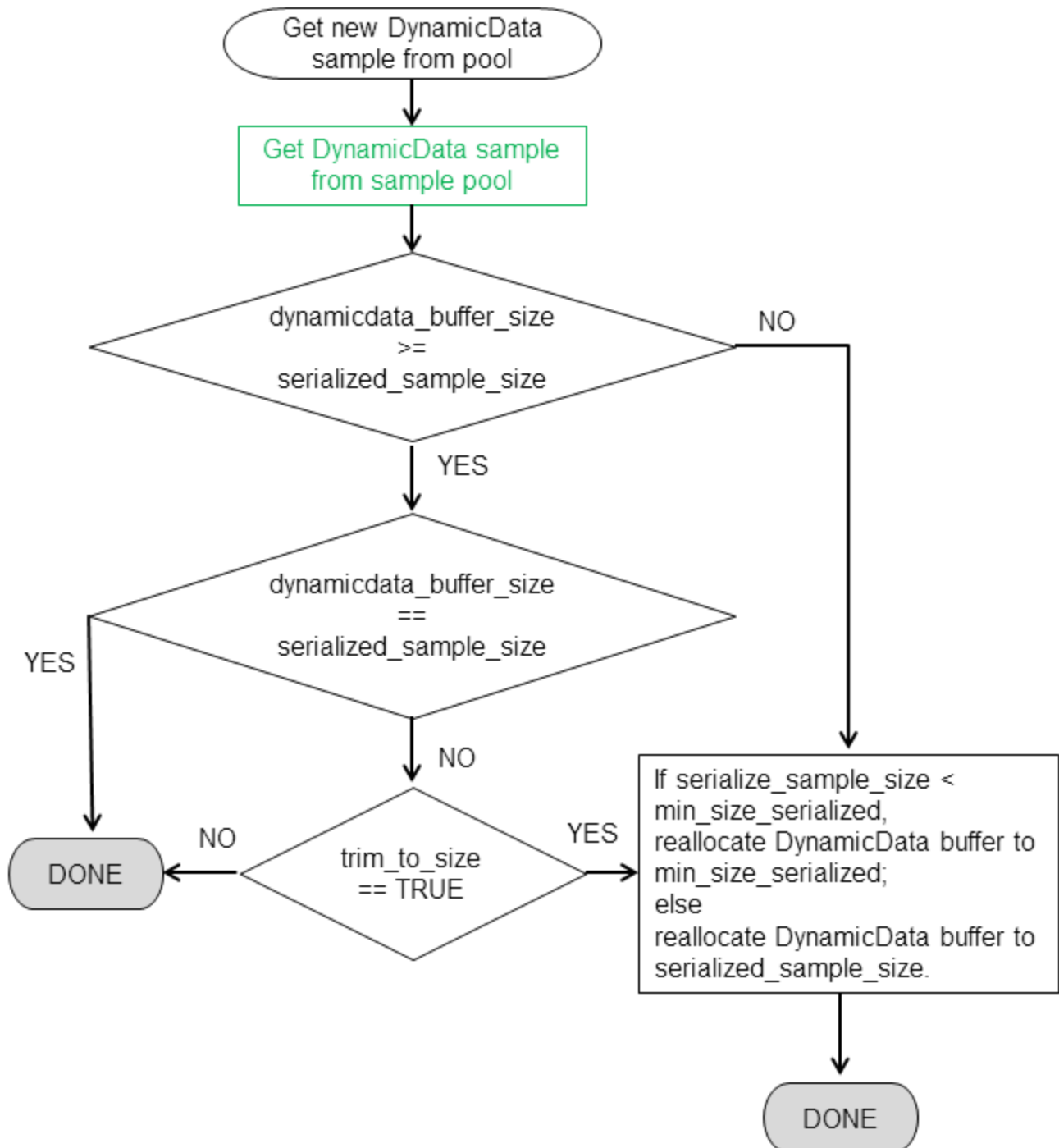
[Table 20.3 struct DDS_DynamicDataTypeSerializationProperty_t](#) describes the members of DDS_DynamicDataTypeSerializationProperty_t.

Table 20.3 struct DDS_DynamicDataTypeSerializationProperty_t

Name	Description
max_size_serialized	Defines the maximum size of the buffer that will contain the serialized DDS sample. Default: 0xFFFFFFFF, indicates that Connex DDS must use the maximum serialized size of a DDS sample according to the type information. Except in very specific scenarios, the value max_size_serialized should always be the default.
min_size_serialized	Defines the minimum size of the buffer used to hold the serialized data in a DynamicData object. Default: 0xFFFFFFFF, a sentinel that indicates that this value must be equal to the value specified in max_size_serialized .
trim_to_size	Controls the growth of the serialization buffer in a DynamicData object. If set to 0 (default): The buffer will not be reallocated unless the serialized size of the incoming DDS sample is greater than the current buffer size. If set to 1: The buffer of a DynamicData object obtained from the DDS sample pool will be re-allocated to just fit the size of the serialized data of the incoming sample. The new size cannot be smaller than min_size_serialized .

[Figure 20.4 Allocation of DDS Samples in DataReader Queue for DynamicData DataReaders on the facing page](#) shows how DDS samples are allocated in the *DataReader* queue for DynamicData *DataReaders*.

Figure 20.4 Allocation of DDS Samples in DataReader Queue for DynamicData DataReaders



20.2.4 Memory Management for Fragmented DDS Samples

When a *DataWriter* writes DDS samples with a serialized size greater than the minimum of the largest transport message sizes across all transports installed with the *DataWriter*, the DDS samples are fragmented into multiple RTPS fragment messages.

The different fragments associated with a DDS sample are assembled in the *DataReader* side into a single buffer that will contain the DDS sample serialized data after the last fragment is received.

By default, the *DataReader* keeps a pool of pre-allocated serialization buffers that will be used to reconstruct the serialized data of a DDS sample from the different fragments. Each buffer hold one individual DDS sample and it has a size equal to the maximum serialized size of a DDS sample. The pool size can be configured using the QoS values **initial_fragmented_samples** and **max_fragmented_samples** in [DATA_READER_RESOURCE_LIMITS QoS Policy \(DDS Extension\)](#) (Section 7.6.2 on page 517).

The main disadvantage in pre-allocating the serialization buffers is an increase in memory usage, especially when the maximum serialized of a DDS sample is quite large. Connex DDS offers a setting that allows memory for a DDS sample to be allocated from the heap the first time a fragment is received. The amount of memory allocated equals the amount of memory needed to store all fragments in the DDS sample.

20.2.5 Reader-Side Memory Management when Working with Large Data

This section describes how to configure the *DataReader* side of the videoconferencing application introduced in [Writer-Side Memory Management when Working with Large Data](#) (Section 20.1.4 on page 851) to optimize memory usage.

The following XML file can be used to optimize the memory usage in the previous example:

```
<?xml version="1.0"?>
<!-- XML QoS Profile for large data -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- QoS Library containing the QoS profile used for large data -->
  <qos_library name="ReliableLargeDataLibrary">
    <!-- QoS profile used to optimize the memory usage in a
    DataWriter sending large data images
    -->
    <qos_profile name="ReliableLargeDataProfile"
    is_default_qos="true">
      <!-- QoS used to configure the DataWriter -->
      <datareader_qos>
        <history>
          <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <resource_limits>
          <max_samples>32</max_samples>
          <!-- No need to pre-allocate 32 frames unless
          needed -->
          <initial_samples>1</initial_samples>
        </resource_limits>
        <reader_resource_limits>
```



```

        <!-- Since the video frame samples have a
             large maximum serialized size we can configure
             the fragmented samples pool to use dynamic
             memory allocation. As an alternative,
             reduce max_fragmented_samples. However, that
             may cause fragment retransmission.
        -->
        <dynamically_allocate_fragmented_samples>
            1
        </dynamically_allocate_fragmented_samples>
    </reader_resource_limits>
    <property>
        <value>
            <!-- Java will use a buffer of 33KB to
                 deserialize all frames with a
                 serialized size smaller or equal than
                 33KB. When an I-frame is received,
                 Java will re-allocate the
                 deserialization buffer to match the
                 serialized size of the new frame.
            -->
            <element>
                <name>
                    dds.data_reader.history.memory_manager.java_stream.min_size
                </name>
                <value>33792</value>
            </element>
            <element>
                <name>
                    dds.data_reader.history.memory_manager.java_stream.trim_to_size
                </name>
                <value>1</value>
            </element>
        </value>
    </property>
</qos_profile>
</qos_library>
</dds>

```

To avoid preallocation of the samples in the *DataReader's* queue to their maximum size for Type-Plugin generated code in C, C++, and .NET, replace the bounded sequence in *VideoFrame* with an unbounded sequence and generate code using the **-unboundedSupport** command-line option:

```

struct VideoFrame {
    boolean keyFrame;
    sequence<octet> data;
};

```

See [Memory Management for DataReaders Using Generated Type-Plugins \(Section 20.2.1 on page 854\)](#) for more details.

To avoid preallocation of the samples in the *DataReader's* queue to their maximum size for *DynamicData*, set the **min_size_serialized** property to avoid the allocation of 1MB buffers for the *DataReader* queue samples (See [Memory Management for DynamicData DataReaders \(Section 20.2.3 on page 857\)](#)).

20.3 Instance-Data Memory Management for DataWriters

When an instance is registered with a *DataWriter*, the *DataWriter* serializes the key value and stores it with the instance.

Each instance maintained by the *DataWriter* has an associated buffer in which the *DataWriter* serializes the key. This buffer is either:

- Obtained from a pre-allocated pool (if the key's serialized size is \leq `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`)
- Dynamically allocated from the heap (if the key's serialized size is $>$ `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`).

See [Table 20.4 Instance-Data Memory Management Properties for DataWriters](#).

Table 20.4 Instance-Data Memory Management Properties for DataWriters

Property	Description
<code>dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size</code>	<p>Controls the memory allocation for the serialized key buffer that is stored with every instance.</p> <p>Default: -1 (UNLIMITED). All DDS sample buffers are obtained from the pre-allocated pool. The buffer size is the maximum serialized size of the DDS samples, as returned by the type plugin <code>get_serialized_sample_max_size()</code> operation.</p> <p>Note: This property also controls DDS sample-data memory management. See DDS Sample-Data Memory Management for DataWriters (Section 20.1 on page 846).</p>

20.4 Instance-Data Memory Management for DataReaders

When an instance is received and registered by a *DataReader*, the *DataReader* serializes the key value and stores it with the instance.

Each instance maintained by the *DataReader* has an associated buffer in which the *DataReader* serializes the key. This buffer is either:

Obtained from a pre-allocated pool (if the key's serialized size is \leq `dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size`)

Dynamically allocated from the heap (if the key's serialized size is $>$ `dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size`)

See [Table 20.5 Instance-Data Memory Management Properties for DataReaders](#) .

Table 20.5 Instance-Data Memory Management Properties for DataReaders

Property	Description
dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size	Controls the memory allocation for the serialized key buffer that is stored with every instance in the <i>DataReader</i> 's queue. Default: -1 (UNLIMITED). All buffers come from the pre-allocated pool. The size of the buffers is the maximum serialized size of the key as returned by the type plugin <code>get_serialized_key_max_size()</code> operation.

Chapter 21 Troubleshooting

This chapter contains tips on troubleshooting Connex DDS applications. For an up-to-date list of frequently asked questions, see the RTI Support Portal, accessible from <https://support.rti.com>—select the **Find Solution** link to see example code, general information on Connex DDS, performance information, troubleshooting tips, and technical details.

This chapter contains the following sections:

21.1 What Version am I Running?

There are two ways to obtain version information:

- By looking at the revision files, as described in [Finding Version Information in Revision Files \(Section 21.1.1 below\)](#).
- Programmatically at run time, as described in [Finding Version Information Programmatically \(Section 21.1.2 on the next page\)](#).

21.1.1 Finding Version Information in Revision Files

In the top-level directory of your Connex DDS installation (`${NDDSHOME}`), you will find text files that include revision information. The files are named `rev_<product>_rtids.<version>`. For example, you might see files called `rev_host_rtids.5.x.y` and `rev_persistence_rtids5.x.y` (where `x` and `y` stand for the version numbers of the current release). Each file contains more details, such as a patch level and if the product is license managed.

For example:

```
Host Build 5.x.y rev 04 (0x04050200)
```

The revision files for Connex DDS target libraries are in the same directory as the libraries (`${NDDSHOME}/lib/<architecture>`).

21.1.2 Finding Version Information Programmatically

The methods in the `NDDSSConfigVersion` class can be used to retrieve version information for the Connex DDS product, the core library, and the C, C++ or Java libraries.

The version information includes four fields:

- A major version number
- A minor version number
- A release number
- A build number

[Table 21.4 NDDSSConfigLogger Operations](#) lists the available operations (they will vary somewhat depending on the programming language you are using; consult the API Reference HTML documentation for more information).

Table 21.1 NDDSSConfigVersion Operations

Purpose	Operation	Description
To retrieve version information in a structured format	<code>get_product_version</code>	Gets version information for the Connex DDS product.
	<code>get_core_version</code>	Gets version information for the Connex DDS core library.
	<code>get_c_api_version</code>	Gets version information for the Connex DDS C library.
	<code>get_cpp_api_version</code>	Gets version information for the Connex DDS C++ library.
To retrieve version information in string format	<code>to_string</code>	Converts the version information for each library into a string. The strings for each library are put in a single hyphen-delimited list.

The `get_product_version()` operation returns a reference to a structure of type `DDS_ProductVersion_t`:

```
struct NDDS_Config_ProductVersion_t {
    DDS_Char  major;
    DDS_Char  minor;
    DDS_Char  release;
    DDS_Char  revision;
};
```

The other `get_*_version()` operations return a reference to a structure of type `NDDSS_Config_LibraryVersion_t`:

```

struct NDDS_Config_LibraryVersion_t {
    DDS_Long major;
    DDS_Long minor;
    char release;
    DDS_Long build;
};

```

The `to_string()` operation returns version information for the Connex DDS core, followed by the C and C++ API libraries, separated by hyphens. For example:

```

nddscore1.0g.rev0-nddsc1.0g.rev1-nddscpp1.0g.rev0

```

<p>Core major: 1 minor: 0 release: g build: 0</p>	<p>C API: major: 1 minor: 0 release: g build: 1</p>	<p>C++ API: major: 1 minor: 0 release: g build: 0</p>
---	---	---

21.2 Controlling Messages from Connex DDS

Connex DDS provides several types of messages to help you debug your system and alert you to errors during run time. You can control how much information is reported and where it is logged.

How much information is logged is known as the *verbosity* setting. [Table 21.2 Message Logging Verbosity Levels](#) describes the increasing verbosity levels.

Table 21.2 Message Logging Verbosity Levels

Verbosity (NDDS_CONFIG_LOG_VERBOSITY_*)	Description
SILENT	No messages will be logged. (lowest verbosity)

Table 21.2 Message Logging Verbosity Levels

Verbosity (NDDS_CONFIG_LOG_VERBOSITY_*)	Description
ERROR (default level for all categories)	Log only high-priority error messages. An error indicates something is wrong with how Connex DDS is functioning. The most common cause of this type of error is an incorrect configuration.
WARNING	Additionally log warning messages. A warning indicates that Connex DDS is taking an action that may or may not be what you intended. Some configuration information is also logged at this verbosity to aid in debugging.
STATUS_LOCAL	Additionally log verbose information about the lifecycles of local Connex DDS objects.
STATUS_REMOTE	Additionally log verbose information about the lifecycles of remote Connex DDS objects.
STATUS_ALL	Additionally log verbose information about periodic activities and Connex DDS threads. (highest verbosity)

Note that the verbosity levels are cumulative: logging at a high verbosity means also logging all lower verbosity messages. If you change nothing, the default verbosity will be set to `NDDS_CONFIG_LOG_VERBOSITY_ERROR`.

Logging at high verbosity levels can be detrimental to your application's performance. You should generally not set the verbosity above `NDDS_CONFIG_LOG_VERBOSITY_WARNING`, unless you are debugging a specific problem.

You will typically change the verbosity of all of Connex DDS at once. However, in the event that such a strategy produces too much output, you can further discriminate among the messages you would like to see. The types of messages logged by Connex DDS fall into the categories listed in [Table 21.3 Message Logging Categories](#); each category can be set to a different verbosity level.

Table 21.3 Message Logging Categories

Category (NDDS_CONFIG_LOG_CATEGORY_*)	Description
PLATFORM	Messages about the underlying platform (hardware and OS).
COMMUNICATION	Messages about data serialization and deserialization and network traffic.
DATABASE	Messages about the internal database of Connex DDS objects.

Table 21.3 Message Logging Categories

Category (NDDS_CONFIG_LOG_CATEGORY_*)	Description
ENTITIES	Messages about local and remote entities and the discovery process.
API	Messages about Connex DDS's API layer (such as method argument validation).

The methods in the **NDDSConfigLogger** class can be used to change verbosity settings, as well as the destination for logged messages. [Table 21.4 NDDSConfigLogger Operations](#) lists the available operations; consult the API Reference HTML documentation for more information.

Table 21.4 NDDSConfigLogger Operations

Purpose	Operation	Description
Change Verbosity for all Categories	get_verbosity	Gets the current verbosity. If per-category verbosity is used, returns the highest verbosity of any category.
	set_verbosity	Sets the verbosity of all categories.
Change Verbosity for a Specific Category	get_verbosity_by_category	Gets/Sets the verbosity for a specific category.
	set_verbosity_by_category	
Change Destination of Logged Messages	get_output_file	Returns the file to which messages are being logged, or NULL for the default destination (standard output on most platforms).
	set_output_file	Redirects future logged messages to the specified file (or NULL to return to the default)
	get_output_device	Returns the logging device installed with the logger.
	set_output_device	Registers a specified logging device with the logger. See Customizing the Handling of Generated Log Messages (Section 21.2.3 on page 872)
Change Message Format	get_print_format	Gets/Sets the current message format that Connex DDS is using to log diagnostic information. See Format of Logged Messages (Section 21.2.1 on the next page) .
	set_print_format	

21.2.1 Format of Logged Messages

You can control the amount of information in each message with the `set_print_format()` operation. The format options are listed in [Table 21.5 Message Formats](#).

Table 21.5 Message Formats

Message Format (<code>NDDS_CONFIG_LOG_PRINT_FORMAT_*</code>)	Description
DEFAULT	Message, method name, and activity context.
TIMESTAMPED	Message, method name, activity context, and timestamp.
VERBOSE	Message with all available context information (includes thread identifier, activity context).
VERBOSE_TIMESTAMPED	Message with all available context information and timestamp.
DEBUG	Information for internal debugging by RTI personnel.
MINIMAL	Message number, method name.
MAXIMAL	All available fields.

Of course, you are not likely to recognize all of the method names; many of the operations that perform logging are deep within the implementation of Connexr DDS. However, in case of errors, logging will typically take place at several points within the call stack; the output thus implies the stack trace at the time the error occurred. You may only recognize the name of the operation that was the last to log its message (i.e., the function that called all the others); however, the entire stack trace is extremely useful to RTI support personnel in the event that you require assistance.

You may notice that many of the logged messages begin with an exclamation point character. This convention indicates an error and is intended to be reminiscent of the negation operator in many programming languages. For example, the message “!create socket” in the second line of the above stack trace means “cannot create socket.”

21.2.1.1 Timestamps

Reported times are in seconds from a system-dependent starting time; these are equivalent to the output format from Connexr DDS. The timestamp is in the form “`sssss.mmmmmm`” where `<sssss>` is a number of seconds, and `<mmmmm>` is a fraction of a second expressed in microseconds. Enabling timestamps will result in some additional overhead for clock access for every message that is logged.

Logging of timestamps is not enabled by default. To enable it, use `NDDS_Config_Logger` method `set_print_format()`.

21.2.1.2 Thread identification

Thread identification strings uniquely identify for active thread when a message is output to the console. A thread may be a user (application) thread or one of several types of internal threads. The possible thread types are:

user thread: U<threadID>

receive thread: rR<thread index><domain ID><app ID>, where thread index is an integer identifying this receive thread

event thread: revt<domain ID><app ID>

asynchronous publisher thread: rDsp

Logging of thread IDs are not enabled by default. To enable it, use NDDS_Config_Logger method **set_print_format()**.

21.2.1.3 Hierarchical Context

Many middleware APIs now store information in thread-specific storage about the current operation, as well as information about which DDS domain (and participant ID) was active, and which entities were being operated on. In the case of objects that are associated with topics, the topic name is also stored.

The context field is output by default.

21.2.1.4 Explanation of Context Strings

- DDS domain context

Dxxyy

In this case, xx = participant ID, yy = domain #. For example, **D0149** means “domain 49, participant 01.”

- Entity context

Operation on an entity will specify the object and a numeric ID, such as **Writer(001A1)**. The name will be one of the following:

String	Object Type
Participant	DDS_DomainParticipant
Pub	DDS_Publisher
Sub	DDS_Subscriber

String	Object Type
Topic	DDS_Topic
Writer	DDS_<*>DataWriter
Reader	DDS_<*>DataReader

- Topic Context

T=Hello refers to topic "Hello."

The operations which report context include:

String	Operation
Entity operations:	
ENABLE	Entity::enable
GET_QOS	Entity::get_qos
SET_QOS	Entity::set_qos
GET_LISTENER	Entity::get_listener
SET_LISTENER	Entity::set_listener
Factory operations (DP Factory, Participant, Pub/Sub):	
CREATE <Entity>	Factory::create_<entity>
DELETE <Entity>	Factory::delete_<entity>
GET_DEFAULT_QOS <Entity>	Factory::get_default_<entity>_qos
SET_DEFAULT_QOS <Entity>	Factory::set_default_<entity>_qos
Participant-specific operations:	
GET_PUBS	Participant::get_publishers
GET_SUBS	Participant::get_subscribers
LOOKUP Topic(<name>)	Participant::lookup_topicdescription
LOOKUP FlowController(<name>)	Participant::lookup_flowcontroller
IGNORE <Entity>(<host ID>)	Participant::ignore_<entity>

21.2.2 Configuring Logging via XML

Logging can also be configured using the DomainParticipantFactory's [LOGGING QoS Policy \(DDS Extension\)](#) (Section 8.4.1 on page 572) with the tags, `<participant_factory_qos><logging>`. The fields in the LoggingQoS Policy are described in XML using a 1-to-1 mapping with the equivalent C representation shown below:

```
struct DDS_LoggingQoSPolicy {
    NDDS_Config_LogVerbosity verbosity;
    NDDS_Config_LogCategory category;
    NDDS_Config_LogPrintFormat print_format;
    char * output_file;
};
```

The equivalent representation in XML:

```
<participant_factory_qos>
  <logging>
    <verbosity></verbosity>
    <category></category>
    <print_format></print_format>
    <output_file></output_file>
  </logging>
</participant_factory_qos>
```

The attribute `<is_default_participant_factory_profile>` can be set to true for the `<qos_profile>` tag to indicate from which profile to use `<participant_factory_qos>`. If multiple QoS profiles have `<is_default_participant_factory_profile>` set to true, the last profile with `<is_default_participant_factory_profile>` set to true will be used.

If none of the profiles have set `<is_default_participant_factory_profile>` to true, the profile with `<is_default_qos>` set to true will be used.

In the following example, DefaultProfile2 will be used:

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../xsd/rti_dds_qos_profiles.xsd">
  <!-- Qos Library -->
  <qos_library name="DefaultLibrary">
    <qos_profile name="DefaultProfile1"
      is_default_participant_factory_profile ="true">
      <participant_factory_qos>
        <logging>
          <verbosity>ALL</verbosity>
          <category>ENTITIES</category>
          <print_format>MAXIMAL</print_format>
          <output_file>LoggerOutput1.txt</output_file>
        </logging>
      </participant_factory_qos>
    </qos_profile>
    <qos_profile name=
      "DefaultProfile2"
      is_default_participant_factory_profile ="true">
```

```

    <participant_factory_qos>
      <logging>
        <verbosity>WARNING</verbosity>
        <category>API</category>
        <print_format>VERBOSE_TIMESTAMPED</print_format>
        <output_file>LoggerOutput2.txt</output_file>
      </logging>
    </participant_factory_qos>
  </qos_profile>
  <qos_profile name="DefaultProfile3" is_default_qos="true">
    <participant_factory_qos>
      <logging>
        <verbosity>ERROR</verbosity>
        <category>DATABASE</category>
        <print_format>VERBOSE</print_format>
        <output_file>LoggerOutput3.txt</output_file>
      </logging>
    </participant_factory_qos>
  </qos_profile>
</qos_library>
</dds>

```

Note: The `LoggingQosPolicy` is currently the only QoS policy that can be configured using the `<participant_factory_qos>` tag.

21.2.3 Customizing the Handling of Generated Log Messages

By default, the log messages generated by Connex DDS are sent to the standard output. You can redirect the log messages to a file by using the `set_output_file()` operation,

To further customize the management of the generated log messages, you can use the `Logger`'s `set_output_device()` operation to install a user-defined logging device. The logging device must implement an interface with two operations: `write()` and `close()`.

Connex DDS will call the `write()` operation to write a new log message to the input device. The log message provides the text and the verbosity corresponding to the message.

Connex DDS will call the `close()` operation when the logging device is uninstalled.

Note: It is not safe to make any calls to the Connex DDS core library including calls to `DDS_DomainParticipant_get_current_time()` from any of the logging device operations.

For additional details on user-defined logging devices, see the API Reference HTML documentation (under **Modules, RTI Connex DDS API Reference, Configuration Utilities**).

Part 4: Request-Reply Communication Pattern

The Request-Reply communication pattern is only available with the Connex DDS Professional, Evaluation, and Basic package types.

As real-time and embedded applications become more complex, and require integration with enterprise applications, you may need additional communication patterns besides publish-subscribe. Perhaps your application needs certain information only occasionally—such as changes in temperature over the past hour, or even just once, such as application configuration data that is required only at start up. To get information only when needed, Connex DDS supports a *request-reply* communication pattern, which is described in the following sections:

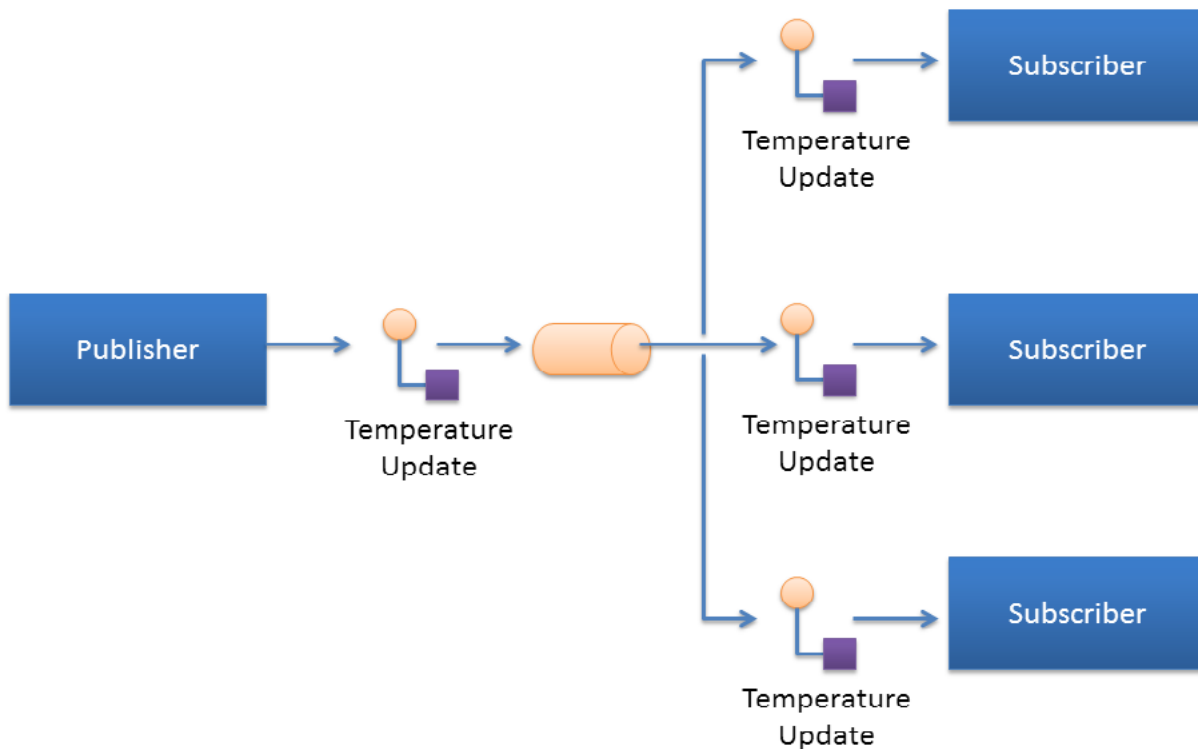
- [Introduction to the Request-Reply Communication Pattern \(Section Chapter 22 on page 874\)](#)
- [Using the Request-Reply Communication Pattern \(Section Chapter 23 on page 880\)](#)

Chapter 22 Introduction to the Request-Reply Communication Pattern

This chapter describes the Request-Reply communication pattern, which is available with the Connex DDS Professional, Evaluation, and Basic package types.

The fundamental communication pattern provided by Connex DDS is known as DDS data-centric *publish-subscribe*. The data-centric publish-subscribe pattern is particularly well-suited in situations where the same data must flow from one producer to many consumers, or when data is streaming continuously from producers to consumers. For example, the values produced by a temperature sensor may be observed by multiple applications, such as control applications, UI applications, supervisory applications, historians, etc.

Figure 22.1 Publish-Subscribe Overview



Sending temperature updates using the publish-subscribe pattern

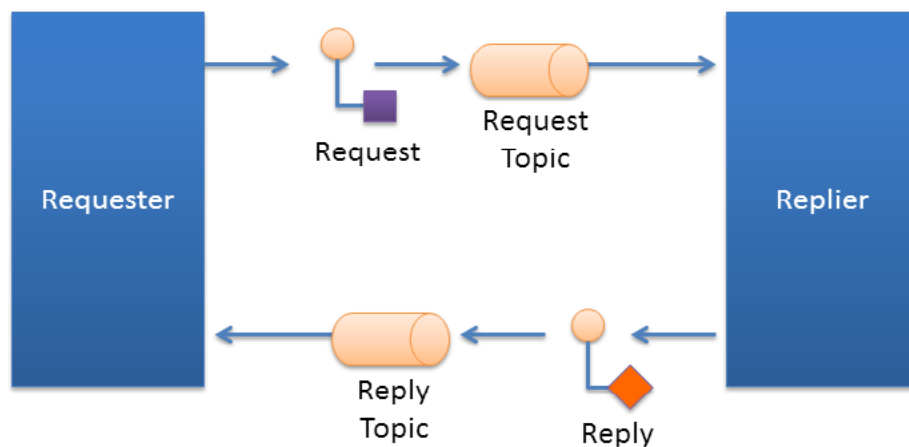
The publish-subscribe pattern supports multicast, which allows efficient distribution from a single source to multiple applications, devices, or subscribers simultaneously. But even with a single subscriber, the publish-subscribe pattern is still advantageous, because the publisher can push new updates to a subscriber as soon as they happen. That way the subscriber always has access to the latest data, with minimum delays, and without incurring the overhead of periodically polling what may be stale data. This efficient, low-latency access to the most current information is important for real-time applications.

22.1 The Request-Reply Pattern

As applications become more complex, it often becomes necessary to use other communication patterns in addition to publish-subscribe. Sometimes an application needs to get a one-time snapshot of information; for example, to make a query into a database or retrieve configuration parameters that never change. Other times an application needs to ask a remote application to perform an action on its behalf; for example, to invoke a remote procedure call or a service.

To support these scenarios, Connex DDS includes support for the request-reply communication pattern. It is available with the Connex DDS Professional, Evaluation, and Basic package types.

Figure 22.2 Request-Reply Overview



Request-Reply communication pattern using a Requester and a Replier

The request-reply pattern has two roles: The requester (service consumer or client) sends a request message and waits for a reply message. The replier (service provider) receives the request message and responds with a reply message.

Using the request-reply pattern with a *Replier* is straightforward. *Connex DDS* provides two Entities: the *Requester* and the *Replier* manage all the interactions on behalf of the application. The *Requester* and *Replier* automatically discover each other based on an application-specified *service name*. When the application invokes a request, the *Requester* sends a message (on an automatically-created request *Topic*) to the *Replier*, which notifies the receiving application. The application, in turn, uses the *Replier* to receive the request and send the reply message. The reply message is sent by *Connex DDS* back to the original *Requester* (using a different automatically created reply *Topic*).

Connex DDS supports both blocking and non-blocking request-reply interactions:

- In a blocking (a.k.a. synchronous) interaction, the requesting application blocks while waiting for the reply. This is typical of applications desiring remote-procedure-call or remote-method-invocation interactions.
- In a non-blocking (a.k.a. asynchronous) interaction, the requesting application can proceed with other work and gets notified when a reply is available.

[Repliers \(Section 23.2 on page 890\)](#) explains how an application can use the methods provided by the *Requester* and the *Replier* to perform both blocking and non-blocking request-reply interactions.

The implementation of request-reply in *Connex DDS* is highly scalable. A *Replier* can receive requests from thousands of *Requesters* at the same time. *Connex DDS* will efficiently deliver each reply only to the original *Requester*, allowing the number of *Requesters* to grow without significantly impacting each other.

22.1.1 Request-Reply Correlation

An application might have multiple outstanding requests, all originating from the same *Requester*. This can be as a result of using a non-blocking request-reply interaction, or as a result of having multiple application threads using the same *Requester*. Because of this, *Connex DDS* provides a way for the application to correlate a reply with the request it is associated with. This meta-data is provided as part of a *SampleInfo* structure that accompanies the reply.

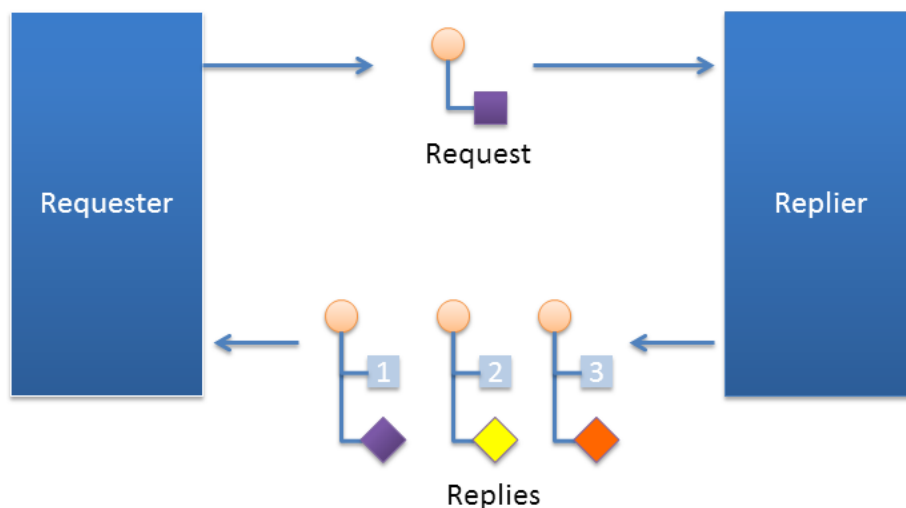
When using a blocking request operation, *Connex DDS* provides an easy-to-use API that automatically does the correlation for you.

22.2 Single-Request, Multiple-Replies

Connex DDS also supports the single-request multiple-reply pattern. This pattern is an extension of the basic request-reply pattern in which multiple reply messages can flow back as a result of a single request.

The single-request multiple-reply pattern is very useful when getting large amounts of data as a reply, such as when querying a system for all data that matches a certain criteria. Another common use-case is invoking a service that goes through multiple stages and provides updates on each: service commencement, progress reports, and final completion.

Figure 22.3 Single Request, Multiple Replies



Request/Reply communication pattern with multiple replies resulting from a single request

For example, a mobile asset management system may need to locate a particular asset (truck, locomotive, etc.). The system sends out the request. The first reply that comes back will read “locating.” The service has not yet determined the position, but it notifies the requester that the search operation has started. The

second reply might provide a status update on the search, perhaps including a rough area of location. The third and final reply will have the exact location of the asset.

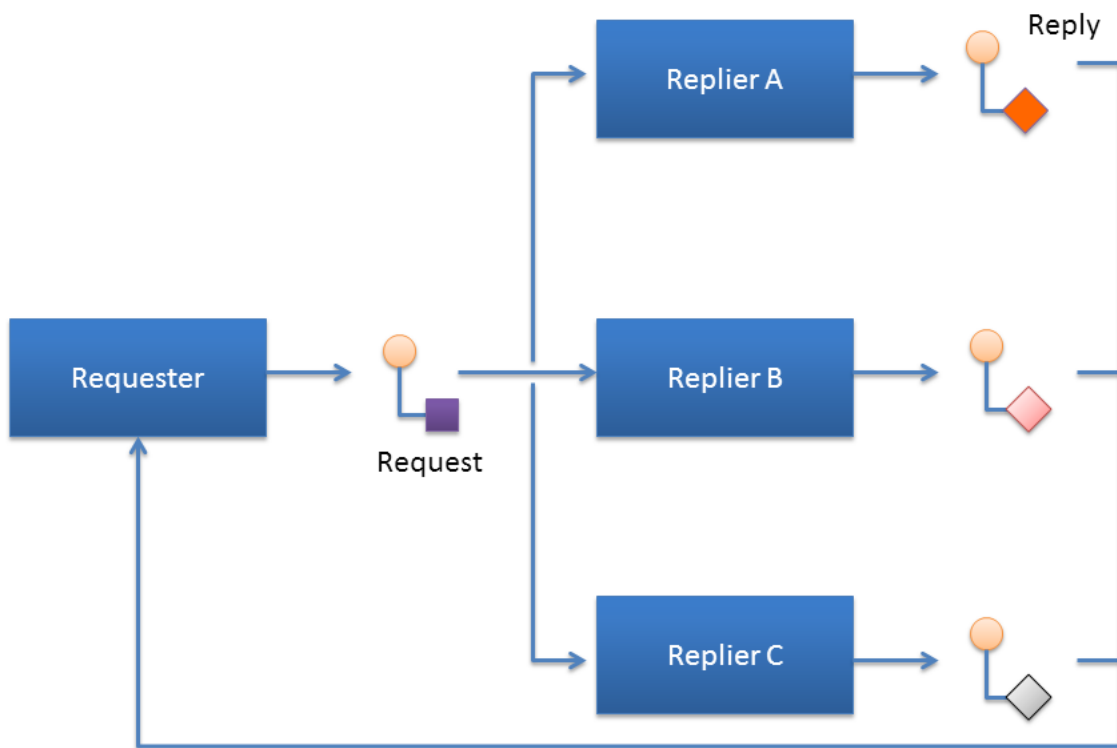
22.3 Multiple Repliers

Connex DDS directly supports applications that obtain results from multiple providers in parallel instead of in sequence, basically implementing functional parallelism.

To illustrate, consider a system managing a fleet of drones, like unmanned aerial vehicles (UAVs). Using the single request-multiple reply pattern, the application can use a *Requester* to send a single ‘DroneInfo’ request to all the drones to query for their current mission and status. Each drone replies with the information on its own status and the *Requester* aggregates all the responses for the application.

As another example, consider a system that would like to locate the best printer to perform a particular job. The application can use a *Requester* to query all the printers that are on-line for their characteristics and load. The *Requester* receives the replies and accumulates them until an application-specified number of replies is received (or a timeout elapses). The application can then use the *Requester* to access all the replies, examine their contents, and select the best printer for the job.

Figure 22.4 Multiple Repliers

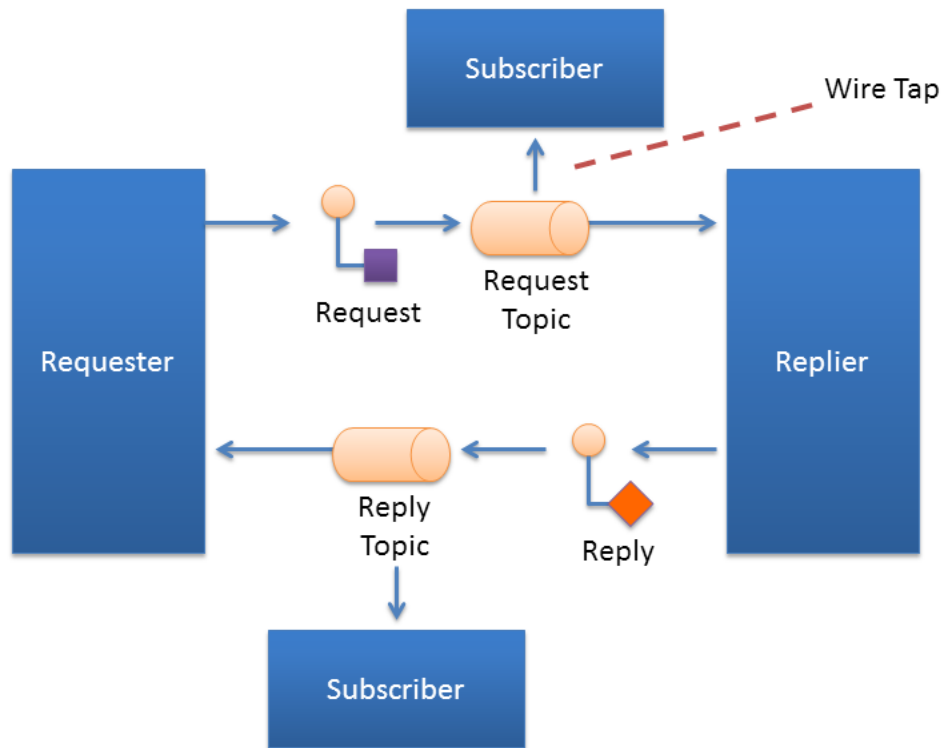


Request/Reply communication pattern with a single Requester and multiple Repliers

22.4 Combining Request-Reply and Publish-Subscribe

Under the hood, *Connex DDS* implements request-reply using the DDS data-centric publish-subscribe pattern. This has a key benefit in that the two patterns can be combined, and mapped without interference.

Figure 22.5 Combining Patterns



Combining Request-Reply and Publish-Subscribe patterns

For example, a pair of applications may be involved in a two-way conversation using request-reply. For debugging purposes or regulatory compliance, you want to inspect those request-reply messages, but without disrupting the conversation.

Since *Connex DDS* implements requests and replies using DDS data-centric publish subscribe, others can simply subscribe to the request and reply messages. You can introduce a subscriber to the reply *Topic*, without interfering with the two-way conversation between the *Requester* and the *Replier*. This pattern is also known as a Wire Tap. For example, you can use *RTI Recording Service* to non-intrusively capture request-reply traffic.

Chapter 23 Using the Request-Reply Communication Pattern

This section explains how to use and configure the Request-Reply communication pattern, which is only available with the Connex DDS Professional, Evaluation, and Basic package types.

There are two basic Connex DDS entities used by the Request-Reply communication pattern: *Requester* and *Replier*.

- A *Requester* publishes a request *Topic* and subscribes to a reply *Topic*. See [Requesters \(Section 23.1 on the next page\)](#).
- A *Replier* subscribes to the request *Topic* and publishes the reply *Topic*. See [Repliers \(Section 23.2 on page 890\)](#).

There is an alternate type of replier known as a *SimpleReplier*:

- A *SimpleReplier* is useful for cases where there is a single reply to each request and the reply can be generated quickly, such as looking up some data from memory.
- A *SimpleReplier* is used in combination with a user-provided *SimpleReplierListener*. Requests are passed to a callback in the *SimpleReplierListener*, which returns the reply.
- The *SimpleReplier* is not suitable if the replier needs to generate more than one reply for a single request or if generating the reply can take significant time or needs to occur asynchronously. For more information, see [SimpleRepliers \(Section 23.3 on page 896\)](#).

Additional resources. In addition to the information in this section, you can find more information and example code here:

- The *Connex DDS* API Reference HTML documentation¹ contains example code that will show you how to use API: From the **Modules** tab, navigate to **Programming How-To's, Request-Reply Examples**.
- The *Connex DDS* API Reference HTML documentation also contains the full API documentation for the *Requester*, *Replier*, and *SimpleReplier*. Under the **Modules** tab, navigate to **RTI Connex DDS API Reference, RTI Connex Request-Reply API Reference**.

Typecodes are required when using the Request-Reply communication pattern. To use this pattern, do not use *RTI Code Generator's* **-noTypeCode** flag. If typecodes are missing, the *Requester* will log an exception.

23.1 Requesters

A *Requester* is an entity with two associated DDS *Entities*: a DDS *DataWriter* bound to a request *Topic* and a DDS *DataReader* bound to a reply *Topic*. A *Requester* sends requests by publishing samples of the request *Topic*, and receives replies for those requests by subscribing to the reply *Topic*.

Valid types for request and reply *Topics* can be:

- For the C API:
 - DDS types generated by *RTI Code Generator*
- For all other APIs:
 - DDS types generated by *RTI Code Generator*
 - Built-in DDS types, such as, *String*, *KeyedString*, *Octets*, and *KeyedOctets*
 - DDS *DynamicData* Types

To communicate, a *Requester* and *Replier* must use the same request *Topic* name, the same reply *Topic* name, and be associated with the same DDS **domain_id**.

A *Requester* has an associated *DomainParticipant*, which can be shared with other requesters or *Connex DDS* entities. All the other entities required for request-reply interaction, including the request and reply *Topics*, the *DataWriter* for writing requests, and a *DataReader* for reading replies, are automatically created when the *Requester* is constructed.

Connex DDS guarantees that a *Requester* will only receive replies associated with the requests it sends.

The *Requester* uses the underlying *DataReader* not only to receive the replies, but also as a cache that can hold replies to multiple outstanding requests or even multiple replies to a single request. Depending on the

¹The API Reference HTML documentation is available for all supported programming languages. Open <NDDSHOME>/README.html.

HistoryQoSPolicy configuration of the *DataReader*, the *Requester* may allow replies to replace previous replies based on the reply data having the same value for the Key fields (see [DDS Samples, Instances, and Keys \(Section 2.3.1 on page 14\)](#)). The default configuration of the *Requester* does not allow replacing.

You can configure the QoS for the underlying *DataWriter* and *DataReader* in a QoS profile. By default, the *DataWriter* and *DataReader* are created with default values (DDS_DATAWRITER_QOS_DEFAULT and DDS_DATAREADER_QOS_DEFAULT, respectively) except for the following:

- [RELIABILITY QoSPolicy \(Section 6.5.19 on page 400\)](#): **kind** is set to RELIABLE.
- [HISTORY QoSPolicy \(Section 6.5.10 on page 376\)](#): **kind** is set to KEEP_ALL.
- Several other protocol-related settings for *Requesters* (see the API Reference HTML documentation: select **Modules, Programming How-To's, Request-Reply Examples**; then scroll down to the section on **Configuring Request-Reply QoS profiles**).

23.1.1 Creating a Requester

Before you can create a *Requester*, you need a *DomainParticipant* and a service name.

Note: The example code snippets in this section use the C++ API. You can find more complete examples in all the supported programming languages (C, C++, Java, C#) in the *Connex DDS API Reference HTML documentation* and in the “example” directory found in your Connex DDS installation.

To create a Requester with the minimum set of parameters, you can use the basic constructor that receives only an existing DDS *DomainParticipant* and the name of the service:

```
Requester <MyRequestType, MyReplyType> *requester =
    new Requester <MyRequestType, MyReplyType> (
        participant, "ServiceName");
```

To create a Requester with specific parameters, you may use a different constructor that receives a *RequesterParams* structure (described in [Setting Requester Parameters \(Section 23.1.3 on the next page\)](#)):

```
Requester (const RequesterParams &params)
```

The **ServiceName** parameter is used to generate the names of the request and reply *Topics* that the *Requester* and *Replier* will use to communicate. For example, if the service name is “MyService”, the topic names for the *Requester* and *Replier* will be “MyServiceRequest” and “MyServiceReply”, respectively. Therefore, for communication to occur, you must use the same service name when creating the *Requester* and the *Replier* entities.

If you want to use topic names different from the ones that would be derived from the ServiceName, you can override the default names by setting the actual request and reply *Topic* names using the **request_**

topic_name() and **reply_topic_name()** accessors to the *RequesterParams* structure prior to creating the *Requester*.

Example: To create a *Requester* with default QoS and topic names derived from the service name, you may use the following code:

```
Requester<Foo, Bar> * requester =
    new Requester<Foo, Bar>(
        participant, "MyService");
```

Example: To create a *Requester* with a specific QoS profile with library name “MyLibrary” and profile “MyProfile” defined inside **USER_QOS_PROFILES.xml** in the current working directory, you may use the following code:

```
Requester<Foo, Bar> * requester = new Requester<Foo, Bar>(
    RequesterParams(participant).
    service_name("MyService").qos_profile(
    "MyLibrary", "MyProfile"));
```

Once you have created a *Requester*, you can use it to perform the operations in [Table 23.2 Requester Operations](#).

23.1.2 Destroying a Requester

To destroy a *Requester* and free its underlying entities you may use the destructor:

```
virtual ~Requester ()
```

23.1.3 Setting Requester Parameters

To change the *RequesterParams* that can be used when creating a *Requester*, you can use the operations listed in [Table 23.1 Operations to Set Requester Parameters](#).

Table 23.1 Operations to Set Requester Parameters

Operation	Description
datareader_qos	Sets the QoS of the reply DataReader.
datawriter_qos	Sets the QoS of the request DataWriter.
publisher	Sets a specific Publisher.
qos_profile	Sets a QoS profile for the DDS entities in this requester.

Table 23.1 Operations to Set Requester Parameters

Operation	Description
request_topic_name	Sets the name of the Topic used for the request. If this parameter is set, then you must also set the reply_topic_name parameter and you should not set the service_name parameter.
reply_topic_name	Sets the name of the Topic used for the reply. If this parameter is set, then you must also set the request_topic_name parameter and you should not set the service_name parameter.
reply_type_support	Sets the type support for the reply type.
request_type_support	Sets the type support for the request type.
service_name	Sets the service name. This will automatically set the name of the request Topic and the reply Topic. If this parameter is set you should not set the request_topic_name or the reply_topic_name.
subscriber	Sets a specific Subscriber.

23.1.4 Summary of Requester Operations

There are several kinds of operations an application can perform using the *Requester*:

- Sending requests (i.e., publishing request samples on the request *Topic*)
- Waiting for replies to be received.
- Taking the reply data. This gets the reply data from the *Requester* and removes from the *Requester* cache.
- Reading the reply data. This gets the reply data from the *Requester* but leaves it in the *Requester* cache so it remain accessible to future operations on the *Requester*.
- Receiving replies (a convenience operation that is a combination of ‘waiting’ and ‘taking’ the data in a single operation)

These operations are summarized in [Table 23.2 Requester Operations](#)

Table 23.2 Requester Operations

Operation	Description	Reference
Sending Requests send_request	Sends a request.	Sending Requests (Section 23.1.5 on the next page)

Table 23.2 Requester Operations

Operation		Description	Reference
Waiting for Replies	wait_for_replies	Waits for replies to any request or to a specific request.	Waiting for Replies (Section 23.1.6.1 on the facing page)
Taking Reply Data	take_reply	Copies a single reply into a Sample container. There are variants that allow getting the next reply available or the next reply to a specific request. This operation removes the reply from the Requester cache. So subsequent calls to take or read replies will not get the same reply again.	Repliers (Section 23.2 on page 890)
	take_replies	Returns a LoanedSamples container with the collection of replies received by the Requester. There are variants that allow accessing all the replies available or only the replies to a specific request. This operation removes the returned replies from the Requester cache. So subsequent calls to take or read replies will not get the same replies again.	
Reading Reply Data	read_reply	Copies a single reply into a Sample container. There are variants that allow getting the next reply available or the next reply to a specific request. This operation leaves the reply on the Requester cache. So subsequent calls to take or read replies can get the same reply again.	Repliers (Section 23.2 on page 890)
	read_replies	Returns a LoanedSamples container with the collection of replies received by the Requester. There are variants that allow accessing all the replies available or only the replies to a specific request. This operation leaves the returned replies in the Requester cache. So subsequent calls to take or read replies can get the same replies again.	
Receiving Replies	receive_reply	Convenience function that combines a call to wait_for_replies with a call to take_reply.	Receiving Replies (Section 23.1.6.3 on page 889)
	receive_replies	Convenience function that combines a call to wait_for_replies with a call to take_replies.	
Getting Underlying Entities	get_request_datawriter	Retrieves the underlying DataWriter that writes requests.	Accessing Underlying DataWriters and DataReaders (Section 23.4 on page 898)
	get_reply_datareader	Retrieves the underlying DataReader that reads replies.	

23.1.5 Sending Requests

To send a request, use the **send_request()** operation on the *Requester*. There are three variants of this operation, depending on the parameters that are passed in:

1. `send_request (const TRequest &request)`
2. `send_request (WriteSample<TRequest> &request)`
3. `send_request (WriteSampleRef<TRequest> &request)`

The first variant simply sends a request.

The second variant sends a request and gets back information about the request in a *WriteSample* container. This information can be used to correlate the request with future replies.

The third variant is just like the second, but puts the information in a *WriteSampleRef*, which holds references to the data and parameters. Both *WriteSample* and *WriteSampleRef* provide information about the request that can be used to correlate the request with future replies.

23.1.6 Processing Incoming Replies with a Requester

The *Requester* provides several operations that can be used to wait for and access replies:

- `wait_for_replies()`, see [Waiting for Replies \(Section 23.1.6.1 below\)](#)
- `take_reply()`, `take_replies()`, `read_reply()` and `read_replies()`, see [Getting Replies \(Section 23.1.6.2 on the next page\)](#)
- `receive_reply()` and `receive_replies()`, see [Receiving Replies \(Section 23.1.6.3 on page 889\)](#)

The `wait_for_replies` operations are used to wait until the replies arrive.

The `take_reply`, `take_replies`, `read_reply`, and `read_replies()` operations access the replies once they have arrived.

The `receive_reply` and `receive_replies` are convenience functions that combine waiting and accessing the replies and are equivalent to calling the ‘wait’ operation followed by the corresponding `take_reply` or `take_replies` operations.

Each of these operations has several variants, depending on the parameters that are passed in.

23.1.6.1 Waiting for Replies

Use the `wait_for_replies()` operation on the *Requester* to wait for the replies to previously sent requests. There are three variants of this operation, depending on the parameters that are passed in. All these variants block the calling thread until either there are replies or a timeout occurs.

```

1. wait_for_replies (const DDS_Duration_t &max_wait)
2. wait_for_replies (int min_count, const DDS_Duration_t &max_wait)
3. wait_for_replies (int min_count,
                    const DDS_Duration_t &max_wait,
                    const SampleIdentity_t &related_request_id)

```

The first variant (only passing in **max_wait**) blocks until a reply is available or until **max_wait** time has elapsed, whichever comes first. The reply can be to any of the requests made by the *Requester*.

The second variant (passing in **min_count** and **max_wait**) blocks until at least **min_count** replies are available or until **max_wait** time has elapsed, whichever comes first. These replies may all be to the same request or to different requests made by the *Requester*.

The third variant (passing in **min_count**, **max_wait**, and **related_request_id**) blocks until at least **min_count** replies to the request identified by the **related_request_id** are available, or until **max_wait** time has passed, whichever comes first. Note that unlike the previous variants, the replies must all be to the same single request (identified by the **related_request_id**) made by the *Requester*.

Typically after waiting for replies, you will call **take_reply**, **take_replies**, **read_reply**, or **read_replies()**, see [Repliers \(Section 23.2 on page 890\)](#).

If you call **wait_for_replies()** several times without ‘taking’ the replies (using the **take_reply** or **take_replies** operation), future calls to **wait_for_replies()** will return immediately and will not wait for new replies.

23.1.6.2 Getting Replies

You can use the following operations to access replies: **take_reply**, **take_replies**, **read_reply**, and **read_replies()**.

As mentioned in [Summary of Requester Operations \(Section 23.1.4 on page 884\)](#), the difference between the ‘take’ operations (**take_reply**, **take_replies**) and the ‘read’ operations (**read_reply**, **read_replies**) is that ‘take’ operations remove the replies from the *Requester* cache. This means that future calls to **take_reply**, **read_reply**, **read_reply**, and **read_reply** will not get the same reply again.

The **take_reply** and **read_reply** operations access a *single* reply, whereas the **take_replies** and **read_replies** can access a *collection* of replies.

There are four variants of the **take_reply** and **read_reply** operations, depending on the parameters that are passed in:

```

1. take_reply (Sample<TReply> &reply)

   read_reply (Sample<TReply> &reply)

2. take_reply (SampleRef<TReply> reply)

   read_reply (SampleRef<TReply> reply)

3. take_reply (Sample<TReply> &reply,
              const SampleIdentity_t &related_request_id)

   read_reply (Sample<TReply> &reply,
              const SampleIdentity_t &related_request_id)

```

```

4. take_reply (SampleRef<TReply> reply,
              const SampleIdentity_t &related_request_id)

   read_reply (SampleRef<TReply> reply,
              const SampleIdentity_t &related_request_id)

```

The first two variants provide access to the next reply in the *Requester* cache. This is the earliest reply to any previous requests sent by the *Requester* that has not been ‘taken’ from the *Requester* cache. The remaining two variants provide access to the earliest non-previously ‘taken’ reply to the request specified by the **related_request_id**.

Notice that some of these variants use a *Sample*, while other use a *SampleRef*. A *SampleRef* can be used much like a *Sample*, but it holds *references* to the reply data and *DDS SampleInfo*, so there is no additional copy. In contrast using the *Sample* obtains a copy of both the data and *DDS SampleInfo*.

The **take_replies** and **read_replies** operations access a collection of (one or more) replies to previously sent requests. These operations are convenient when you expect multiple replies to a single request, or when issuing multiple requests concurrently without waiting for intervening replies.

The **take_replies** and **read_replies** operations return a *LoanedSamples* container that holds the replies. To increase performance, the *LoanedSamples* does not copy the reply data. Instead it ‘loans’ the necessary resources from the *Requester*. The resources loaned by the *LoanedSamples* container must be eventually returned, either explicitly calling the **return_loan()** operation on the *LoanedSamples* or through the destructor of the *LoanedSamples*.

There are three variants of the **take_replies** and **read_replies** operations, depending on the parameters that are passed in:

```

1. take_replies (int max_count=DDS_LENGTH_UNLIMITED)

   read_replies (int max_count=DDS_LENGTH_UNLIMITED)

2. take_replies (int max_count,
                const SampleIdentity_t &related_request_id)

   read_replies (int max_count,
                const SampleIdentity_t &related_request_id)

3. take_replies (const SampleIdentity_t &related_request_id)

   read_replies (const SampleIdentity_t &related_request_id)

```

The first variant (only passing in **max_count**) returns a container holding up to **max_count** replies.

The second variant (passing in **max_count** and **related_request_id**) returns a *LoanedSamples* container holding up to **max_count** replies that correspond to the request identified by the **related_request_id**.

The third variant (only passing in **related_request_id**) returns a *LoanedSamples* container holding an unbounded number of replies that correspond to the request identified by the **related_request_id**. This is equivalent to the second variant with **max_count** = DDS_LENGTH_UNLIMITED.

The resources for the *LoanedSamples* container must be eventually be returned, either by calling the **return_loan()** operation on the *LoanedSamples* or through the *LoanedSamples* destructor.

For multi-reply scenarios, in which a *Requester* receives multiple replies from a *Replier* for a given request, the *Requester* can check if a reply is the last reply in a sequence of replies. To do so, see if the bit INTERMEDIATE_REPLY_SEQUENCE_SAMPLE is set in DDS_SampleInfo's flag field (see [Table 7.18 DDS_SampleInfo Structure](#)) after receiving each reply. This bit indicates it is NOT the last reply.

23.1.6.3 Receiving Replies

The **receive_reply()** operation is a shortcut that combines calls to **wait_for_replies()** and to **take_reply()**. Similarly the **receive_replies()** operation combines **wait_for_replies()** and **take_replies()**.

There is only one variant of the **receive_reply()** operation:

```
1. receive_reply (Sample<TReply> &reply, const DDS_Duration_t &timeout)
```

This operation blocks until either a reply is received or a timeout occurs. The contents of the reply are copied into the provided sample (**reply**).

There are two variants of the **receive_replies()** operation, depending on the parameters that are passed in:

```
1. receive_replies (const DDS_Duration_t &max_wait)

2. receive_replies (int min_count, int max_count,
                   const DDS_Duration_t &max_wait)
```

These two variants block until *multiple* replies are available or a timeout occurs.

The first variant (only passing in **max_wait**) blocks until at least one reply is available or until **max_wait** time has passed, whichever comes first. The operation returns a *LoanedSamples* container holding the replies. Note that there could be more than one reply. This can occur if, for example, there were already replies available in the *Requester* from previous requests that were not processed. This operation does not limit the number of replies that can be returned on the *LoanedSamples* container.

The second variant (passing in **min_count**, **max_count**, and **max_wait**) will block until **min_count** replies are available or until **max_wait** time has passed, whichever comes first. Up to **max_count** replies will be stored into the *LoanedSamples* container which is returned to the caller.

The resources held in the *LoanedSamples* container must eventually be returned, either with an explicit call to **return_loan()** on the *LoanedSamples* or through the *LoanedSamples* destructor.

23.2 Repliers

A *Replier* is an entity with two associated DDS *Entities*: a DDS *DataReader* bound to a request *Topic* and a DDS *DataWriter* bound to a reply *Topic*. The *Replier* receives requests by subscribing to the request *Topic* and sends replies to those requests by publishing on the reply *Topic*.

Valid data types for these topics are the same as specified for the *Requester*, see [Requesters \(Section 23.1 on page 881\)](#).

For multi-reply scenarios in which a *Replier* generates more than one reply for a request, the *Replier* should mark all intermediate replies (all but the last reply) with the INTERMEDIATE_REPLY_SEQUENCE_SAMPLE bit-flag in the WriteParams_t flag field (see [Table 6.16 DDS_WriteParams_t](#)).

Much like a *Requester*, a *Replier* has an associated DDS *DomainParticipant* which can be shared with other Connex DDS entities. All the other entities required for the request-reply interaction, including a *DataWriter* for writing replies and a *DataReader* for reading requests, are automatically created when the *Replier* is constructed.

You can configure the QoS for the underlying *DataWriter* and *DataReader* in a QoS profile. By default, the *DataWriter* and *DataReader* are created with default QoS values (using DDS_DATAWRITER_QOS_DEFAULT and DDS_DATAREADER_QOS_DEFAULT, respectively) except for the following:

- [RELIABILITY QoSPolicy \(Section 6.5.19 on page 400\)](#): **kind** is set to RELIABLE
- [HISTORY QoSPolicy \(Section 6.5.10 on page 376\)](#): **kind** is set to KEEP_ALL

The *Replier* API supports several ways in which the application can be notified of, and process, requests:

- **Blocking**: The application thread blocks waiting for requests, processes them, and dispatches the reply. In this situation, if the computation necessary to process the request and produce the reply is small, you may consider using the *SimpleReplier*, which offers a simplified API.
- **Polling**: The application thread checks (polls) for requests periodically but does not block to wait for them. To check for data without blocking, call **take_requests()** or **read_requests()**.
- **Asynchronous notification**: The application installs a *ReplierListener* to receive notifications whenever a request is received.

23.2.1 Creating a Replier

To create a *Replier* with the minimum set of parameters you can use the basic constructor that receives only an existing DDS *DomainParticipant* and the name of the service:

```
Replier (DDSDomainParticipant * participant,
         const std::string & service_name)
```

Example:

```
Replier<Foo, Bar> * replier =
    new Replier<Foo, Bar>(participant, "MyService");
```

To create a *Replier* with specific parameters you may use a different constructor that receives a *ReplierParams* structure:

```
Replier (const ReplierParams<TRequest, TReply> &params)
```

Example:

```
Replier<Foo, Bar> * replier = new Replier<Foo, Bar>(
    ReplierParams(participant).service_name("MyService")
    .qos_profile("MyLibrary", "MyProfile"));
```

The **service_name** is used to generate the names of the request and reply *Topics* that the *Requester* and *Replier* will use to communicate. For example, if the service name is “MyService”, the topic names for the *Requester* and *Replier* will be “MyServiceRequest” and “MyServiceReply”, respectively. Therefore it is important to use the same **service_name** when creating the *Requester* and the *Replier*.

If you need to specify different *Topic* names, you can override the default names by setting the actual request and reply *Topic* names using **request_topic_name()** and **reply_topic_name()** accessors to the *ReplierParams* structure prior to creating the *Replier*.

23.2.2 Destroying a Replier

To destroy a Replier and free its underlying entities:

```
virtual ~Replier ()
```

23.2.3 Setting Replier Parameters

To change the *ReplierParams* that are used to create a *Replier*, use the operations listed in [Table 23.3 Operations to Set Replier Parameters](#).

Table 23.3 Operations to Set Replier Parameters

Operation	Description
datareader_qos	Sets the quality of service of the request <i>DataReader</i> .
datawriter_qos	Sets the quality of service of the reply <i>DataWriter</i> .
publisher	Sets a specific Publisher.

Table 23.3 Operations to Set Replier Parameters

Operation	Description
qos_profile	Sets a QoS profile for the entities in this replier.
replier_listener	Sets a listener that is called when requests are available.
reply_topic_name	Sets a specific reply topic name.
reply_type_support	Sets the type support for the reply type.
request_topic_name	Sets a specific request topic name.
request_type_support	Sets the type support for the request type.
service_name	Sets the service name the Replier offers and Requesters use to match.
subscriber	Sets a specific Subscriber.

23.2.4 Summary of Replier Operations

There are four kinds of operations an application can perform using the *Replier*:

- Waiting for requests to be received
- Reading/taking the request data and associated information
- Receiving requests (a convenience operation that combines waiting and getting the data into a single operation)
- Sending a reply for received request (i.e., publishing a reply sample on the reply *Topic* with special meta-data so that the original *Requester* can identify it).

The *Replier* operations are summarized in [Table 23.4 Replier Operations](#).

Table 23.4 Replier Operations

Operation	Description	Reference
Waiting for Requests wait_for_requests	Waits for requests.	Waiting for Requests (Section 23.2.5.1 on page 894)

Table 23.4 Replier Operations

Operation		Description	Reference
Taking Requests	take_request	Copies the contents of a single request into a <i>Sample</i> and removes it from the <i>Replier</i> cache.	Reading and Taking Requests (Section 23.2.5.2 on the facing page)
	take_requests	Returns a <i>LoanedSamples</i> to access multiple requests and removes the requests from the <i>Replier</i> cache.	
Reading Requests	read_request	Copies the contents of a single request into a <i>Sample</i> , leaving it in the <i>Replier</i> cache	
	read_requests	Returns a <i>LoanedSamples</i> to access multiple requests, leaving them in the <i>Replier</i> cache.	
Receiving Requests	receive_request	Waits for a single request and copies its contents into a <i>Sample</i> container.	Receiving Requests (Section 23.2.5.3 on page 895)
	receive_requests	Waits for multiple requests and provides a <i>LoanedSamples</i> container to access them.	
Sending Replies	send_reply	Sends a reply for a previous request.	Sending Replies (Section 23.2.6 on page 896)
Getting Underlying Entities	get_request_datareader	Retrieves the underlying <i>DataReader</i> .	Accessing Underlying DataWriters and DataReaders (Section 23.4 on page 898)
	get_reply_datawriter	Retrieves the underlying <i>DataWriter</i> .	

23.2.5 Processing Incoming Requests with a Replier

The *Replier* provides several operations that can be used to wait for and access the requests:

- **wait_for_requests()**, see [Waiting for Requests \(Section 23.2.5.1 on the facing page\)](#)
- **take_request()**, **take_requests()**, **read_request()**, and **read_requests()**, see [Reading and Taking Requests \(Section 23.2.5.2 on the facing page\)](#)
- **receive_request()** and **receive_requests()**, see [Receiving Requests \(Section 23.2.5.3 on page 895\)](#)

The **wait_for_requests()** operations are used to wait until requests arrive.

The **take_request()**, **take_requests()**, **read_request()**, and **read_requests()** operations access the requests, once they have arrived.

The **receive_request()** and **receive_requests()** operations are convenience functions that combine waiting for and accessing requests and are equivalent to calling the ‘wait’ operation followed by the corresponding **take_request()** or **take_requests()** operations.

Each of these operations has several variants, depending on the parameters that are passed in.

23.2.5.1 Waiting for Requests

Use the `wait_for_requests()` operation on the *Replier* to wait for requests. There are two variants of this operation, depending on the parameters that are passed in. All these variants block the calling thread until either there are replies or a timeout occurs.:

```
1. wait_for_requests (const DDS_Duration_t &max_wait)
2. wait_for_requests (int min_count, const DDS_Duration_t &max_wait)
```

The first variant (only passing in **max_wait**) blocks until one request is available or until **max_wait** time has passed, whichever comes first.

The second variant blocks until **min_count** number of requests are available or until **max_wait** time has passed.

Typically after waiting for requests, you will call **take_request**, **take_requests**, **read_request**, or **read_requests**, see [Sending Replies \(Section 23.2.6 on page 896\)](#).

23.2.5.2 Reading and Taking Requests

You can use the following four operations to access requests: **take_request**, **take_requests**, **read_request**, or **read_requests**.

As mentioned in [Summary of Replier Operations \(Section 23.2.4 on page 892\)](#), the difference between the ‘take’ operations (**take_request**, **take_requests**) and the ‘read’ operations (**read_request**, **read_requests**) is that ‘take’ operations remove the requests from the *Replier* cache. This means that future calls to **take_request**, **take_requests**, **read_request**, or **read_requests** will not get the same request again.

The **take_request** and **read_request** operations access a *single* reply, whereas the **take_requests** and **read_requests** can access a *collection* of replies.

There are two variants of the **take_request** and **read_request** operations, depending on the parameters that are passed in:

```
1. take_request (connext::Sample<TRequest> & request)
   read_request (connext::Sample<TRequest> & request)

2. take_request (connext::SampleRef<TRequest> request)
   read_request (connext::SampleRef<TRequest> request)
```

The first variant returns the request using a *Sample* container. The second variant uses a *SampleRef* container instead. A *SampleRef* can be used much like a *Sample*, but it holds *references* to the request data and *DDS SampleInfo*, so there is no additional copy. In contrast, using the *Sample* makes a copy of both the data and *DDS SampleInfo*.

The **take_requests** and **read_requests** operations access a collection of (one or more) requests in the *Replier* cache. These operations are convenient when you want to batch-process a set of requests.

The **take_requests** and **read_requests** operations return a *LoanedSamples* container that holds the requests. To increase performance, the *LoanedSamples* does not copy the request data. Instead it ‘loans’ the necessary resources from the *Replier*. The resources loaned by the *LoanedSamples* container must be eventually returned, either explicitly by calling the **return_loan()** operation on the *LoanedSamples* or through the destructor of the *LoanedSamples*.

There is only one variant of these operations:

```
1. take_requests (int max_samples = DDS_LENGTH_UNLIMITED)
   read_requests (int max_samples = DDS_LENGTH_UNLIMITED)
```

The returned container may contain up to **max_samples** number of requests.

23.2.5.3 Receiving Requests

The **receive_request()** operation is a shortcut that combines calls to **wait_for_requests()** and **take_request()**. Similarly, the **receive_requests()** operation combines **wait_for_requests()** and **take_requests()**.

There are two variants of the **receive_request()** operation:

```
1. receive_request (connext::Sample<TRequest> & request,
                  const DDS_Duration_t & max_wait)
2. receive_request (connext::SampleRef<TRequest> request,
                  const DDS_Duration_t & max_wait)
```

The **receive_request** operation blocks until either a request is received or a timeout occurs. The contents of the request are copied into the provided container (**request**). The first variant uses a *Sample* container, whereas the second variant uses a *SampleRef* container. A *SampleRef* can be used much like a *Sample*, but it holds *references* to the request data and *DDS SampleInfo*, so there is no additional copy. In contrast, using the *Sample* obtains a copy of both the data and the *DDS SampleInfo*.

There are two variants of the **receive_requests()** operation, depending on the parameters that are passed in:

```
1. receive_requests (const DDS_Duration_t & max_wait)
2. receive_requests (int min_request_count,
                  int max_request_count,
                  const DDS_Duration_t & max_wait)
```

The **receive_requests** operation blocks until one or more requests are available, or a timeout occurs.

The first variant (only passing in **max_wait**) blocks until one request is available or until **max_wait** time has passed, whichever comes first. The contents of the request are copied into a *LoanedSamples* container which is returned to the caller. An unlimited number of replies can be copied into the container.

The second variant blocks until **min_request_count** number of requests are available or until **max_wait** time has passed, whichever comes first. Up to **max_request_count** number of requests will be copied into a *LoanedSamples* container which is returned to the caller.

The resources for the *LoanedSamples* container must eventually be returned, either with **return_loan()** or through the *LoanedSamples* destructor.

23.2.6 Sending Replies

There are three variants for **send_reply()**, depending on the parameters that are passed in:

```
1. send_reply (const TReply & reply,
              const SampleIdentity_t & related_request_id)
2. send_reply (WriteSample<TReply> & reply,
              const SampleIdentity_t & related_request_id)
3. send_reply (WriteSampleRef<TReply> & reply,
              const SampleIdentity_t & related_request_id)
```

This operation sends a reply for a previous request. The related request ID can be retrieved from an existing request *Sample*.

The first variant is recommended if you do not need to change any of the default write parameters.

The other two variants allow you to set custom parameters for writing a reply. Unlike the *Requester*, where retrieving the sample ID for correlation is common, on the *Replier* side using a *WriteSample* or *WriteSampleRef* is only necessary when you need to overwrite the default write parameters. If that's not the case, use the first variant.

One reason to override the default write parameters is a multi-reply scenario in which a *Replier* generates more than one reply for a request. In this case, all the intermediate replies (all but the last reply) should be marked with the `INTERMEDIATE_REPLY_SEQUENCE_SAMPLE` bit-flag in the **flag** field within **WriteSample::info** or **WriteSampleRef::info**.

A *Requester* can detect if a reply is the last reply in a sequence of replies by seeing if `INTERMEDIATE_REPLY_SEQUENCE_SAMPLE` is NOT set in the **flag** field of **Sample::info** after receiving each reply.

23.3 SimpleRepliers

The *SimpleReplier* offers a simplified API to receive and process requests. The API is based on a user-provided object that implements the *SimpleReplierListener* interface. Requests are passed to the listener operation implemented by the user-provided object, which processes the request and returns a reply.

The *SimpleReplier* is recommended if each request generates a single reply and computing the reply can be done quickly with very little CPU resources and without calling any operations that may block the

processing thread. For example, looking something up in an internal memory-based data structure would be a good use case for using a *SimpleReplier*.

23.3.1 Creating a SimpleReplier

To create a *SimpleReplier* with the minimum set of parameters, you can use the basic constructor:

```
SimpleReplier (DDSDomainParticipant *participant,
               const std::string &service_name,
               SimpleReplierListener<TRequest, TReply> &listener)
```

To create a *SimpleReplier* with specific parameters, you may use a different constructor that receives a *SimpleReplierParams* structure:

```
SimpleReplier (const SimpleReplierParams<TRequest, TReply> &params)
```

23.3.2 Destroying a SimpleReplier

To destroy a *SimpleReplier* and free its resources use the destructor:

```
virtual ~SimpleReplier ()
```

23.3.3 Setting SimpleReplier Parameters

To change the *SimpleReplierParams* used to create a *SimpleReplier*, use the operations in [Table 23.5 Operations to Set SimpleReplier Parameters](#).

Table 23.5 Operations to Set SimpleReplier Parameters

Operation	Description
datareader_qos	Sets the quality of service of the reply DataReader.
datawriter_qos	Sets the quality of service of the reply DataWriter.
publisher	Sets a specific Publisher.
qos_profile	Sets a QoS profile for the entities in this replier.
reply_topic_name	Sets a specific reply topic name.
reply_type_support	Sets the type support for the reply type.
request_topic_name	Sets a specific request topic name.
request_type_support	Sets the type support for the request type.

Table 23.5 Operations to Set SimpleReplier Parameters

Operation	Description
service_name	Sets the service name the Replier offers and Requesters use to match.
subscriber	Sets a specific Subscriber.

23.3.4 Getting Requests and Sending Replies with a SimpleReplierListener

The `on_request_available()` operation on the *SimpleReplierListener* receives a request and returns a reply.

```
on_request_available(TRequest &request)
```

This operation gets called when a request is available. It should immediately return a reply. After calling `on_request_available()`, Connex DDS will call the operation `return_loan()` on the *SimpleReplierListener*; this gives the application-defined listener an opportunity to release any resources related to computing the previous reply.

```
return_loan(TReply &reply)
```

23.4 Accessing Underlying DataWriters and DataReaders

Both *Requester* and *Replier* entities have underlying DDS *DataWriter* and *DataReader* entities. These are created automatically when the *Requester* and *Replier* are constructed.

Accessing the *DataWriter* used by a *Requester* may be useful for a number of advanced use cases, such as:

- Finding matching subscriptions (e.g., *Replier* entities), see [Finding Matching Subscriptions \(Section 6.3.16.1 on page 309\)](#)
- Setting a *DataWriterListener*, see [Setting Up DataWriterListeners \(Section 6.3.4 on page 269\)](#)
- Getting *DataWriter* protocol or cache statuses, see [Statuses for DataWriters \(Section 6.3.6 on page 271\)](#)
- Flushing a data batch after sending a number of request samples, see [Flushing Batches of DDS Data Samples \(Section 6.3.9 on page 287\)](#)
- Modifying the QoS

Accessing the reply *DataReader* may be useful for a number of advanced use cases, such as:

- Finding matching publications (e.g., *Requester* entities), see [Navigating Relationships Among Entities \(Section 7.3.9 on page 489\)](#)

- Getting *DataReader* protocol or cache statuses, see [Checking DataReader Status and StatusConditions \(Section 7.3.5 on page 468\)](#) and [Statuses for DataReaders \(Section 7.3.7 on page 470\)](#).
- Modifying the QoS

To access these underlying objects:

```
RequestDataWriter * get_request_datawriter()  
RequestDataReader * get_request_datareader()  
ReplyDataWriter * get_reply_datawriter()  
ReplyDataReader * get_reply_datareader()
```


Part 5: RTI Secure WAN Transport

The material in this part of the manual is only relevant if you have installed *Secure WAN Transport*.

This feature is not installed as part of a Connex DDS package; it must be downloaded and installed separately. It is only available on specific architectures. See the *Secure WAN Transport Release Notes* and *Installation Guide* for details.

Secure WAN Transport is an optional package that enables participant discovery and data exchange in a secure manner over the public WAN. *Secure WAN Transport* enables Connex DDS to address the challenges in NAT traversal and authentication of all participants. By implementing UDP hole punching using the STUN protocol and providing security to channels by leveraging DTLS (Datagram TLS), you can securely exchange information between different sites separated by firewalls.

This section includes:

- [Introduction to Secure WAN Transport \(Section Chapter 24 on page 901\)](#)
- [Configuring RTI Secure WAN Transport \(Section Chapter 25 on page 914\)](#)

Chapter 24 Introduction to Secure WAN Transport

Secure WAN Transport provides transport plugins that can be used by developers of Connex DDS applications. These transport plugins allow Connex DDS applications running on private networks to communicate securely over a Wide-Area Network (WAN), such the internet. There are two primary components in the package which may be used independently or together: communication over Wide-Area Networks that involve Network Address Translators (NATs), and secure communication with support for peer authentication and encrypted data transport.

The Connex DDS core is transport-agnostic. Connex DDS offers three built-in transports: UDP/IPv4, UDP/IPv6, and inter-process shared memory. The implementation of NAT traversal and secure communication is done at the transport level so that the Connex DDS core is not affected and does not need to be changed, although there is additional on-the-wire traffic.

The basic problem to overcome in a WAN environment is that messages sent from an application on a private local-area network (LAN) appear to come from the LAN's router address, not from the internal IP address of the host running the application. This is due to the existence of a Network Address Translator (NAT) at the gateway. This does not cause problems for client/server systems because only the server needs to be globally addressable; it is only a problem for systems with peer-to-peer communication models, such as Connex DDS. Secure WAN Transport solves this problem, allowing communication between peers that are in separate LAN networks, using a UDP hole-punching mechanism based on the STUN protocol (IETF RFC 3489bis) for NAT traversal. This requires the use of an additional rendezvous server application, the RTI WAN Server.

Once the transport has enabled traffic to cross the NAT gateway to the WAN, it is flowing on network hardware that is shared (in some cases, over the public internet). In this context, it is important to consider the security of data transmission. There are three primary issues involved:

- Authenticating the communication peer (source or destination) as a trusted partner;
- Encrypting the data to hide it from other parties that may have access to the network;

- Validating the received data to ensure that it was not modified in transmission.

Secure WAN Transport addresses these problems by wrapping all RTPS-encoded data using the DTLS protocol (IETF RFC 4347), which is a variant of SSL/TLS that can be used over a datagram network-layer transport such as UDP. The security features of the WAN Transport may also be used on an untrusted local-area network with the Secure Transport.

In summary, the package includes two transports:

- The WAN Transport is for use on a WAN and includes security. It must be used with the WAN Server, a rendezvous server that provides the ability to discover public addresses and to register and look up peer addresses based on a unique WAN ID. The WAN Server is based on the STUN (Session Traversal Utilities for NAT) protocol [draft-ietf-behave-rfc3489bis], with some extensions. Once information about public addresses for the application and its peers has been obtained and connections have been initiated, the server is no longer required to maintain communication with a peer. (Note: security is disabled by default.)
- The Secure Transport is an alternate transport that provides security on an untrusted LAN. Use of the RTI WAN Server is not required.

Multicast communication is not supported by either of these transports.

This chapter provides a technical overview of:

- [WAN Traversal via UDP Hole-Punching \(Section 24.1 below\)](#)
- [WAN Locators \(Section 24.2 on page 907\)](#)
- [Datagram Transport-Layer Security \(DTLS\) \(Section 24.3 on page 908\)](#)
- [Certificate Support \(Section 24.4 on page 909\)](#)

For information on how to use Secure WAN Transport with your Connexx DDS application, see [Configuring RTI Secure WAN Transport \(Section Chapter 25 on page 914\)](#).

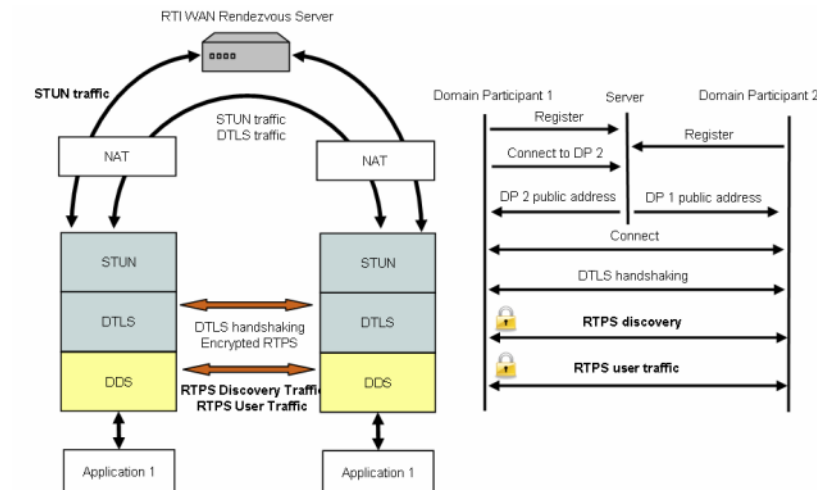
24.1 WAN Traversal via UDP Hole-Punching

In order to resolve the problem of communication across NAT boundaries, the WAN Transport implements a UDP hole-punching solution for NAT traversal [draft-ietf-behave-p2p-state]. This solution uses a rendezvous server, which provides the ability to discover public addresses, and to register and lookup peer addresses based on a unique WAN ID. This server is based on the STUN (Session Traversal Utilities for NAT) protocol [draft-ietf-behave-rfc3489bis], with some extensions. This protocol is a part of the solution used for standards-based voice over IP applications; similar technology has been used by systems such as Skype and has proven to be highly reliable. A key advantage of STUN is that it is based on UDP and therefore is able to preserve the real-time characteristics of the DDS Interoperability Wire Protocol.

Once information about public addresses for the application and its peers has been obtained, and connections have been initiated, the server is no longer required to maintain communication with a peer. However, if communication fails, possibly due to changes in dynamically-allocated addresses, the server will be needed to reopen new public channels.

[Figure 24.1 RTI WAN Transport Architecture below](#) shows the RTI WAN transport architecture.

Figure 24.1 RTI WAN Transport Architecture



24.1.1 Protocol Details

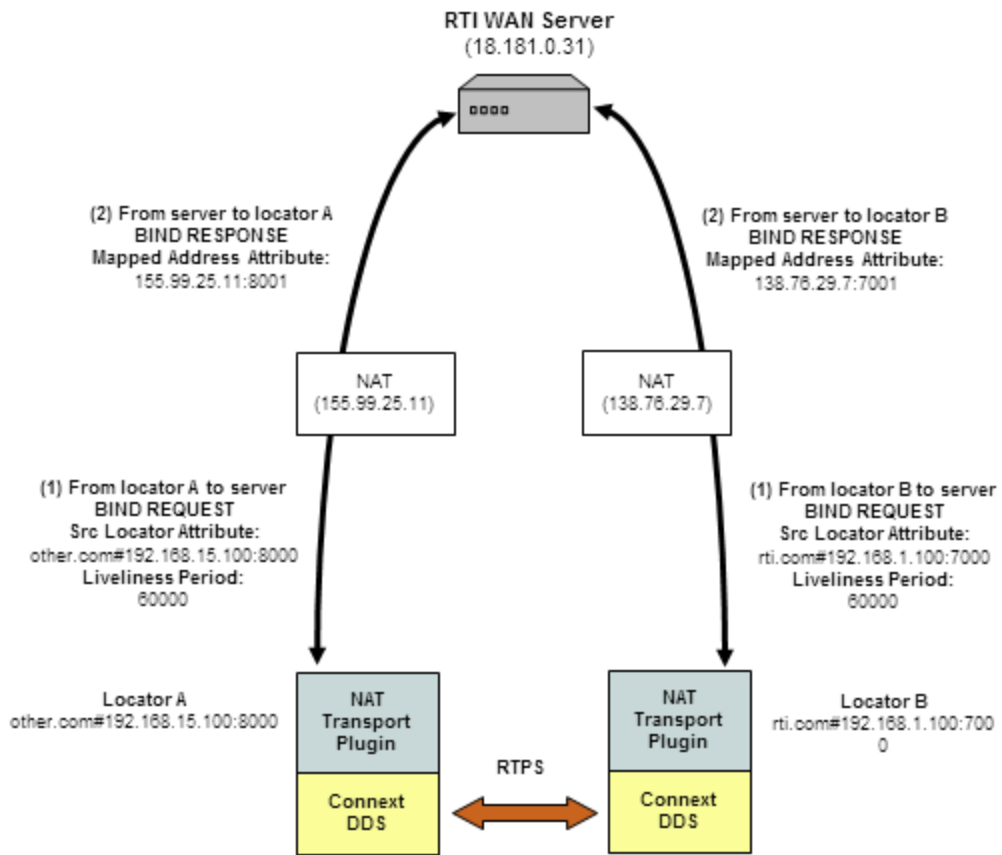
The UDP hole-punching algorithm implemented by the WAN transport has two different phases: registration and connection. This algorithm only works with cone or asymmetric NATs where the same public address/port is assigned to all the sessions with the same private address/port address.

- Registration Phase

The RTI WAN Server application runs on a machine that resides on the WAN network (i.e., not in a private LAN). It has to be globally accessible to LAN applications. It is started by a script and acts as a rendezvous point for LAN applications. During the registration phase, each transport locator is registered with the RTI WAN Server using a STUN binding request message.

The RTI WAN Server associates RTPS locators with their corresponding public IPv4 transport addresses (a combination of IP address and port) and stores that information in an internal table. [Figure 24.2 Registration Phase on the next page](#) illustrates the registration phase.

Figure 24.2 Registration Phase



- Connection Phase

The connection phase starts when locator A wants to establish a connection with locator B. Locator A obtains information about locator B via Connex DDS discovery traffic or the initial NDDS_DISCOVERY_PEERS list. To establish a connection with locator B, locator A sends a STUN connect request to the RTI WAN server. The server sends a STUN connect response to locator A, including information about the public IP transport address (IP address and port) of locator B. In parallel, the RTI WAN server contacts locator B using another STUN connect request to let it know that locator A wants to establish a connection with it.

When locator A receives the public IP address of locator B, it will try to contact B using two STUN binding request messages. The first message is sent to the public address of B and the second message is sent to the private address of B. The private address was obtained using the last 32 bits of the locator address of B. The STUN binding request message directed to the public transport address of B sent by locator A will open a hole in A's NAT to receive messages from B.

When locator B receives the public address of locator A, it will try to contact A sending a STUN binding request message to that public address. This message will open a hole in B's NAT to receive messages from A. When locator A receives the first STUN binding response from locator B, it starts sending RTPS traffic.

The connection phase includes two processes: the connect process ([Figure 24.3 Connect Process on the next page](#)) and the NAT hole punching process ([Figure 24.4 NAT Hole Punching Process on the next page](#)).

Figure 24.3 Connect Process

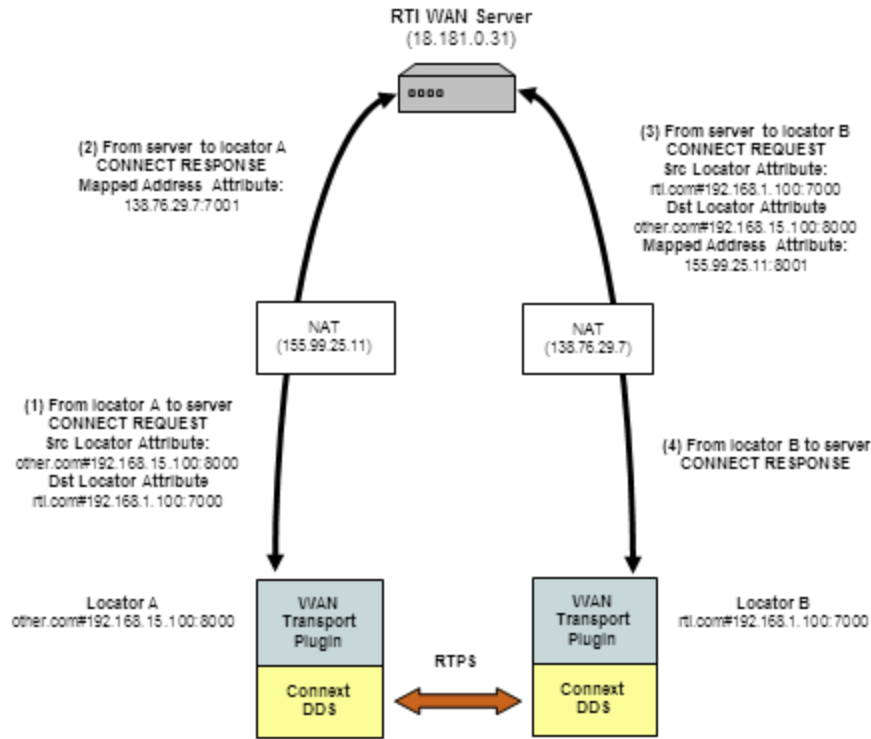
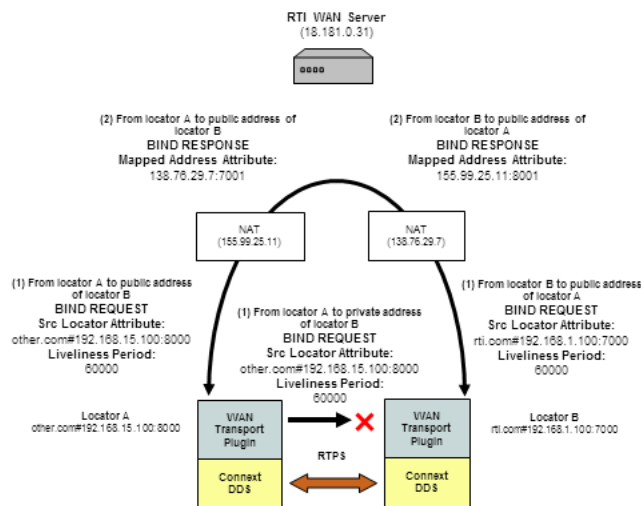


Figure 24.4 NAT Hole Punching Process

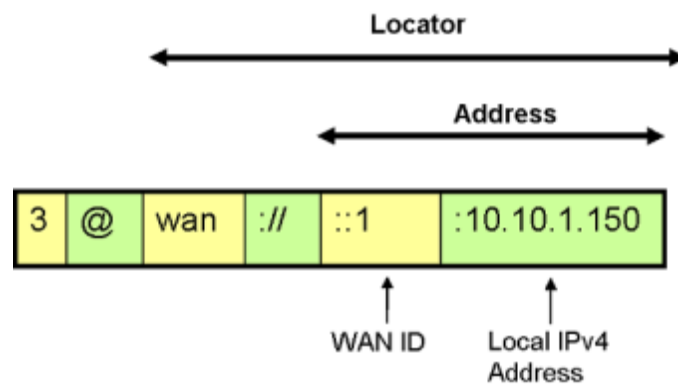


- STUN Liveliness

Finally, since bindings allocated by NAT expire unless refreshed, the clients (locators) must generate binding request messages for the server and other clients to refresh the bindings. The RTI STUN protocol implementation uses the attribute `LIVELINESS-PERIOD` in the STUN binding request to indicate the period in milliseconds at which a client will assert its liveliness. The WAN Server will remove a locator from its mapping table when the liveliness contract is not met. Likewise, a transport instance will remove a STUN connection with a locator when this locator does not assert its liveliness as indicated in the last binding request.

24.2 WAN Locators

The WAN transport does not use simple IP addresses to locate peers. A WAN transport locator consists of a WAN ID, which is an arbitrary 12-byte value, and a bottom 4-byte value that specifies a fallback local IPv4 address. Your peers list (`NDDS_DISCOVERY_PEERS`) must be configured to look for peers with locators of the form:



- The address is a 128-bit address in IPv6 notation.
- The "wan://" part specifies that the address is for the WAN transport.
- The next part, "::1", specifies the top 12 bytes of the address to be 11 zero bytes, followed by a byte with value 1 (this corresponds to the peer's WAN ID).
- The last part, "10.10.1.150" refers to the peers local IPv4 address, which will be used if the peers are on the same local network.

A *DomainParticipant* using the WAN transport will have to initialize the `DDS_DiscoveryQosPolicy`'s **initial_peers** field with the WAN locator addresses corresponding to the peers to which it wants to connect to. The value of **initial_peers** can be set using the environment variable `NDDS_DISCOVERY_PEERS` or the `NDDS_DISCOVERY_PEERS` configuration file. (See [Configuring the Peers List Used in Discovery \(Section 14.2 on page 711\)](#).)

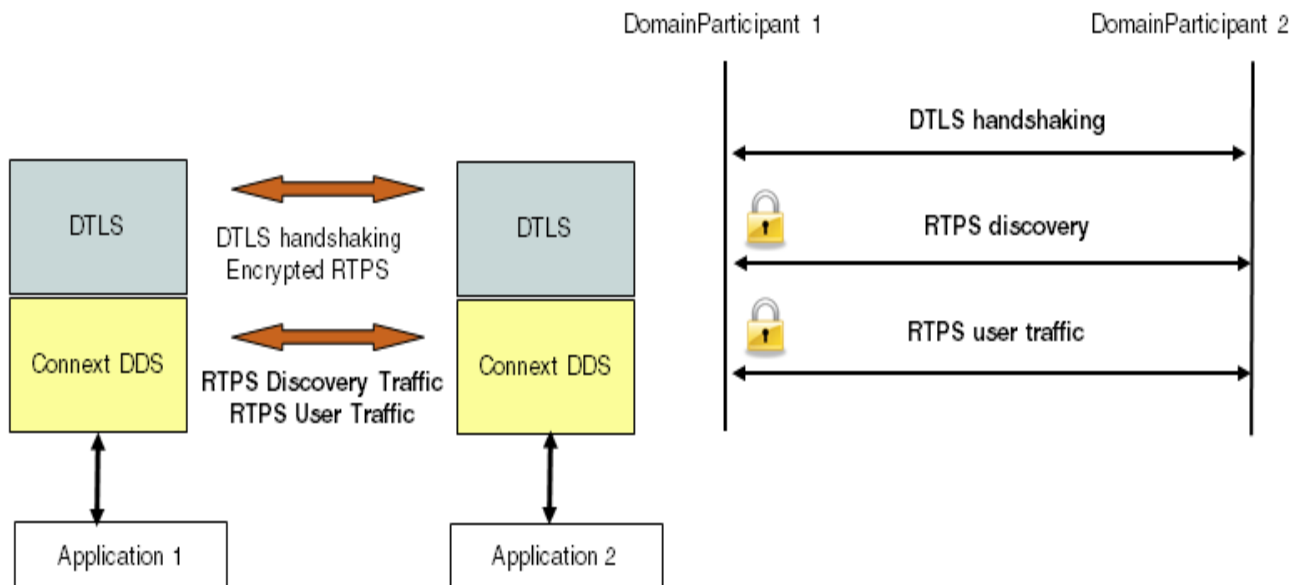
24.3 Datagram Transport-Layer Security (DTLS)

Data security is provided by wrapping all Connex DDS network traffic with the Datagram Transport Layer Security (DTLS) protocol (IETF RFC 4347). DTLS is a relatively recent variant of the mature SSL/TLS family of protocols which adds the capability to secure communication over a connectionless network-layer transport such as UDP. UDP is the preferred network layer transport for the DDS wire protocol RTPS, as well as for NAT traversal. Like SSL/TLS, the DTLS protocol provides capabilities for certificate-based authentication, data encryption, and message integrity. The protocol specifies a number of standard cryptographic algorithms that must be available; the base set is listed in the TLS 1.1 specification (IETF RFC 4346).

Secure protocol support is provided by the open source OpenSSL library, which has supported the DTLS protocol since the release of OpenSSL 0.9.8. Note however that many critical issues in DTLS were resolved by the OpenSSL 0.9.8f release. For more detailed information about available ciphers, certificate support, etc. please refer to the OpenSSL documentation. The DTLS protocol securely authenticates with each individual peer; as such, multicast communication is not supported by the Secure Transport. There is also a FIPS security-certified version of OpenSSL (OpenSSL-FIPS 1.1.1), but this does not yet support DTLS.

The Secure Transport protocol stack is similar to the Secure WAN transport stack, but without the STUN layer and server. See [DTLS Architecture \(Section Figure 24.5 below\)](#).

Figure 24.5 DTLS Architecture



24.3.1 Security Model

In order to communicate securely, an instance of the secure plugin requires: 1) a certificate authority (shared with all peers), 2) an identifying certificate which has been signed by the authority, 3) the private key associated with the public key contained in the certificate.

The Certificate Authority (CA) is specified by using a PEM format file containing its public key or by using a directory of PEM files following standard OpenSSL naming conventions. If a single CA file is used, it may contain multiple CA keys. In order to successfully communicate with a peer, the CA keys that are supplied must include the CA that has signed that peer's identifying certificate.

The identifying certificate is specified by using a PEM format file containing the chain of CAs used to authenticate the certificate. The identifying certificate must be signed by a CA. It will either be directly signed by a root CA (one of the CAs supplied above), by an authority whose certificate has been signed by the root CA, or by a longer chain of certificate authorities. The file must be sorted starting with the certificate to the highest level (root CA). If the certificate is directly signed by a root CA, then this file will only contain the root CA certificate followed by the identity certificate.

Finally, a private key is required. In order to avoid impersonation of an identity, this should be kept private. It can be stored in its own PEM file specified in one of the private key properties, or it can be appended to the certificate chain file.

One complication in the use of DTLS for communication by Connex DDS is that even though DTLS is a connectionless protocol, it still has client/server semantics. The *RTI Secure Transport* maps a bidirectional communication channel between two peer applications into a pair of unidirectional encrypted channels. Both peers are playing the part of a client (when sending data) and a server (when receiving).

24.3.2 Liveliness Mechanism

When a peer shuts down cleanly, the DTLS protocol ensures that resources are released. If a peer crashes or otherwise stops responding, a liveliness mechanism in the DTLS transport cleans up resources. You can configure the DTLS handshake retransmission interval and the connection liveliness interval.

24.4 Certificate Support

Cryptographic certificates are required to use the security features of the WAN transport. This section describes a mechanism to use the OpenSSL command line tool to generate a simple private certificate authority. For more information, see the manual page for the **openssl** tool (<http://www.openssl.org/docs/apps/openssl.html>) or the book, "*Network Security with OpenSSL*" by Viega, Messier, & Chandra (O'Reilly 2002), or other references on Public Key Infrastructure.

1. Initialize the Certificate Authority:
 - a. Create a copy of the openssl.cnf file and edit fields to specify the proper default names and paths.

- b. Create the required CA directory structure:

```
mkdir myCA
```

```
mkdir myCA/certs
```

```
mkdir myCA/private
```

```
mkdir myCA/newcerts
```

```
mkdir myCA/crl
```

```
touch myCA/index.txt
```

- c. Create a self-signed certificate and CA private key:

```
openssl req -nodes -x509 -days 1095 -newkey rsa:2048 \  
-keyout myCA/private/akey.pem -out myCA/cacert.pem \  
-config openssl.cnf
```

2. For each identifying certificate:

- a. You may want to create a copy of your customized openssl.cnf file with default identifying information to be used as a template for certificate request creation; the commands below refer to this file as **template.cnf**.
- b. Generate a certificate request and private key:

```
openssl req -nodes -new -newkey rsa:2048 -config template.cnf \  
-keyout peer1key.pem -out peer1req.pem
```

- c. Use the CA to sign the certificate request to generate certificate:

```
openssl ca -create_serial -config openssl.cnf -days 365 \  
-in peer1req.pem -out myCA/newcerts/peer1cert.pem
```

- d. Optionally, append the private key to the peer certificate:

```
cat myCA/newcerts/peer1cert.pem peer1key.pem \  
$>${private location}/ peer1.pem
```

24.5 License Issues

The OpenSSL toolkit stays under a dual license, i.e., both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

```

/* =====
 * Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)" *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without
 * prior written permission. For written permission, please contact
 * openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 * nor may "OpenSSL" appear in their names without prior written
 * permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 * acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 * =====
 *
 * This product includes cryptographic software written by Eric Young
 * (eay@cryptsoft.com). This product includes software written by Tim
 * Hudson (tjh@cryptsoft.com).
 *

```

```
*/
Original SSLeay License
-----
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to. The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code. The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given
 * attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 * must display the following acknowledgement:
 * "This product includes cryptographic software written by
 * Eric Young (eay@cryptsoft.com)"
 * The word 'cryptographic' can be left out if the routines from the
 * library
 * being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof)
 * from the apps directory (application code) you must include an
 * acknowledgement:
 * "This product includes software written by Tim Hudson
 * (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
```

```
* STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publicly available
* version or
* derivative of this code cannot be changed.  i.e. this code cannot
* simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.] */
```

Chapter 25 Configuring RTI Secure WAN Transport

The Secure WAN Transport package includes two transports:

- **The WAN Transport** is for use on a WAN and includes security.¹ It must be used with the WAN Server, a separate application that provides additional services needed for Connex DDS applications to communicate with each other over a WAN.
- **The Secure Transport** is an alternate transport that provides security on an untrusted LAN. Use of the RTI WAN Server is not required.

There are two ways in which these transports can be configured:

- By setting up predefined strings in the Property QoS Policy of the *DomainParticipant* (on UNIX, Solaris and Windows systems only). This process is described in [Setting Up a Transport with the Property QoS \(Section 25.2 on the next page\)](#).
- By instantiating a new transport ([Explicitly Instantiating a WAN or Secure Transport Plugin \(Section 25.5 on page 930\)](#)) and then registering it with the *DomainParticipant*, see [Installing Additional Builtin Transport Plugins with register_transport\(\) \(Section 15.7 on page 765\)](#) (not available in Java API).

Refer to the API Reference HTML documentation for details on these two approaches.

25.1 Example Applications

A simple example is available to show how to configure the WAN transport. It includes example settings to enable communication over WAN, and optional settings to enable security (along with

¹Security is disabled by default.

example certificate files to use for secure communication). The example is located in `<path to examples>a/connext_dds/<language>/hello_world_wan`.

As seen in the example, you can configure the properties of either transport by setting the appropriate name/value pairs in the *DomainParticipant*'s PropertyQoS, as described in [Setting Up a Transport with the Property QoS \(Section 25.2 below\)](#). This will cause Connex DDS to dynamically load the WAN or Secure Transport libraries at run time and then implicitly create and register the transport plugin.

Another way to use the WAN or Secure transports is to explicitly create the plugin and use `register_transport()` to register the transport with Connex DDS (see [Installing Additional Builtin Transport Plugins with register_transport\(\) \(Section 15.7 on page 765\)](#)). This way is *not* shown in the example. See [Explicitly Instantiating a WAN or Secure Transport Plugin \(Section 25.5 on page 930\)](#).

25.2 Setting Up a Transport with the Property QoS

The [PROPERTY QoS Policy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#) allows you to set up name/value pairs of data and attach them to an entity, such as a *DomainParticipant*. This will cause Connex DDS to dynamically load the WAN or Secure Transport libraries at run time and then implicitly create and register the transport plugin.

Please refer to [Setting Builtin Transport Properties with the PropertyQoS Policy \(Section 15.6 on page 748\)](#).

To assign properties, use the `add_property()` operation:

```
DDS_ReturnCode_t DDSPropertyQoSPolicyHelper::add_property
    (DDS_PropertyQoSPolicy policy,
     const char * name,
     const char * value,
     DDS_Boolean propagate)
```

For more information on `add_property()` and the other operations in the `DDSPropertyQoSPolicyHelper` class, please see [Table 6.57 PropertyQoS Policy Helper Operations](#), as well as the API Reference HTML documentation.

The 'name' part of the name/value pairs is a predefined string, described in [WAN Transport Properties \(Section 25.3 on page 917\)](#) and [Secure Transport Properties \(Section 25.4 on page 925\)](#).

Here are the basic steps, taken from the example Hello World application (for details, please see the example application.)

^aSee [Paths Mentioned in Documentation \(Section on page xxxviii\)](#).

1. Get the default *DomainParticipant* QoS from the *DomainParticipantFactory*.

```
DDSDomainParticipantFactory::get_instance()->
  get_default_participant_qos(participant_qos);
```

2. Disable the builtin transports.

```
participant_qos.transport_builtin.mask =
  DDS_TRANSPORTBUILTIN_MASK_NONE;
```

3. Set up the *DomainParticipant*'s Property QoS.

- a. Load the plugin.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
  "dds.transport.load_plugins",
  "dds.transport.wan_plugin.wan",
  DDS_BOOLEAN_FALSE);
```

- b. Specify the transport plugin library.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
  "dds.transport.wan_plugin.wan.library",
  "libnddstransportwan.so",
  DDS_BOOLEAN_FALSE);
```

- c. Specify the transport's 'create' function.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
  "dds.transport.wan_plugin.wan.create_function"
  "NDDS_Transport_WAN_create",
  DDS_BOOLEAN_FALSE);
```

- d. Specify the WAN Server and instance ID.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property
  "dds.transport.wan_plugin.wan.server",
  "192.168.1.1",
  DDS_BOOLEAN_FALSE);
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
```

```
"dds.transport.wan_plugin.wan.transport_instance_id",
1,
DDS_BOOLEAN_FALSE);
```

e. Specify any other properties, as needed.

4. Create the `DomainParticipant`, using the modified QoS.

```
participant = DDSTheParticipantFactory->create_participant (
    domainId,
    participant_qos,
    NULL /* listener */,
    DDS_STATUS_MASK_NONE);
```

Property changes should be made before the transport is loaded: either before the `DomainParticipant` is enabled, before the first `DataWriter/DataReader` is created, or before the builtin topic reader is looked up, whichever one happens first.

25.3 WAN Transport Properties

Table 25.1 Properties for `NDDS_Transport_WAN_Property_t` lists the properties that you can set for the WAN Transport.

Table 25.1 Properties for `NDDS_Transport_WAN_Property_t`

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
<code>dds.transport.load_plugins</code> <i>(note: this does not take a prefix)</i>	<p>Required</p> <p>Comma-separated strings indicating the prefix names of all plugins that will be loaded by Connex DDS. You will use this string as the prefix to the property names.</p> <p>For example: "dds.transport.WAN.wan1". (This assumes you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with <code>dds.transport.load_plugins</code>.)</p> <p>This prefix must begin with 'dds.transport.'</p> <p>Note: You can load up to 8 plugins.</p>

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for `NDDS_Transport_WAN_Property_t`

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
library	<p>Required</p> <p>Must set to "libnddstransportwan.so" (for UNIX/Solaris systems) or "nddstransportwan.dll" (for Windows system).</p> <p>This library and the dependent OpenSSL libraries need to be in your library search path (pointed to by the environment variable <code>LD_LIBRARY_PATH</code> on UNIX/Solaris systems, <code>Path</code> on Windows systems, <code>LIBPATH</code> on AIX systems, <code>DYLD_LIBRARY_PATH</code> on Mac OS systems).</p>
create_function	<p>Required</p> <p>Must be "NDDS_Transport_WAN_create"</p>
aliases	<p>Used to register the transport plugin returned by NDDS_Transport_WAN_create() (as specified by <code><WAN_prefix>.create_function</code>) to the <i>DomainParticipant</i>. Aliases should be specified as a comma-separated string, with each comma delimiting an alias.</p> <p>If it is not specified, the prefix is used as the default alias for the plugin.</p>
verbosity	<p>Specifies the verbosity of log messages from the transport.</p> <p>Possible values:</p> <ul style="list-style-type: none"> -1: silent 0 (default): errors only 1: errors and warnings 2: local status 5 or higher: all messages
parent.parent.address_bit_count	<p>Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8.</p>
parent.parent.properties_bitmap	<p>A bitmap that defines various properties of the transport to the Connexx DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connexx DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.</p>

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for `NDDS_Transport_WAN_Property_t`

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
parent.parent.gather_send_buffer_count_max	<p>Specifies the maximum number of buffers that Connex DDS can pass to the <code>send()</code> function of the transport plugin.</p> <p>The transport plugin <code>send()</code> API supports a gather-send concept, where the <code>send()</code> call can take several discontinuous buffers, assemble and send them in a single message. This enables Connex DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.</p> <p>However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connex DDS from trying to gather too many buffers into a send call for the transport plugin.</p> <p>Connex DDS requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is defined as <code>NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN</code>.</p>
parent.parent.message_size_max	<p>The maximum size of a message in bytes that can be sent or received by the transport plugin.</p> <p>This value must be set before the transport plugin is registered, so that Connex DDS can properly use the plugin.</p>
parent.parent.allow_interfaces	<p>A list of strings, each identifying a range of interface addresses.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>If the list is non-empty, this "white" list is applied before the <code>parent.parent.deny_interfaces</code> (Section on the facing page) list.</p> <p>It is up to the transport plugin to interpret the list of strings passed in. Usually this interpretation will be consistent with <code>NDDS_Transport_String_To_Address_Fcn_cEA()</code>.</p> <p>This property is not interpreted by the Connex DDS core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.</p> <p>You must manage the memory of the list. The memory may be freed after the <i>DomainParticipant</i> is enabled.</p>

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for NDDS_Transport_WAN_Property_t

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
parent.parent.deny_interfaces	<p>A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of these interfaces.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>This "black" list is applied after the parent.parent.allow_interfaces (Section on the previous page) list and filters out the interfaces that should <i>not</i> be used.</p> <p>It is up to the transport plugin to interpret the list of strings passed in. Usually this interpretation will be consistent with <code>NDDS_Transport_String_To_Address_Fcn_cEA()</code>.</p> <p>This property is not interpreted by the Connexx DDS core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user.</p> <p>You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</p>
parent.send_socket_buffer_size	<p>Size in bytes of the send buffer of a socket used for sending. On most operating systems, <code>setsockopt()</code> will be called to set the <code>SO_SNDBUF</code> to the value of this parameter.</p> <p>This value must be greater than or equal to parent.parent.message_size_max (Section on the previous page).</p> <p>The maximum value is operating system-dependent.</p> <p>If <code>NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT</code>, then <code>setsockopt()</code> (or equivalent) will not be called to size the send buffer of the socket.</p>
parent.recv_socket_buffer_size	<p>Size in bytes of the receive buffer of a socket used for receiving.</p> <p>On most operating systems, <code>setsockopt()</code> will be called to set the <code>SO_RCVBUF</code> to the value of this parameter.</p> <p>This value must be greater than or equal to parent.parent.message_size_max (Section on the previous page). The maximum value is operating system-dependent.</p> <p>If <code>NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT</code>, then <code>setsockopt()</code> (or equivalent) will not be called to size the receive buffer of the socket.</p>
parent.unicast_enabled	<p>Allows the transport plugin to use unicast UDP for sending and receiving. By default, it will be turned on. Also by default, it will use all the allowed network interfaces that it finds up and running when the plugin is instanced.</p>

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for NDDS_Transport_WAN_Property_t

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
parent.ignore_loopback_interface	<p>Prevents the transport plugin from using the IP loopback interface. Three values are allowed:</p> <p>0: Enable local traffic via this plugin. This plugin will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system.</p> <p>1: Disable local traffic via this plugin. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plugin like Shared Memory instead of the IP loopback.</p>
parent.ignore_nonrunning_interfaces	<p>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.</p> <p>The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged.</p> <p>Two values are allowed:</p> <p>0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.</p> <p>1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.</p>
parent.no_zero_copy	<p>Prevents the transport plugin from doing a zero copy.</p> <p>By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application.</p> <p>The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the H/W, device driver, or the OS to allow the zero copy feature to work for your application, you may have no choice but to turn off zero copy use.</p> <p>By default this is set to 0, so Connnext DDS will use the zero-copy API if offered by the OS.</p>

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for NDDS_Transport_WAN_Property_t

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
parent.send_blocking	<p>Controls the blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.</p> <p>Two values are defined:</p> <ul style="list-style-type: none"> • NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS: Sockets are blocking (default socket options for Operating System). • NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER: Sockets are modified to make them non-blocking. THIS IS NOT A SUPPORTED CONFIGURATION AND MAY CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.
parent.transport_priority_mask	<p>Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low/high to define the mapping from DDS transport priority to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket.</p> <p>For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high.</p> <p>If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.</p>
parent.transport_priority_mapping_low	<p>Sets the low and high values of the output range to IPv4 TOS.</p> <p>These values are used in conjunction with transport_priority_mask to define the mapping from DDS transport priority to the IPv4 TOS field. Defines the low and high values of the output range for scaling.</p> <p>Note that IPv4 TOS is generally an 8-bit value.</p>
parent.transport_priority_mapping_high	
enable_security	Required if you want to use security.
recv_decode_buffer_size	Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data. The minimum value for this property is parent.parent.message_size_max (Section on page 919).
port_offset	Port offset to allow coexistence with non-secure UDP transport.
dtls_handshake_resend_interval	DTLS handshake retransmission interval in milliseconds

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for `NDDS_Transport_WAN_Property_t`

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
dtls_connection_liveliness_interval	<p>Liveliness interval (multiple of resend interval)</p> <p>The connection will be dropped if no message from the peer is received in this amount of time. This enables cleaning up state for peers that are no longer responding. A secure keep-alive message will be sent every half-interval if no other sends have occurred for a given DTLS connection during that time.</p> <p>Default: 60 ms</p>
tls.verify.ca_file	<p>A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for <code>SSL_load_verify_locations</code> for more information.</p> <p>If you want to use security, tls.verify.ca_file (Section above) or tls.verify.ca_path (Section below) must be specified; both may be specified.</p>
tls.verify.ca_path	<p>A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format, and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for <code>SSL_CTX_load_verify_locations</code> for more information.</p> <p>If you want to use security, tls.verify.ca_file (Section above) or tls.verify.ca_path (Section above) must be specified; both may be specified.</p>
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.verify_peer	If non-zero, use mutual authentication when performing TLS hand-shake (default). If zero, only the reader side will present a certificate, which will be verified by the writer side.
tls.verify.callback	<p>This can be set to one of three values:</p> <ul style="list-style-type: none"> • "default" selects <code>NDDS_Transport_TLS_default_verify_callback()</code> • "verbose" selects <code>NDDS_Transport_TLS_verbose_verify_callback()</code> • "none" requests no callback be registered
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for <code>SSL_set_cipher_list</code> for more information on the format of this string.

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 25.1 Properties for NDDS_Transport_WAN_Property_t

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
tls.cipher.dh_param_files	List of available Diffie-Hellman (DH) key files. For example: "foo.h:2048,bar.h:1024" means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 2048, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 1024
tls.cipher.engine_id	String ID of OpenSSL cipher engine to request.
tls.identity.certificate_chain_file	Required if you want to use security. A string that specifies the name of a file containing an identifying certificate chain (in PEM format). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If no private key is specified, this file will be used to load a non-RSA private key.
tls.identity.private_key_password	A string that specifies the password for private key.
tls.identity.private_key_file	A string that specifies that name of a file containing private key (in PEM format). If no private key is specified (all values are NULL), this value will default to the same file as the specified certificate chain file.
tls.identity.rsa_private_key_file	A string that specifies that name of a file containing an RSA private key (in PEM format).
transport_instance_id[0] to [NDDS_TRANSPORT_ WAN_TRANSPORT_ INSTANCE_ID_LENGTH ^b]	Required A set of comma-separated values to specify the elements of the array. This value must be unique for all transport instances communicating with the same WAN Rendezvous Server. If less than the full array is specified, it will be right-aligned. For example, the string "01,02" results in the array being set to: {0,0,0,0,0,0,0,0,0,1,2}
interface_address	Locator, as a string
server	Required Server locator, as a string.

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

^bNDDS_TRANSPORT_WAN_TRANSPORT_INSTANCE_ID_LENGTH = 12

Table 25.1 Properties for NDDS_Transport_WAN_Property_t

Property Name (prefix with 'dds.transport.WAN.wan1.' ¹)	Property Value Description
server_port	Server port number.
stun_retransmission_interval	STUN request messages requiring a response are resent with this interval. The interval is doubled after each retransmission. Specified in msec.
stun_number_of_retransmissions	Maximum number of times STUN messages are resent unless a response is received.
stun_liveliness_period	Period at which messages are sent to peers to keep NAT holes open; and to the WAN server to refresh bound ports. Specified in msec.

25.4 Secure Transport Properties

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t lists the properties that you can set for the Secure Transport.

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t

Property Name (prefix with 'dds.transport.DTLS.dtls1' ^b)	Property Value Description
dds.transport.load_plugins (note: this does not take a prefix)	<p>Required</p> <p>Comma-separated strings indicating the prefix names of all plugins that will be loaded by Connex DDS. You will use this string as the prefix to the property names.</p> <p>For example: "dds.transport.DTLS.dtls1". (This assumes you used used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins.)</p> <p>This prefix must begin with 'dds.transport.'</p> <p>Note: you can load up to 8 plugins.</p>

¹ Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

^b Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t

Property Name (prefix with 'dds.transport.DTLS.dtls1' ^a)	Property Value Description
library	<p>Only required if linking dynamically</p> <p>Must set to "libnndstransporttls.so" (for UNIX/Solaris) or "nndstransporttls.dll" (for Windows). This library and the dependent OpenSSL libraries must be in your library search path (pointed to by the environment variable LD_LIBRARY_PATH on UNIX/Solaris systems, Path on Windows systems, LIBPATH on AIX systems, DYLD_LIBRARY_PATH on Mac OS systems).</p>
create_function	<p>Only required if linking dynamically</p> <p>Must be "NDDS_Transport_DTLS_create"</p>
create_function_ptr	<p>Only required if linking statically</p> <p>Defines the function pointer to the DTLS Transport Plugin creation function. Used for loading the DTLS Transport plugin statically.</p> <p>Must be set to the NDDS_Transport_DTLS_create function pointer.</p>
aliases	<p>Used to register the transport plugin returned by NDDS_Transport_DTLS_create() (as specified by <DTLS_prefix>.create_function) to the DomainParticipant. Aliases should be specified as comma-separated strings, with each comma delimiting an alias. If it is not specified, the prefix is used as the default alias for the plugin.</p>
network_address	<p>The network address at which to register this transport plugin.</p> <p>The least significant transport_in.property.address_bit_count will be truncated. The remaining bits are the network address of the transport plugin.</p> <p>This value overwrites the value returned by the output parameter in NDDS_Transport_create_plugin function as specified in "<DTLS_prefix>.create_function".</p>
verbosity	<p>Specifies the verbosity of log messages from the transport.</p> <p>Possible values:</p> <ul style="list-style-type: none"> -1: silent 0 (default): errors only 1: errors and warnings 2: local status 5 or higher: all messages
parent.address_bit_count	<p>Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8.</p>

^a Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t

Property Name (prefix with 'dds.transport.DTLS.dtls1' ^a)	Property Value Description
parent.properties_bitmap	A bitmap that defines various properties of the transport to the Connex DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connex DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.
parent.gather_send_buffer_count_max	Specifies the maximum number of buffers that Connex DDS can pass to the transport plugin's send () function.
parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plugin. Maximum value: 16384.
parent.allow_interfaces	<p>A list of strings, each identifying a range of interface addresses.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>If the list is non-empty, this "white" list is applied before the parent.deny_interfaces (Section below) list.</p> <p>You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</p>
parent.deny_interfaces	<p>A list of strings, each identifying a range of interface addresses.</p> <p>Interfaces should be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>This "black" list is applied after the parent.allow_interfaces (Section above) list and filters out the interfaces that should not be used.</p> <p>You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</p>
send_socket_buffer_size	Size in bytes of the send buffer of a socket used for sending.
recv_socket_buffer_size	Size in bytes of the receive buffer of a socket used for sending.
ignore_loopback_interface	Prevents the Transport Plugin from using the IP loopback interface.

^a Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t

Property Name (prefix with 'dds.transport.DTLS.dtls1' ^a)	Property Value Description
ignore_nonrunning_interfaces	<p>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.</p> <p>The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged.</p> <p>Two values are allowed:</p> <p>0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.</p> <p>1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.</p>
transport_priority_mask	Mask for use of transport priority field.
transport_priority_mapping_low	Low and high values of output range to IPv4 TOS.
transport_priority_mapping_high	
recv_decode_buffer_size	Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data. The minimum value for this property is parent.message_size_max (Section on the previous page).
port_offset	Port offset to allow coexistence with non-secure UDP transport.
dtls_handshake_resend_interval	DTLS handshake retransmission interval in milliseconds
dtls_connection_liveliness_interval	<p>Liveliness interval (multiple of resend interval)</p> <p>The connection will be dropped if no message from the peer is received in this amount of time. This enables cleaning up state for peers that are no longer responding. A secure keep-alive message will be sent every half-interval if no other sends have occurred for a given DTLS connection during that time.</p> <p>Default: 60 ms</p>

^a Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t

Property Name (prefix with 'dds.transport.DTLS.dtls1' ^a)	Property Value Description
tls.verify.ca_file	A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information. tls.verify.ca_file (Section above) or tls.verify.ca_path (Section below) must be specified; both may be specified.
tls.verify.ca_path	A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format, and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_load_verify_locations for more information. tls.verify.ca_file (Section above) or tls.verify.ca_path (Section above) must be specified; both may be specified.
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.verify_peer	If non-zero, use mutual authentication when performing TLS hand- shake (default). If zero, only the reader side will present a certificate, which will be verified by the writer side.
tls.verify.callback	This can be set to one of three values: "default" selects NDDS_Transport_TLS_default_verify_callback() "verbose" selects NDDS_Transport_TLS_verbose_verify_callback() "none" requests no callback be registered
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string.
tls.cipher.dh_param_files	List of available Diffie-Hellman (DH) key files. For example: "foo.h:2048,bar.h:1024" means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 2048, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 1024
tls.cipher.engine_id	String ID of OpenSSL cipher engine to request.

^a Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 25.2 Properties for NDDS_Transport_DTLS_Property_t

Property Name (prefix with 'dds.transport.DTLS.dtls1' ^a)	Property Value Description
tls.identity.certificate_chain_file	RequiredA string that specifies the name of a file containing an identifying certificate chain (in PEM format). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If no private key is specified, this file will be used to load a non-RSA private key.
tls.identity.private_key_password	A string that specifies the password for private key.
tls.identity.private_key_file	A string that specifies that name of a file containing private key (in PEM format). If no private key is specified (all values are NULL), this value will default to the same file as the specified certificate chain file.
tls.identity.rsa_private_key_file	A string that specifies that name of a file containing an RSA private key (in PEM format).

25.5 Explicitly Instantiating a WAN or Secure Transport Plugin

As described on [Page 914](#), there are two ways to instantiate a transport plugin. This section describes the mechanism that includes calling `NDDSTransportSupport::register_transport()`. (The other way is to use the Property QoS mechanism, described in [Setting Up a Transport with the Property QoS \(Section 25.2 on page 915\)](#)).

Notes:

- This way of instantiating a transport is not supported in the Java API. If you are using Java, use the Property QoS mechanism, described in [Setting Up a Transport with the Property QoS \(Section 25.2 on page 915\)](#).
- To use this mechanism, there are **extra libraries that you must link into your program and an additional header file** that you must include. Please see the [Additional Header Files and Include Directories \(Section 25.5.1 on the next page\)](#) and [Additional Libraries \(Section 25.5.2 on the next page\)](#) for details.

To instantiate a WAN or Secure Transport prior to explicitly registering it with `NDDSTransportSupport::register_transport()`, use one of the following functions:

```
NDDS_Transport_Plugin* NDDS_Transport_WAN_new (
    const struct NDDS_Transport_WAN_Property_t * property_in)
```

^a Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

```

NDDS_Transport_Plugin* NDDS_Transport_DTLS_new (
    const struct NDDS_Transport_DTLS_Property_t * property_in)

```

See the API Reference HTML documentation for details on these functions.

25.5.1 Additional Header Files and Include Directories

- To use the Secure WAN Transport API, you must include an extra header file (in addition to those in [Table 9.1 Header Files to Include for Connex DDS \(All Architectures\)](#)).

```
#include "ndds/ndds_transport_secure_wan.h"
```

Assuming that Secure WAN Transport is installed in the same directory as Connex DDS (see [Table 9.2 Include Paths for Compilation \(All Architectures\)](#)), no additional include paths need to be added for the Secure WAN Transport API. If this is not the case, you will need to specify the appropriate include path.

- If you want to access OpenSSL data structures, add the OpenSSL include directory, `<openssl install dir>/<arch>/include`, and include the OpenSSL headers *before* `ndds_transport_secure_wan.h`:

```

#include <openssl/ssl.h>

#include <openssl/x509.h> (if accessing certificate functions)

etc.

```

On Windows systems, if you are loading statically: you should also include the OpenSSL file, `applink.c`, in your application. It can be found in the OpenSSL include directory, or included as `<openssl/applink.c>`.

25.5.2 Additional Libraries

To use the Secure WAN Transport API, you must link in additional libraries, which are listed in the [RTI Connex DDS Core Libraries Platform Notes](#) (in the appropriate section for your architecture). Refer to [Required Libraries \(Section 9.3.1 on page 625\)](#) for the differences between shared and static libraries.

25.5.3 Compiler Flags

No additional compiler flags are required.

Part 6: RTI Persistence Service

Persistence Service is only available with the Connex DDS Professional, Basic, and Evaluation packages types.

The material in this part of the manual describes *Persistence Service*. It saves DDS data samples so they can be delivered to subscribing applications that join the system at a later time—even if the publishing application has already terminated.

This section includes:

- [Introduction to RTI Persistence Service \(Section Chapter 26 on page 933\)](#)
- [Configuring Persistence Service \(Section Chapter 27 on page 934\)](#)
- [Running RTI Persistence Service \(Section Chapter 28 on page 962\)](#)
- [Administering Persistence Service from a Remote Location \(Section Chapter 29 on page 966\)](#)
- [Advanced Persistence Service Scenarios \(Section Chapter 30 on page 972\)](#)

Chapter 26 Introduction to RTI Persistence Service

Persistence Service is a Connext DDS application that saves DDS data samples to transient or permanent storage, so they can be delivered to subscribing applications that join the system at a later time—even if the publishing application has already terminated.

Persistence Service runs as a separate application; you can run it on the same node as the publishing application, the subscribing application, or some other node in the network.

When configured to run in PERSISTENT mode, Persistence Service can use the filesystem or a relational database that provides an ODBC driver. For each persistent topic, it collects all the data written by the corresponding persistent *DataWriters* and stores them into persistent storage. See the *RTI Persistence Service Release Notes* for the list of platforms and relational databases that have been tested.

When configured to run in TRANSIENT mode, Persistence Service stores the data in memory.

The following chapters assume you have a basic understanding of DDS terms such as *DomainParticipants*, *Publishers*, *DataWriters*, *Topics*, and Quality of Service (QoS) policies. For an overview of DDS terms, please see [Data-Centric Publish-Subscribe Communications \(Section Chapter 2 on page 10\)](#). You should also have already read [Mechanisms for Achieving Information Durability and Persistence \(Section Chapter 12 on page 675\)](#).

Chapter 27 Configuring Persistence Service

To use Persistence Service:

1. Modify your Connex DDS applications.
 - The [DURABILITY QosPolicy \(Section 6.5.7 on page 368\)](#) controls whether or not, and how, published DDS samples are stored by Persistence Service for delivery to late-joining *DataReaders*. See [Data Durability \(Section 12.5 on page 692\)](#).
 - For each *DataWriter* whose data must be stored, set the Durability QosPolicy's *kind* to `DDS_PERSISTENT_DURABILITY_QOS` or `DDS_TRANSIENT_DURABILITY_QOS`.
 - For each *DataReader* that needs to receive stored data, set the Durability QosPolicy's *kind* to `DDS_PERSISTENT_DURABILITY_QOS` or `DDS_TRANSIENT_DURABILITY_QOS`.
 - Optionally, modify the [DURABILITY SERVICE QosPolicy \(Section 6.5.8 on page 372\)](#), which can be used to configure Persistence Service.

By default, the History and ResourceLimits QosPolicies for a Persistence Service *DataReader* (PRSTDataReader) and Persistence Service *DataWriter* (PRSTDataWriter) with topic 'A' will be configured using the values specified in the XML file (unless you use the tag `<use_durability_service>` in the persistence group definition, see [Creating Persistence Groups \(Section 27.8 on page 947\)](#)). Setting the `<use_durability_service>` tag to true will cause the History and ResourceLimits QosPolicies for a PRSTDataReader and PRSTDataWriter to be configured using the [DURABILITY SERVICE QosPolicy \(Section 6.5.8 on page 372\)](#) of the first-discovered *DataWriter* publishing 'A'. (For more information on the PRSTDataReader and PRSTDataWriter, see [RTI Persistence Service \(Section 12.5.1 on page 692\)](#).)

2. Create a configuration file or edit an existing file, as described in [XML Configuration File \(Section 27.2 on the facing page\)](#).
3. Start Persistence Service with your configuration file, as described in [Starting Persistence Service \(Section 28.1 on page 962\)](#). You can start it on either application's node, or even an entirely different node (provided that node is included in one of the applications' NDDS_DISCOVERY_PEERS lists).

27.1 How to Load the Persistence Service XML Configuration

Persistence Service loads its XML configuration from multiple locations. This section presents the various approaches, listed in load order.

The first three locations only contain QoS Profiles and are inherited from Connex DDS (see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#)).

- \$NDDSHOME/resource/xml/NDDS_QOS_PROFILES.xml

This file contains the DDS default QoS values; it is loaded automatically if it exists. (*First to be loaded.*)

- File specified in the NDDS_QOS_PROFILES Environment Variable

The files (or XML strings) separated by semicolons referenced in this environment variable are loaded automatically.

- *<working directory>*/USER_QOS_PROFILES.xml

This file is loaded automatically if it exists.

The next locations are specific to Persistence Service.

- *<NDDSHOME>*/resource/xml/RTI_PERSISTENCE_SERVICE.xml

This file contains the default Persistence Service configurations; it is loaded if it exists. There are two default configurations: **default** and **defaultDisk**. The **default** configuration persists all the topics into memory. The **defaultDisk** configuration persists all the topics into files located in the current working directory.

- *<working directory>*/USER_PERSISTENCE_SERVICE.xml

This file is loaded automatically if it exists.

- File specified using the command line option, **-cfgFile**

The command-line option **-cfgFile** (see [Table 28.1 Persistence Service Command-Line Options](#)) can be used to specify a configuration file.

27.2 XML Configuration File

The configuration file uses XML format. Let's look at a very basic configuration file, just to get an idea of its contents. You will learn the meaning of each line as you read the rest of this section:

- [QoS Configuration \(Section 27.3 on page 939\)](#)
- [Configuring the Persistence Service Application \(Section 27.4 on page 940\)](#)
- [Configuring Remote Administration \(Section 27.5 on page 942\)](#)
- [Configuring Persistent Storage \(Section 27.6 on page 943\)](#)
- [Configuring Participants \(Section 27.7 on page 946\)](#)
- [Creating Persistence Groups \(Section 27.8 on page 947\)](#)
- [Enabling Distributed Logger in RTI Services \(Section Chapter 39 on page 1049\)](#)
- [Enabling RTI Monitoring Library in Persistence Service \(Section 27.12 on page 958\)](#)

Example Configuration File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A Configuration file may be used by several
      persistence services specifying multiple
      <persistence_service> entries
-->
<dds>
  <!-- QoS LIBRARY SECTION -->
  <qos_library name="QosLib1">
    <qos_profile name="QosProfile1">
      <datawriter_qos name="WriterQos1">
        <history>
          <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
        </history>
      </datawriter_qos>
      <datareader_qos name="ReaderQos1">
        <reliability>
          <kind>DDS_RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <history>
          <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
        </history>
      </datareader_qos>
    </qos_profile>
  </qos_library>
  <!-- PERSISTENCE SERVICE SECTION -->
  <persistence_service name="Srv1">
    <!-- REMOTE ADMINISTRATION SECTION -->
    <administration>
      <domain_id>72</domain_id>
      <distributed_logger>
        <enabled>true</enabled>
      </distributed_logger>
    </administration>
  </persistence_service>
</dds>
```

```

    </administration>
<!-- PERSISTENT STORAGE SECTION -->
<persistent_storage>
  <filesystem>
    <directory>/tmp</directory>
    <file_prefix>PS</file_prefix>
  </filesystem>
</persistent_storage>
<!-- DOMAINPARTICIPANT SECTION -->
<participant name="Part1">
  <domain_id>71</domain_id>
  <!-- PERSISTENCE GROUP SECTION -->
  <persistence_group name="PerGroup1" filter="*">
    <single_publisher>true</single_publisher>
    <single_subscriber>true</single_subscriber>
    <datawriter_qos base_name="QosLib1::QosProfile1"/>
    <datareader_qos base_name="QosLib1::QosProfile1"/>
  </persistence_group>
</participant>
</persistence_service>
</dds>

```

27.2.1 Configuration File Syntax

The configuration file must follow these syntax rules:

- The syntax is XML and the character encoding is UTF-8.
- Opening tags are enclosed in `<>`; closing tags are enclosed in `</>`.
- A value is a UTF-8 encoded string. Legal values are alphanumeric characters. All leading and trailing spaces are removed from the string before it is processed.

For example, "`<tag> value </tag>`" is the same as "`<tag>value</tag>`".

- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: `<!-- comment -->`.
- The root tag of the configuration file must be `<dds>` and end with `</dds>`.
- The primitive types for tag values are specified in [Table 27.1 Supported Tag Values](#).

Table 27.1 Supported Tag Values

Type	Format	Notes
DDS_Boolean	yes, 1, true, BOOLEAN_TRUE or DDS_BOOLEAN_TRUE: these all mean TRUE	Not case-sensitive
	no, 0, false, BOOLEAN_FALSE or DDS_BOOLEAN_FALSE: these all mean FALSE	

Table 27.1 Supported Tag Values

Type	Format	Notes
DDS_Enum	A string. Legal values are those listed in the C or Java API Reference HTML documentation.	Must be specified as a string. (Do not use numeric values.)
DDS_Long	-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff or LENGTH_UNLIMITED or DDS_LENGTH_UNLIMITED	A 32-bit signed integer
DDS_UnsignedLong	0 to 4294967296 or 0 to 0xffffffff	A 32-bit unsigned integer
String	UTF-8 character string	All leading and trailing spaces are ignored between two tags

27.2.2 XML Validation

27.2.2.1 Validation at Run Time

Persistence Service validates the input XML files using a builtin Document Type Definition (DTD). You can find a copy of the builtin DTD in `<NDDSHOME>a/resource/schema/rti_persistence_service.dtd`. (This is only a copy of what the Persistence Service core uses. Changing this file has no effect unless you specify its path with the DOCTYPE tag, described below.)

You can overwrite the builtin DTD by using the XML tag, `<!DOCTYPE>`. For example, the following indicates that Persistence Service must use a different DTD file to perform validation:

```
<!DOCTYPE dds SYSTEM
  "/local/usr/rti/dds/modified_rtipersistenceservice.dtd">
```

If you do not specify the DOCTYPE tag in the XML file, the builtin DTD is used.

The DTD path can be absolute, or relative to the application's current working directory.

27.2.2.2 Validation During Editing

Persistence Service provides DTD and XSD files that describe the format of the XML content. We recommend including a reference to one of these documents in the XML file that contains the persistence service's configuration—this provides helpful features in code editors such as Visual Studio and Eclipse,

^aSee [Paths Mentioned in Documentation \(Section on page xxxviii\)](#).

including validation and auto-completion while you are editing the XML file. Including a reference to the XSD file in the XML documents provides stricter validation and better auto-completion than the corresponding DTD file.

The DTD and XSD definitions of the XML elements are in `<NDDSHOME>/resource/schema (rti_persistence_service.dtd` and `rti_persistence_service.xsd`, respectively).

To include a reference to the XSD document in your XML file, use the attribute `xsi:noNamespaceSchemaLocation` in the `<dds>` tag. For example (in the following, replace `<NDDSHOME>` with the Connex DDS installation directory, see [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)):

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
"<NDDSHOME>/resource/schema/rti_persistence_service.xsd">
  ...
</dds>
```

To include a reference to the DTD document in your XML file, use the `<!DOCTYPE>` tag. For example (in the following, replace `<NDDSHOME>` with the Connex DDS installation directory):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dds SYSTEM
"<NDDSHOME>/resource/schema/rti_persistence_service.dtd">
<dds>
  ...
</dds>
```

27.3 QoS Configuration

Each persistence group and participant has a set of DDS QoSs. There are six tags:

- `<participant_qos>`
- `<publisher_qos>`
- `<subscriber_qos>`
- `<topic_qos>`
- `<datawriter_qos>`
- `<datareader_qos>`

Each QoS is identified by a name. The QoS can inherit its values from other QoSs described in the XML file. For example:

```
<datawriter_qos name="DerivedWriterQos" base_name="Lib::BaseWriterQos">
  <history>
    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
```



```

</history>
</datawriter_qos>

```

In the above example, the writer QoS named 'DerivedWriterQos' inherits the values from the writer QoS 'BaseWriterQos' contained in the library 'Lib'. The HistoryQosPolicy **kind** is set to DDS_KEEP_ALL_HISTORY_QOS.

Each XML tag with an associated name can be uniquely identified by its fully qualified name in C++ style. For more information on tags, see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#)

The persistence groups and participants can use QoS libraries and profiles to configure their QoS values. For example:

```

<dds>
  <!-- QoS LIBRARY SECTION -->
  <qos_library name="QosLib1">
    <qos_profile name="QosProfile1">
      <datawriter_qos name="WriterQos1">
        <history>
          <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
        </history>
      </datawriter_qos>
    </qos_profile>
  </qos_library>
  <!-- PERSISTENCE SERVICE SECTION -->
  <persistence_service name="Srv1">
    ...
  <!-- PERSISTENCE GROUP SECTION -->
  <persistence_group name="PerGroup1" filter="*">
    <single_publisher>true</single_publisher>
    <single_subscriber>true</single_subscriber>
    <datawriter_qos base_name="QosLib1::QosProfile1"/>
  </persistence_group>
</persistence_service>
</dds >

```

For more information about QoS libraries and profiles see [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#).

27.4 Configuring the Persistence Service Application

Each execution of the Persistence Service application is configured using the content of a tag: <persistence_service>. When you start Persistence Service (described in [Starting Persistence Service \(Section 28.1 on page 962\)](#)), you must specify which <persistence_service> tag to use to configure the service.

For example:

```

<dds>
  <persistence_service name="Srv1">
    ...
  </persistence_service>
</dds>

```

If you do not specify a service name when you start Persistence Service, the service will print the list of available configurations and then exit.

Because a configuration file may contain multiple `<persistence_service>` tags, one file can be used to configure multiple Persistence Service executions.

[Table 27.2 Persistence Service Application Tags](#) lists the tags you can specify for a persistence service. Notice that `<participant>` is required. For default values, please see the API Reference HTML documentation.

Table 27.2 Persistence Service Application Tags

Tags within <code><persistence_service></code>	Description	Number of Tags Allowed
<code><administration></code>	Enables and configures remote administration. See Configuring Remote Administration (Section 27.5 on the facing page) .	0 or 1
<code><annotation></code>	Provides a description for the persistence service configuration. Example: <pre><annotation> <documentation> Persists in the file system all topics published with PERSISTENT durability </documentation> </annotation></pre>	0 or 1
<code><purge_samples_after_acknowledgment></code>	A <code>DDS_Boolean</code> that indicates whether or not a <code>PRSTDataWriter</code> will purge a DDS sample from its cache once it is acknowledged by all the matching/active <i>DataReaders</i> and all the Durable Subscriptions. Default: 0 See Configuring Durable Subscriptions in Persistence Service (Section 27.9 on page 955) .	0 or 1
<code><participant></code>	For each <code><participant></code> tag, Persistence Service creates two <i>DomainParticipants</i> on the same domain ID: one to subscribe to changes and one to publish changes. There may be more Participant pairs created when there are multiple versions of a type (see Support for Extensible Types (Section 27.13 on page 959)). The QoS values used to configure both <i>DomainParticipants</i> are the same, except for: <ul style="list-style-type: none"> The participant_id in the WIRE_PROTOCOL QoSPolicy (DDS Extension) (Section 8.5.9 on page 610). If participant_id is not -1 (the default value, which means automatic selection), Persistence Service uses participant_id for the first <i>DomainParticipant</i> and participant_id+1 for the second <i>DomainParticipant</i>. <p>The TCP server ports are configured with the properties <code>dds.transport.tcp.server_bind_port</code> and <code>dds.transport.tcp.public_address</code>. See TCP/TLS Transport Properties (Section 35.1.6 on page 1002).</p>	1 or more (required)

Table 27.2 Persistence Service Application Tags

Tags within <persistence_ service>	Description	Number of Tags Allowed
<persistent_ storage>	When this tag is present, the topic data will be persisted to disk. You can select between file storage and relational database storage. See Configuring Persistent Storage (Section 27.6 on the next page) .	0 or 1
<synchronization>	Enables synchronization in redundant persistence service instances. See Synchronizing of Persistence Service Instances (Section 27.10 on page 956) . Default: Synchronization is not enabled	0 or 1

27.5 Configuring Remote Administration

You can create a Connex DDS application that can remotely control Persistence Service. The **<administration>** tag is used to enable remote administration and configure its behavior.

By default, remote administration is turned off in Persistence Service.

When remote administration is enabled, Persistence Service will create a *DomainParticipant*, *Publisher*, *Subscriber*, *DataWriter*, and *DataReader*. These *Entities* are used to receive commands and send responses. You can configure these entities with QoS tags within the **<administration>** tag.

[Table 27.3 Remote Administration Tags](#) lists the tags allowed within **<administration>** tag. Notice that the **<domain_id>** tag is required.

For more details, please see [Administering Persistence Service from a Remote Location \(Section Chapter 29 on page 966\)](#).

Note: The command-line options used to configure remote administration take precedence over the XML configuration (see [Table 28.1 Persistence Service Command-Line Options](#)).

Table 27.3 Remote Administration Tags

Tags within <administration>	Description	Number of Tags Allowed
<datareader_qos>	Configures the <i>DataReader</i> QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults with the following changes: reliability.kind = DDS_RELIABLE_RELIABILITY_QOS (this value cannot be changed) history.kind = DDS_KEEP_ALL_HISTORY_QOS resource_limits.max_samples = 32	0 or 1

Table 27.3 Remote Administration Tags

Tags within <administration>	Description	Number of Tags Allowed
<datawriter_qos>	Configures the <i>DataWriter</i> QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults with the following changes: history.kind = DDS_KEEP_ALL_HISTORY_QOS resource_limits.max_samples = 32	0 or 1
<distributed_logger>	Configures <i>RTI Distributed Logger</i> . See	0 or 1
<domain_id>	Specifies which domain ID Persistence Service will use to enable remote administration.	1 (required)
<participant_qos>	Configures the DomainParticipant QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults.	0 or 1
<publisher_qos>	Configures the Publisher QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults.	0 or 1
<subscriber_qos>	Configures the Subscriber QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults.	0 or 1

27.6 Configuring Persistent Storage

The <persistent_storage> tag is used to persist DDS samples into permanent storage. If the <persistent_storage> tag is not specified, the service will operate in TRANSIENT mode and all the data will be kept in memory. Otherwise, the persistence service will operate in PERSISTENT mode and all the topic data will be stored into the filesystem or into a relational database that provides an ODBC driver.

Table 27.4 [Persistent Storage tags](#) lists the tags that you can specify in <persistent_storage>.

Relational Database Limitations: The ODBC storage does not support BLOBs. The maximum size for a serialized DDS sample is 65535 bytes in MySQL.

Table 27.4 Persistent Storage tags

Tags within <persistent_ storage>	Description	Number of Tags Allowed
<external_ database>	When this tag is present, the topic data will be persisted in a relational database. This tag is required if <filesystem> is not specified. See Table 27.5 External Database Tags .	0 or 1
<filesystem>	When this tag is present, the topic data will be persisted into files. This tag is required if <external_database> is not specified. See Table 27.6 Filesystem tags .	0 or 1
<restore>	This DDS_Boolean (see Table 27.1 Supported Tag Values) indicates if the topic data associated with a previous execution of the persistence service must be restored or not. If the topic data is not restored, it will be deleted from the persistent storage. Default: 1	0 or 1
<type_object_ max_ serialized_ length>	Defines the length in bytes of the database column used to store the TypeObjects associated with PRSTDataWriters and PRSTDataReader. For additional information on TypeObjects, see the RTI Connex DDS Core Libraries Getting Started Guide Addendum for Extensible Types . Default: 10488576	0 or 1

Table 27.5 External Database Tags

Tags within <external_ database>	Description	Number of Tags Allowed
<dsn>	DSN used to connect to the database using ODBC. You should create this DSN through the ODBC settings on Windows systems, or in your .odbc.ini file on UNIX/Linux systems. This tag is required.	1 (required)
<odbc_library>	Specifies the ODBC driver to load. By default, Connex DDS will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).	0 or 1
<password>	Password to connect to the database. Default: no username is used	0 or 1
<username>	Username to connect to the database. Default: no username is used	0 or 1

Table 27.6 Filesystem tags

Tags within <filesystem>	Description	Number of Tags Allowed
<directory>	<p>Specifies the directory of the files in which topic data will be persisted. There will be one file per PRSTDataWriter/PRSTDataReader pair.</p> <p>The directory must exist; otherwise the service will report an error upon start up.</p> <p>Default: current working directory</p>	0 or 1
<file_prefix>	<p>A name prefix associated with all the files created by Persistence Service.</p> <p>Default: PS</p>	0 or 1
<journal_mode>	<p>Sets the journal mode of the persistent storage. This tag can take these values:</p> <ul style="list-style-type: none"> • DELETE: Deletes the rollback journal at the conclusion of each transaction. • TRUNCATE: Commits transactions by truncating the rollback journal to zero-length instead of deleting it. • PERSIST: Prevents the rollback journal from being deleted at the end of each transaction. Instead, the header of the journal is overwritten with zeros. • MEMORY: Stores the rollback journal in volatile RAM. This saves disk I/O. • WAL: Uses a write-ahead log instead of a rollback journal to implement transactions. • OFF: Completely disables the rollback journal. If the application crashes in the middle of a transaction when the OFF journaling mode is set, the files containing the DDS samples will very likely be corrupted. <p>Default: DELETE</p>	0 or 1
<synchronization>	<p>Determines the level of synchronization with the physical disk.</p> <p>This tag can take three values:</p> <ul style="list-style-type: none"> • FULL: Every DDS sample is written into physical disk as Persistence Service receives it. • NORMAL: DDS samples are written into disk at critical moments. • OFF: No synchronization is enforced. Data will be written to physical disk when the OS flushes its buffers. <p>Default: OFF</p>	0 or 1

Table 27.6 Filesystem tags

Tags within <filesystem>	Description	Number of Tags Allowed
<trace_file>	Specifies the name of the trace file for debugging purposes. The trace file contains information about all SQL statements executed by the persistence service. Default: No trace file is generated	0 or 1
<vacuum>	Sets the auto-vacuum status of the storage. This tag can take these values: <ul style="list-style-type: none"> NONE: When data is deleted from the storage files, the files remain the same size. FULL: The storage files are compacted every transaction. Default: FULL	0 or 1

27.7 Configuring Participants

An XML <persistence_service> tag will contain a set of <participants>. The persistence service will persist topics published in the domainIDs associated with these participants. For example:

```
<persistence_service name="Srv1">
  <participant name="Part1">
    <domain_id>71</domain_id>
    ...
  </participant>
  <participant name="Part2">
    <domain_id>72</domain_id>
    ...
  </participant>
</persistence_service>
```

Using the above example, the persistence service will create two pairs of *DomainParticipants* on DDS domains 71 and 72, respectively. In each pair, one *DomainParticipant* is used to receive data and the other to publish.

After the *DomainParticipants* are created, the persistence service will monitor the discovery traffic, looking for topics to persist.

Notice that in some cases there may be more than one pair of *DomainParticipants* per domain when there are multiple versions of a type for a given topic. (See [Support for Extensible Types \(Section 27.13 on page 959\)](#).)

The <domain_id> tag can be specified alternatively as an attribute of <participant>. For example:

```
<persistence_service name="Srv1">
  <participant name="Part1" domain_id="71">
```

```

...
</participant>
</persistence_service>

```

Table 27.7 Participant Tags describes the participant tags. Notice that `<persistence_group>` is required.

Table 27.7 Participant Tags

Tags within <code><participant></code>	Description	Number of Tags Allowed
<code><domain_id></code>	Domain ID associated with the Participant. The domain ID can be specified as an attribute of the participant tag. Default: 0	0 or 1
<code>< durable_ subscriptions></code>	Configures a set of Durable Subscriptions for a given topic. This is a sequence of <code><element></code> tags, each of which has a <code><role_name></code> , a <code><topic_name></code> , and a <code><quorum></code> . For example: <pre> <durable_subscriptions> <element> <role_name>DurSub1</role_name> <topic_name>Example MyType</topic_name> <quorum>2</quorum> </element> <element> <role_name>DurSub2</role_name> <topic_name>Example MyType</topic_name> </element> </durable_subscriptions> </pre> Default: Empty list See Configuring Durable Subscriptions in Persistence Service (Section 27.9 on page 955) for additional information	0 or 1
<code><participant_qos></code>	Participant QoS. Default: DDS defaults	0 or 1
<code><persistence_group></code>	A persistence group describes a set of topics whose data that must be persisted by the persistence service.	1 or more (required)

27.8 Creating Persistence Groups

The topics that must be persisted in a specific domain ID are specified using `<persistence_group>` tags. A `<persistence_group>` tag defines a set of topics identified by a POSIX expression.

For example:

```

<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="H*">

```



```

...
  </persistence_group>
</participant>

```

In the above example, the persistence group 'PerGroup1' is associated with all the topics published in DDS domain 71 whose name starts with 'H'.

When a participant discovers a topic that matches a persistence group, it will create a PRSTDataReader and a PRSTDataWriter. The PRSTDataReader and PRSTDataWriter will be configured using the QoS policies associated with the persistence group. The DDS samples received by the PRSTDataReader will be persisted in the queue of the corresponding PRSTDataWriter.

A **<participant>** tag can contain multiple persistence groups; the set of topics that each one represents can intersect.

[Table 27.8 Persistence Group Tags](#) further describes the persistence group tags. For default values, please see the API Reference HTML documentation.

Table 27.8 Persistence Group Tags

Tags within <persistence_group>	Description	Number of Tags Allowed
<allow_durable_subscriptions>	A DDS_Boolean (see Table 27.1 Supported Tag Values) that enables support for durable subscriptions in the PRSTDataWriters created in a persistence group. When Durable Subscriptions are not required, setting this property to 0 will increase performance. Default: 1	0 or 1
<content_filter>	Content filter topic expression. A persistence group can subscribe to a specific set of data based on the value of this expression. A filter expression is similar to the WHERE clause in SQL. For more information on the syntax, please see the API Reference Documentation (from the Modules page, select RTI Connexx DDS DDS API Reference, Queries and Filters Syntax). Default: no expression	0 or 1
<datareader_qos>	PRSTDataReader QoS ¹ . See QoSs (Section 27.8.1 on page 952) . Default: DDS defaults	0 or 1
<datawriter_qos>	PRSTDataWriter QoS ² . See QoSs (Section 27.8.1 on page 952) . Default: DDS defaults	0 or 1

¹These fields cannot be set and are assigned automatically: protocol.virtual_guid, protocol.rtps_object_id, durability.kind.

²These fields cannot be set and are assigned automatically: protocol.virtual_guid, protocol.rtps_object_id, durability.kind.

Table 27.8 Persistence Group Tags

Tags within <persistence_ group>	Description	Number of Tags Allowed
<deny_filter>	<p>Specifies a list of POSIX expressions separated by commas that describe the set of topics to be denied in the persistence group.</p> <p>This "black" list is applied to the topics that pass the filter specified with the <filter> tag</p> <p>Default: *</p>	0 or 1
<filter>	<p>Specifies a list of POSIX expressions separated by commas that describe the set of topics associated with the persistence group.</p> <p>The filter can be specified as an attribute of <persistence_group> as well.</p> <p>Default: *</p>	0 or 1
<memory_ management>	<p>This flag configures the memory allocation policy for DDS samples in PRSTDataReaders and PRSTDataWriters.</p> <p>See Memory Management (Section 27.8.5 on page 954).</p>	0 or 1
<propagate_ dispose>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that controls whether or not the persistence service propagates dispose messages from DataWriters to DataReaders.</p> <p>Default: 1</p>	0 or 1
<propagate_ source_ timestamp>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values). When this tag is 1, the DDS data samples sent by the PRSTDataWriters preserve the source timestamp that was associated with them when they were published by the original DataWriter.</p> <p>Default: 0</p>	0 or 1
<propagate_ unregister>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that controls whether or not the persistence service propagates unregister messages from DataWriters to DataReaders.</p> <p>Default: 0</p>	0 or 1
<publisher_qos>	<p>Publisher QoS. See QoSs (Section 27.8.1 on page 952).</p> <p>Default: DDS defaults</p>	0 or 1
<reader_ checkpoint_ frequency>	<p>This property controls how often (expressed as a number of DDS samples) the PRSTDataReader state is stored in the database. The PRSTDataReaders are the DataReaders created by the persistence service.</p> <p>A high frequency will provide better performance. However, if the persistence service is restarted, it may receive some duplicate DDS samples. The persistence service will send these duplicates DDS samples on the wire but they will be filtered by the DataReaders and they will not be propagated to the application.</p> <p>This property is only applicable when the persistence service operates in persistent mode (the <persistent_storage> tag is present).</p> <p>Default: 1</p>	0 or 1

Table 27.8 Persistence Group Tags

Tags within <persistence_ group>	Description	Number of Tags Allowed
<share_ database_ connection>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that indicates if the persistence service will create an independent database connection per PRSTDataWriter in the group (0) or per Publisher (1) in the group.</p> <p>When <single_publisher> is 0 and <share_database_connection> is 1, there is a single database connection per group. All the PRSTDataWriters will share the same connection.</p> <p>When <single_publisher> is 1 or <share_database_connection> is 0, there is a database connection per PRSTDataWriter.</p> <p>This parameter is only applicable to configurations persisting the data into a relational database using the tag <external_database> in <persistent_storage>.</p> <p>See Sharing a Database Connection (Section 27.8.4 on page 954)</p> <p>Default: 0</p>	0 or 1
<single_ publisher>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that indicates if the persistence service should create one Publisher per persistence group or one Publisher per PRSTDataWriter inside the persistence group. See Sharing a Publisher/Subscriber (Section 27.8.3 on page 953).</p> <p>Default: 1</p>	0 or 1
<single_ subscriber>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that indicates if the persistence service should create one Subscriber per persistence group or one Subscriber per PRSTDataReader in the persistence group.</p> <p>See Sharing a Publisher/Subscriber (Section 27.8.3 on page 953).</p> <p>Default: 1</p>	0 or 1
<subscriber_ qos>	<p>Subscriber QoS. See QoSs (Section 27.8.1 on page 952).</p> <p>Default: DDS defaults</p>	0 or 1
<topic_qos>	<p>Topic QoS. See QoSs (Section 27.8.1 on page 952).</p> <p>Default: DDS defaults</p>	0 or 1
<use_durability_ service>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that indicates if the HISTORY and RESOURCE_LIMITS QoS policy of the PRSTDataWriters and PRSTDataReaders should be configured based on the DURABILITY_SERVICE value of the discovered DataWriters.</p> <p>See DurabilityService QoS Policy (Section 27.8.2 on page 953)</p> <p>Default: 0</p>	0 or 1
<writer_ack_ period>	<p>Controls how often (expressed in milliseconds) DDS samples are marked as ACK'd in the database by the PRSTDataWriter.</p> <p>Default: 0</p>	0 or 1

Table 27.8 Persistence Group Tags

Tags within <persistence_ group>	Description	Number of Tags Allowed
<writer_ checkpoint_ period>	<p>Controls how often (expressed in milliseconds) transactions are committed for a PRSTDataWriter.</p> <p>A value of 0 indicates that transactions will be committed immediately. This is the recommended setting to avoid losing data in the case of an unexpected error in Persistence Service and/or the underlying hardware/software infrastructure.</p> <p>For applications that can tolerate some data losses, setting this tag to a value greater than 0 will increase performance.</p> <p>Default: 0</p>	0 or 1
<writer_ checkpoint_ volume>	<p>Controls how often (expressed as a number of DDS samples) transactions are committed for a PRSTDataWriter.</p> <p>A value of 1 indicates that DDS samples will be persisted by the PRSTDataWriters immediately. This is the recommended setting to avoid losing data in the case of an unexpected error in persistence service and/or the underlying hardware/software infrastructure.</p> <p>For application that can tolerate some data losses, setting this tag to a value greater than 1 will increase performance.</p> <p>Default: 1</p>	0 or 1
<late_joiner_ read_batch>	<p>Defines how many DDS samples will be pre-fetched by a PRSTDataWriter to satisfy requests from late-joiners.</p> <p>When a DataReader requests DDS samples from a PRSTDataWriter by sending a NACK message, the PRSTDataWriter may retrieve additional DDS samples from the database to minimize disk access.</p> <p>This parameter determines that amount of DDS samples that will be retrieved preemptively from the database by the PRSTDataWriter.</p> <p>Default: 20000</p>	0 or 1
<sample_ logging>	<p>This tag can be used to enable and configure a DDS sample log for the PRSTDataWriters in a persistence group. A DDS sample log is a buffer of DDS samples on disk that, when used in combination with delegate reliability, allow decoupling the original DataWriters from slow DataReaders.</p> <p>For additional information on the DDS sample log, see Scenario: Slow Consumer (Section 30.3 on page 975).</p> <p>Default: DDS sample log is disabled</p>	0 or 1

Table 27.8 Persistence Group Tags

Tags within <persistence_ group>	Description	Number of Tags Allowed
<writer_in_memory_state>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that determines how much state will be kept in memory by the PRSTDataWriters in order to avoid accessing the persistent storage.</p> <p>The property is only applicable when the persistence service operates in persistent mode (the <persistent_storage> tag is present).</p> <p>If this property is 1, the PRSTDataWriters will keep a copy of all the instances in memory. They will also keep a fixed state overhead of 24 bytes per DDS sample. This mode provides the best performance. However, the restore operation will be slower and the maximum number of DDS samples that a PRSTDataWriter can manage will be limited by the available physical memory.</p> <p>If this property is 0, all the state will be kept in the underlying persistent storage. In this mode, the maximum number of DDS samples that a PRSTDataWriter can manage will not be limited by the available physical memory.</p> <p>Default: If the HistoryQosPolicy's kind is KEEP_LAST or the ResourceLimitsQosPolicy's max_samples != DDS_UNLIMITED_LENGTH, the default is 1. Otherwise, the default is 0.</p>	0 or 1
<use_wait_set>	<p>A DDS_Boolean (see Table 27.1 Supported Tag Values) that indicates if Persistence Service will use Waitsets or Listeners to read data from the PRSTDataReaders of the group.</p> <p>By default, the usage of Waitsets is disabled. With this configuration, Persistence Service uses the on_data_available() listener callback to take the data from the PRSTDataReaders within the persistence group. The write operation in a PRSTDataWriter is called within the listener callback.</p> <p>When Waitsets are enabled, Persistence Service will use them to read the data:</p> <p>If <single_subscriber> is set to 1, there will be a single Waitset and a read thread shared across all the PRSTDataReaders in the group.</p> <p>If <single_subscriber> is set to 0, there will be a Waitset and a read thread per PRSTDataReader in the group.</p> <p>The write operation in a PRSTDataWriter is called by the read thread associated with the PRSTDataReader.</p> <p>Default: 0</p>	0 or 1

27.8.1 QoSs

When a persistence service discovers a topic 'A' that matches a specific persistence group, it creates a reader (known as 'PRSTDataReader') and writer ('PRSTDataWriter') to persist that topic. The QoSs associated with these readers and writers, as well as the corresponding publishers and subscribers, can be configured inside the persistence group using QoS tags.

For example:

```
<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <publisher_qos base_name="QosLib1::PubQos1"/>
```

```

<subscriber_qos base_name="QosLib1::SubQos1"/>
<datawriter_qos base_name="QosLib1::WriterQos1"/>
<datareader_qos base_name="QosLib1::ReaderQos1"/>
...
</persistence_group>
</participant>

```

For instance, the number of DDS samples saved by Persistence Service is configurable through the [HISTORY QoS Policy \(Section 6.5.10 on page 376\)](#) of the PRSTDataWriters.

If a QoS tag is not specified the persistence service will use the corresponding DDS default values ([DurabilityService QoS Policy \(Section 27.8.2 below\)](#) describes an exception to this rule).

27.8.2 DurabilityService QoS Policy

The [DURABILITY SERVICE QoS Policy \(Section 6.5.8 on page 372\)](#) associated with a *DataWriter* is used to configure the HISTORY and the RESOURCE_LIMITS associated with the PRSTDataReaders and PRSTDataWriters.

By default, the HISTORY and RESOURCE_LIMITS of a PRSTDataReader and *PRSTDataWriter* with topic 'A' will be configured using the values specified in the XML file used to configure Persistence Service. To overwrite those values and use the values in the [DURABILITY SERVICE QoS Policy \(Section 6.5.8 on page 372\)](#) of the first discovered *DataWriter* publishing 'A', you can use the tag `<use_durability_service>` in the persistence group definition:

```

<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <use_durability_service/>1</ use_durability_service>
    ...
  </persistence_group>
</participant>

```

27.8.3 Sharing a Publisher/Subscriber

By default, the PRSTDataWriters and PRSTDataReaders associated with a persistence group will share the same Publisher and Subscriber.

To associate a different Publisher and Subscriber with each PRSTDataWriter and PRSTDataReader, use the tags `<single_publisher>` and `<single_subscriber>`, as follows:

```

<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <single_publisher/>0</single_publisher>
    <single_subscriber/>0</single_subscriber>
    ...
  </persistence_group>
</participant>

```

```

    </persistence_group>
</participant>

```

27.8.4 Sharing a Database Connection

By default, the persistence service will share a single ODBC database connection to persist the topic data received by each PRSTDataReader.

To associate an independent database connection to the PRSTDataReaders created by the persistence service, use the tag `<share_database_connection>`, as follows:

```

<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <share_database_connection>0</share_database_connection>
    ...
  </persistence_group>
</participant>

```

Sharing a database connection optimizes the resource usage. However, the concurrency of the system decreases because the access to the database connection must be protected.

27.8.5 Memory Management

The DDS samples received and stored by the PRSTDataReaders and PRSTDataWriters are in serialized form.

The serialized size of a DDS sample is the number of bytes required to send the DDS sample on the wire. The maximum serialized size of a DDS sample is the number of bytes that the largest DDS sample for a given type requires on the wire.

By default, the PRSTDataReaders and PRSTDataWriters created by the persistence service try to allocate multiple DDS samples to their maximum serialized size. This may cause memory allocation issues when the maximum serialized size is significantly large.

For PRSTDataReaders, the number of DDS samples in the *DataReader's* queues can be controlled using the QoS values **resource_qos.resource_limits.max_samples** and **resource_qos.resource_limits.initial_samples**.

The PRSTDataWriters keep a cache of DDS samples so that they do not have to access the database every time. The minimum size of this cache is 32 DDS samples.

In addition, each PRSTDataWriter keeps an additional DDS sample called the DB sample, which is used to move information from the *DataWriter* cache to the database and vice versa

The **<memory_management>** tag in a persistence group can be used to control the memory allocation policy for the DDS samples created by PRSTDataReaders and PRSTDataWriters in the persistence group.

Table 27.9 Memory Management Tags describes the memory management tags.

Table 27.9 Memory Management Tags

Tags within <memory_management>	Description	Number of Tags Allowed
<persistent_sample_buffer_max_size>	<p>This tag is used to control the memory associated with the DB sample in a PRSTDataWriter. The persistence service will not be able to store a DDS sample into persistent storage if the serialized size is greater than this value. Therefore, this parameter must be used carefully.</p> <p>Default: LENGTH_UNLIMITED (DB sample is allocated to the maximum size).</p>	0 or 1
<pool_sample_buffer_max_size>	<p>This tag applies to both PRSTDataReaders and PRSTDataWriters. Its value determines the maximum size (in bytes) of the buffers that will be pre-allocated to store the DDS samples. If the space required for a new DDS sample is greater than this size, the persistence service will allocate the memory dynamically to the exact size required by the DDS sample.</p> <p>This parameter is used to control the memory allocated for the DDS samples in the PRSTDataReaders queues and the PRSTDataWriters caches.</p> <p>The size of the DB sample in the PRSTDataWriters is controlled by the value of the tag <persistent_sample_buffer_max_size>.</p> <p>Default: LENGTH_UNLIMITED (DDS samples are allocated to the maximum size).</p>	0 or 1

27.9 Configuring Durable Subscriptions in Persistence Service

This section assumes you are familiar with the concept of [Required Subscriptions \(Section 6.3.13 on page 294\)](#).

A Durable Subscription is a Required Subscription where DDS samples are stored and forwarded by *Persistence Service*.

There are two ways to create a Durable Subscriptions:

1. Programmatically using a *DomainParticipant* API:

A subscribing application can register a Durable Subscription by providing the topic name and the endpoint group information, consisting of the Durable Subscription **role_name** and the **quorum**. To register or delete a Durable Subscription, use the *DomainParticipant's* **register_durable_subscription()** and **delete_durable_subscription()** operations, respectively (see [Table 8.3 DomainParticipant Operations](#)). The Durable Subscription information is propagated via a built-in topic to *Persistence Service*.

2. Preconfigure *Persistence Service* with a set of Durable Subscriptions:

Persistence Service can be (pre-)configured with a list of Durable Subscriptions using the <**durable_subscriptions**> XML tag under <**participant**>.


```

<participant name="Participant">
  ...
  < durable_subscriptions>
    < element>
      < role_name>Logger</role_name>
      < topic_name>Track</topic_name>
      < quorum>2</quorum>
    </element>
    < element>
      < role_name>Processor</role_name>
      < topic_name>Track</topic_name>
      < quorum>1</quorum>
    </element>
  </durable_subscriptions>
</participant>

```

After registering or configuring the persistence service with specific Durable Subscriptions, the persistence service will keep DDS samples until they are acknowledged by all the required Durable Subscriptions. In the above example, the DDS samples must be acknowledged by two *DataReaders* that belong to the “Logger” Durable Subscription and one *DataReader* belonging to the “Processor” Durable Subscription.

27.9.1 DDS Sample Memory Management With Durable Subscriptions

The maximum number of DDS samples that will be kept in a PRSTDataWriter queue is determined by the value of `<resource_limits><max_samples>` in the `<writer_qos>` used to configure the PRSTDataWriter.

By default, a PRSTDataWriter configured with `KEEP_ALL <history><kind>` will keep the DDS samples in its cache until they are acknowledged by all the Durable Subscriptions associated with the PRSTDataWriter. After the DDS samples are acknowledged by the Durable Subscriptions, they will be marked as reclaimable but they will not be purged from the PRSTDataWriter’s queue until the *DataWriter* needs these resources for new DDS samples. This may lead to inefficient resource utilization, especially when `<max_samples>` is high or UNLIMITED.

The PRSTDataWriter behavior can be changed to purge DDS samples after they have been acknowledged by all the active/matching *DataReaders* and all the Durable Subscriptions configured for the `<persistence_service>`. To do so, set the tag `<purge_samples_after_acknowledgment>` under `<persistence_service>` to TRUE. Notice that this setting is global to the service and applies to all the PRSTDataWriters created by each `<persistence_group>`.

27.10 Synchronizing of Persistence Service Instances

By default, different *Persistence Service* instances do *not* synchronize with each other. For example, in a scenario with two *Persistence Service* instances, the first persistence service could receive a DDS sample ‘S1’ from the original *DataWriter* that is not received by the second persistence service. If the disk where the first persistence service stores its DDS samples fails, ‘S1’ will be lost.

To enable synchronization between *Persistence Service* instances, use the tag **<synchronization>** under **<persistence_service>**. When it comes to synchronization, there are two different kinds of information that can be synchronized independently:

- Information about Durable Subscriptions and their states (see [Configuring Durable Subscriptions in Persistence Service \(Section 27.9 on page 955\)](#))
- DDS data samples

Table 27.10 Synchronization Tags

Tags within <synchronization>	Description	Number of Tags Allowed
<synchronize_data>	<p>Enables synchronization of DDS data samples in redundant <i>Persistence Service</i> instances.</p> <p>When set to 1, DDS samples lost on the way to one service instance can be repaired by another without impacting the original publisher of that message.</p> <p>To synchronize the instances, the tag <synchronize_data> must be set to 1 in every instance involved in the synchronization.</p> <p>Note: This DDS sample synchronization mechanism is not equivalent to database replication. The extent to which database instances have identical contents depends on the destination ordering and other QoS settings for the <i>Persistence Service</i> instances.</p> <p>Default: 0</p>	0 or 1
<synchronize_durable_subscription>	<p>Enables synchronization of Durable Subscriptions in redundant <i>Persistence Service</i> instances.</p> <p>When set to 1, the different <i>Persistence Service</i> instances will synchronize their Durable Subscription information. This information includes the set of Durable Subscriptions as well as information about the Durable Subscription's state, such as the DDS samples that have already been received by the Durable Subscriptions.</p> <p>Default: 0</p>	0 or 1
<durable_subscription_synchronization_period>	<p>The period (in milliseconds) at which the information about Durable Subscriptions is synchronized.</p> <p>Default: 5000 milliseconds</p>	0 or 1

27.11 Enabling RTI Distributed Logger in Persistence Service

Persistence Service provides integrated support for *RTI Distributed Logger* (see [Part 10: RTI Distributed Logger \(Section on page 1039\)](#)).

Distributed Logger is included in *Connex DDS* but it is not supported on all platforms; see the [RTI Connex DDS Core Libraries Platform Notes](#) to see which platforms support *Distributed Logger*.

When you enable *Distributed Logger*, *Persistence Service* will publish its log messages to *Connex DDS*. Then you can use *RTI Monitor*¹ to visualize the log message data. Since the data is provided in a *Connex DDS* topic, you can also use *rtiddspy* or even write your own visualization tool.

To enable *Distributed Logger*, modify the *Persistence Service* XML configuration file. In the `<administration>` section, add the `<distributed_logger>` tag as shown in the example below.

```
<persistence_service name="default">
  ...
  <administration>
    ...
    <distributed_logger>
      <enabled>true</enabled>
    </distributed_logger>
    ...
  </administration>
  ...
</persistence_service>
```

There are more configuration tags that you can use to control *Distributed Logger's* behavior. For example, you can specify a filter so that only certain types of log messages are published. For details, see [Enabling Distributed Logger in RTI Services \(Section Chapter 39 on page 1049\)](#)

27.12 Enabling RTI Monitoring Library in Persistence Service

Persistence Service provides integrated support for *RTI Monitoring Library* (see [Part 9: RTI Monitoring Library \(Section on page 1022\)](#)).

To enable monitoring in *Persistence Service*, you must specify the property `rti.monitor.library` for the participants that you want to monitor. For example:

```
<persistence_service name="monitoring_test">
  <participant name="monitoring_enabled_participant">
    <domain_id>54</domain_id>
    <participant_qos>
      <property>
        <value>
          <element>
            <role_name>rti.monitor.library</role_name>
            <value>rtimonitoring</value>
            <propagate>>false</propagate>
          </element>
        </value>
      </property>
```

¹*RTI Monitor* is a separate GUI application that can run on the same host as your application or on a different host.

```

    </participant_qos>
    <persistence_group name="persistAll">
        ...
    </persistence_group>
</participant>
</persistence_service>

```

Since Persistence Service is statically linked with *RTI Monitoring Library*, you do *not* need to have it in your library search path.

For details on how to configure the monitoring process, see [Configuring Monitoring Library \(Section Chapter 37 on page 1034\)](#).

27.13 Support for Extensible Types

Persistence Service includes partial support for the "Extensible and Dynamic Topic Types for DDS" specification from the Object Management Group (OMG)¹. This section assumes that you are familiar with Extensible Types and you have read the [RTI Connex DDS Core Libraries Getting Started Guide Addendum for Extensible Types](#).

Persistence groups can publish and subscribe to topics associated with final and extensible types.

The service will automatically create different pairs (PRSTDataReader, PRSTDataWriter) for each version of a type discovered for a topic in a persistence group. In Connex DDS 5.0, it is not possible to associate more than one type with a topic within a single *DomainParticipant*, therefore each version of a type requires its own *DomainParticipant*.

The [TYPE_CONSISTENCY_ENFORCEMENT QosPolicy \(Section 7.6.6 on page 532\)](#) **kind** for each PRSTDataReader is set to `DISALLOW_TYPE_COERCION`. This value cannot be overwritten by the user.

For example:

```

struct A {
    long x;
};
struct B {
    long x;
    long y;
};

```

Let's assume that *Persistence Service* is configured as follows and we have two *DataWriters* on Topic "T" publishing type "A" and type "B" and sending *TypeObject* information.

¹<http://www.omg.org/spec/DDS-XTypes/>

```

<persistence_service name="XTypes">
  <participant name="XTypesParticipant">
    <persistence_group name="XTypesPersistenceGroup">
      <filter>T</filter>
    </persistence_group>
  </participant>
</persistence_service>

```

When *Persistence Service* discovers the first *DataWriter* with type “A”, it will create a *DataReader* (PRSTDataReader) to read DDS samples from that *DataWriter*, and a *DataWriter* (PRSTDataWriter) to publish and store the received DDS samples so they can be available to late-joiners.

When *Persistence Service* discovers the second *DataWriter* with type “B”, it will see that type “B” is not equal to type “A”; then it will create a new pair (PRSTDataReader, PRSTDataWriter) to receive and store DDS samples from the second *DataWriter*.

Since the PRSTDataReaders are created with the TypeConsistencyEnforcementQosPolicy’s **kind** set to `DISALLOW_TYPE_COERCION`, the PRSTDataReader with type “A” will not match the *DataWriter* with type “B”. Likewise, the PRSTDataReader with type “B” will not match the *DataWriter* with type “A”.

27.13.1 Type Version Discrimination

Persistence Service uses the rules described in the [RTI Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types](#) to decide whether or not to create a new pair (PRSTDataReader, PRSTDataWriter) when it discovers a *DataWriter* for a topic “T”.

For *DataWriters* created with previous Connext DDS releases, *Persistence Service* will select the first pair (PRSTDataReader, PRSTDataWriter) with a registered type name equal to the discovered registered type name since *DataWriters* created with previous Connext DDS releases (before 5.0) do not send TypeObject information.

27.14 TCP Transport Support in Persistence Service

You can configure *Persistence Service*’s Participants to use the TCP Transport. To do so, enable the TCP Transport under the proper XML Persistence Service’s `<participant_qos>` tag.

Make sure the string prefix passed in the property `dds.transport.load_plugins` is `"dds.transport.tcp"`. For more information about how to enable the TCP Transport, please see [TCP/TLS Transport Properties \(Section 35.1.6 on page 1002\)](#).

Note that the *Persistence Service*’s `participant_qos` will be used at least by two Participants: one for sending data and another for receiving data. Consequently, at least two TCP Transport plugins will be instantiated when enabling the TCP Transport. In order to avoid port collisions, *Persistence Service* will automatically assign consecutive ports. For a base, it will use the values set for `dds.transport.tcp.server_`

bind_port (only when it is non-zero) and **dds.transport.tcp.public_address** (only if it is set). Consequently, the Participants creating a TCP Transport running as a server will open a minimum of two TCP ports.

Chapter 28 Running RTI Persistence Service

This chapter describes how to start and stop Persistence Service.

You can run Persistence Service on any node in the network. It does not have to be run on the same node as the publishing or subscribing applications for which it is saving/delivering data. If you run it on a separate node, make sure that the other applications can find it during the discovery process—that is, it must be in one of the `NDDS_DISCOVERY_PEERS` lists.

28.1 Starting Persistence Service

The script to run Persistence Service’s executable is located in `<NDDSHOME>1/bin`.

```
RTI Persistence Service
Usage: rtipersistenceservice [options]
Options:
  -cfgFile <file> Configuration file. This parameter is optional
                  since the configuration can be loaded from
                  other locations
  -cfgName <name> Configuration name. This parameter is required
                  and it is used to find a <persistence_service>
                  matching tag in the configuration files
  -appName <name> Application name. Used to identify this
                  execution
                  for remote administration and to name the
                  DomainParticipants
                  Default: -cfgName
  -identifyExecution Appends the host name and process ID to the
                    appName to help
                    ensure unique names
  -domainId <int> domain ID for the DomainParticipants created
                  by the service
                  Default: Use XML value
```

¹See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

```

-remoteAdministrationDomainId    <int>    Enables remote administration and sets
                                     the domain ID for the communication
                                     Default: Use XML value
-restore                          <0|1>    Indicates whether or not persistence service
                                     must restore its state from the persistent
                                     storage
                                     Default: Use XML value
-noAutoStart                      Use this option if you plan to start RTI
                                     Persistence Service remotely
-infoDir                          <dir>    The info directory of the running persistence
                                     service. The service writes a ps.pid file into
                                     this directory when is started. When the
                                     service finalizes the file is deleted
                                     Default: None
-maxObjectsPerThread             <int>    Sets the maximum number of objects that can
                                     be stored per thread for a
                                     DomainParticipantFactory
                                     Default: Connex DDS default
-serviceThreadStackSize          <int>    Service thread stack size
                                     Default: OS default
-verbosity                        [0-6]    RTI Persistence Service verbosity
                                     * 0 - silent
                                     * 1 - exceptions (Core Libraries and Service)
                                     * 2 - warnings (Service)
                                     * 3 - information (Service)
                                     * 4 - warnings (Core Libraries and Service)
                                     * 5 - tracing (Service)
                                     * 6 - tracing (Core Libraries and Service)
                                     Default: 1 (exceptions)
-version                          Prints RTI Persistence Service version
-help                             Displays this information

```

The command-line options are described with more detail in [Table 28.1 Persistence Service Command-Line Options](#)

Table 28.1 Persistence Service Command-Line Options

Command-line Option	Description
-appName <string>	<p>Assigns a name to the execution of Persistence Service.</p> <p>Remote commands will refer to the persistence service using this name.</p> <p>In addition, the name of the <i>DomainParticipants</i> created by Persistence Service will be based on this name as follows:</p> <p>RTI Persistence Service: <appName>: <participantName>(<pub sub>)</p> <p>Default: The name given with -cfgName if present, otherwise it is “RTI_Persistence_Service”</p>
-cfgFile <string>	<p>Specifies an XML configuration file for the Persistence Service.</p> <p>The parameter is optional since the Persistence Service configuration can be loaded from other locations. See How to Load the Persistence Service XML Configuration (Section 27.1 on page 935) for further details.</p>

Table 28.1 Persistence Service Command-Line Options

Command-line Option	Description
-cfgName <string>	<p>Required.</p> <p>Selects a Persistence Service configuration.</p> <p>The same configuration files can be used to configure multiple persistence services. Each Persistence Service instance will load its configuration from a different <persistence_service> tag based on the name specified with this option.</p> <p>If not specified, Persistence Service will print the list of available configurations and then exit.</p>
-identifyExecution	<p>Appends the host name and process ID to the service name provided with the -appName option. This helps ensure unique names for remote administration.</p>
-domainId <ID>	<p>Sets the domain ID for the <i>DomainParticipants</i> created by Persistence Service.</p> <p>If not specified, the value in the <participant> XML tag (see Table 27.7 Participant Tags) is used.</p>
-remoteAdministrationDomainId <ID>	<p>Enables remote administration and sets the domain ID for remote communication.</p> <p>When remote administration is enabled, Persistence Service will create a <i>DomainParticipant</i>, <i>Publisher</i>, <i>Subscriber</i>, <i>DataWriter</i>, and <i>DataReader</i> in the designated DDS domain.</p> <p>This option overwrites the value of the tag <domain_id> within <administration>.</p> <p>Default: Use the value <domain_id> under <administration>.</p>
-help	<p>Prints the Persistence Service version and list of command-line options.</p>
-licenseFile <file>	<p>Specifies the license file (path and filename). Only applicable to licensed versions of Persistence Service.</p> <p>If not specified, Persistence Service looks for the license as described in the RTI Connexx DDS Core Libraries Getting Started Guide.</p>
-restore <0 1>	<p>Indicates whether or not Persistence Service must restore its state from the persistent storage. 0 = do not restore; 1 = do restore.</p> <p>If this option is not specified, the corresponding XML value in the <persistent_storage> tag (see Table 27.4 Persistent Storage tags) is used.</p>
-noAutoStart	<p>Indicates that Persistence Service will not be started when the process is executed.</p> <p>Use this option if you plan to start Persistence Service remotely, as described in Administering Persistence Service from a Remote Location (Section Chapter 29 on page 966).</p>

Table 28.1 Persistence Service Command-Line Options

Command-line Option	Description
-infoDir <dir>	<p>The info directory of the running Persistence Service.</p> <p>Using this command line option, Persistence Service can be configured to create a file used to monitor the status of the last shutdown.</p> <p>At startup, the Persistence Service instance will create a file called ps.pid into the directory specified by -infoDir.</p> <p>If Persistence Service is shutdown gracefully, the file will be deleted before the process exists.</p> <p>If Persistence Service is not shutdown gracefully, the file will not be deleted.</p> <p>You can detect the shutdown state of Persistence Service by checking for the presence of the ps.pid file.</p> <p>If the file is present and Persistence Service is no longer running, the previous shutdown was not graceful.</p> <p>If Persistence Service is started and a ps.pid file exists, Persistence Service will immediately shutdown. In this case, you must remove the file before Persistence Service can be restarted again.</p> <p>Default: The file ps.pid will not be generated.</p>
-maxObjectsPerThread <int>	<p>Parameter used to configure the maximum objects per thread in the DomainParticipantFactory created by Persistence Service.</p> <p>Default: DDS default</p>
-serviceThreadStackSize <int>	<p>Service thread stack size.</p> <p>Default: DDS default</p>
-verbosity	<p>Persistence Service verbosity:</p> <ul style="list-style-type: none"> 0 - No verbosity 1 - Exceptions (Core Libraries and Persistence Service) (default) 2 - Warning (Persistence Service) 3 - Information (Persistence Service) 4 - Warning (Core Libraries and Persistence Service) 5 - Tracing (Persistence Service) 6 - Tracing (Core Libraries and Persistence Service) <p>Each verbosity level, <i>n</i>, includes all the verbosity levels smaller than <i>n</i>.</p>
-version	<p>Prints the Persistence Service version.</p>

28.2 Stopping Persistence Service

To stop Persistence Service: press **Ctrl-C**. *Persistence Service* will close all files and perform a clean shutdown. *Persistence Service* can also be stopped and shutdown remotely (see [Administering Persistence Service from a Remote Location \(Section Chapter 29 on page 966\)](#)).

Chapter 29 Administering Persistence Service from a Remote Location

Persistence Service can be controlled remotely by sending commands through a special Topic. Any Connex DDS application can be implemented to send these commands and receive the corresponding responses. A shell application that sends/receives these commands is provided with Persistence Service.

The script for the shell application is `$NDDSHOME/bin/rtipssh`.

Entering `rtipssh -help` will show you the command-line options:

```
RTI Persistence Service Shell v5.2.0
Usage: rtipssh [options]...
Options:
  -domainId <integer>   Domain ID for the remote configuration
  -timeout <seconds>    Max time to wait a remote response
  -cmdFile <file>       Run commands in this file
  -help                 Displays this information
```

29.1 Enabling Remote Administration

By default, remote administration is disabled in *Persistence Service*.

To enable remote administration you can use the `<administration>` tag (see [Configuring Remote Administration \(Section 27.5 on page 942\)](#)) or the `-remoteAdministrationDomainId` command-line parameter (see [Table 28.1 Persistence Service Command-Line Options](#)), which enables remote administration and sets the domain ID for remote communication.

When remote administration is enabled, Persistence Service will create a *DomainParticipant*, *Publisher*, *Subscriber*, *DataWriter*, and *DataReader* in the designated DDS domain. (The QoS values for these entities are described in [Configuring Remote Administration \(Section 27.5 on page 942\)](#).)

29.2 Remote Commands

This section describes the remote commands using the shell interface; [Accessing Persistence Service from a Connex DDS Application \(Section 29.3 on the facing page\)](#) explains how to use remote administration from a Connex DDS application.

Remote commands:

```
start (Section 29.2.1 below) <target_persistence_service>
stop (Section 29.2.2 below) <target_persistence_service>
shutdown (Section 29.2.3 on the facing page) <target_persistence_service>
status (Section 29.2.4 on the facing page) <target_persistence_service>
```

Parameters:

<target_persistence_service> can be:

- The application name of a persistence service, such as “**MyPersistenceService1**”, as specified at start-up with the command-line option **-appName**
- A wildcard expression¹ for a persistence service name, such as “**MyPersistenceService***”

29.2.1 start

```
start <target_persistence_service>
```

The **start** command starts the persistence service instance. DDS samples will not be persisted until the persistence service is started.

By default, the persistence service is started automatically when the process is executed. To start the service remotely use the command line option **-noAutoStart** (see [Table 28.1 Persistence Service Command-Line Options](#)).

29.2.2 stop

```
stop <target_persistence_service>
```

The **stop** command stops the persistence service instance.

An instance that has been stopped can be started again using the command start.

¹As defined by the POSIX fnmatch API (1003.2-1992 section B.6)

29.2.3 shutdown

```
shutdown <target_persistence_service>
```

The command **shutdown** stops the persistence service instance and finalizes the process

29.2.4 status

```
status <target_persistence_service>
```

The **status** command gets the status of a running persistence service instance. Possible values are STARTED and STOPPED.

29.3 Accessing Persistence Service from a Connex DDS Application

You can send commands to control an Persistence Service instance from your own Connex DDS application. You will need to create a *DataWriter* for a specific topic and type. Then, you can send a DDS sample that contains a command and its parameters. Optionally, you can create a *DataReader* for a specific topic to receive the results of the execution of your commands.

The topics are:

- rti/persistence_service/administration/command_request
- rti/persistence_service/administration/command_response

The types are:

- RTI::PersistenceService::Administration::CommandRequest
- RTI::PersistenceService::Administration::CommandResponse

You can find the IDL definitions for these types in `<NDDSHOME>/resource/idl/PersistenceServiceAdministration.idl`.

The QoS configuration of your *DataWriter* and *DataReader* must be compatible with the one used by the persistence service (see how this QoS is configured in [Configuring Remote Administration \(Section 27.5 on page 942\)](#)).

The following example in C shows how to send a command to shutdown a persistence service instance:

```

/*****
/** Create the Entities needed to send command request ****
/*****
participant = DDS_DomainParticipantFactory_create_participant(
    DDS_TheParticipantFactory, domainId,
    &DDS_PARTICIPANT_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);

```

```

if (participant == NULL)
{ /* Error */ }
if (publisher == NULL)
{ /* Error */ }

subscriber = DDS_DomainParticipant_create_subscriber(
    participant, &DDS_SUBSCRIBER_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
publisher = DDS_DomainParticipant_create_publisher(
    participant, &DDS_PUBLISHER_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL)
{ /* Error */ }

typeName =
RTI_PersistenceService_Administration_CommandRequestTypeSupport_get_type_name();
retcode =
RTI_PersistenceService_Administration_CommandRequestTypeSupport_register_type(
    participant, typeName);
if (retcode != DDS_RETCODE_OK)
{ /* Error */ }

topicCmd = DDS_DomainParticipant_create_topic(
    participant,
    "rti/persistence_service/administration/command_request",
    typeName, &DDS_TOPIC_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
if (topicCmd == NULL)
{ /* Error */ }

typeName =
RTI_PersistenceService_Administration_CommandResponseTypeSupport_get_type_name();
retcode =
RTI_PersistenceService_Administration_CommandResponseTypeSupport_register_type(
    participant, typeName);
if (retcode != DDS_RETCODE_OK)
{ /* Error */ }

topicResponse = DDS_DomainParticipant_create_topic(
    participant,
    "rti/persistence_service/administration/command_response",
    typeName, &DDS_TOPIC_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);
if (topicResponse == NULL)
{ /* Error */ }

writerQos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
writerQos.history.kind = DDS_KEEP_ALL_HISTORY_QOS;
writer = DDS_Publisher_create_datawriter(
    publisher, topicCmd, &writerQos,
    NULL /* listener */,
    DDS_STATUS_MASK_NONE);
if (writer == NULL)
{ /* Error */ }

readerQos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

```

```

readerQos.history.kind      = DDS_KEEP_ALL_HISTORY_QOS;
reader = DDS_Subscriber_create_datareader(
    subscriber,
    DDS_Topic_as_topicdescription(topicResponse),
    &readerQos, NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL)
{ /* Error */ }

/*****
/** Wait for discovery *****/
/*****/
/* Wait until we discover one reader and one writer matching
 * with the command request DataWriter and the command response
 * DataReader */
while (count < maxPollPeriods)
{
    retcode = DDS_DataWriter_get_publication_matched_status(
        writer, &pubMatchStatus);
    if (retcode != DDS_RETCODE_OK)
    { /* Error */ }

    retcode = DDS_DataReader_get_subscription_matched_status(
        reader, &subMatchStatus);
    if (retcode != DDS_RETCODE_OK) { /* Error */ }

    if (pubMatchStatus.total_count == 1 &&
        subMatchStatus.total_count == 1)
    { break; }
    count++;
    NDDS_Utility_sleep(&pollPeriod);
}
if (count == maxPollPeriods)
{ /* Error */ }

/*****
/** Send the command request *****/
/*****/
request =
    RTI_PersistenceService_Administration_CommandRequestTypeSupport_create_data();
if (request == NULL)
{ /* Error */ }

/* request->id provides an unique way to identify a request so that
 * it can be correlated with a response. Although one of the fields is
 * called host it does not necessarily has to contain the IP address of
 * the host. Same applies to app */
request->id.host = 0;
request->id.app = 0;
request->id.invocation = 0;
strcpy(request->target_ps, "MyPersistenceService");
request->command._d = RTI_PERSISTENCE_SERVICE_COMMAND_SHUTDOWN;
retcode = RTI_PersistenceService_Administration_CommandRequestDataWriter_write(
    (RTI_PersistenceService_Administration_CommandRequestDataWriter *) writer,
    request, &instance_handle);
if (retcode != DDS_RETCODE_OK)
{ /* Error */ }

```

```
/*
*****
*** Wait for response
*****
*/
response =
    RTI_PersistenceService_Administration_CommandResponseTypeSupport_create_data();
if (response == NULL)
{ /* Error */ }
count = 0;
while (count < maxPollPeriods) {
    retcode =
        RTI_PersistenceService_Administration_CommandResponseDataReader_take_next_sample(
            (RTI_PersistenceService_Administration_CommandResponseDataReader*) reader,
            response, &sampleInfo);
    if (retcode == DDS_RETCODE_OK) {
        break;
    } else if (retcode != DDS_RETCODE_NO_DATA) {
        /* Error */
    }
    NDDS_Utility_sleep(&pollPeriod);
    count++;
}
if (count == maxPollPeriods) {
    printf("No response received\n");
} else {
    printf("Response received: %s\n", response->message);
}
}
```


Chapter 30 Advanced Persistence Service Scenarios

This section covers several advanced scenarios for using *Persistence Service*.

30.1 Scenario: Load-balanced Persistence Services

Each running instance of the *Persistence Service* executes as a single process in a single computer. In high-throughput scenarios the *Persistence Service* may become a bottleneck. The main reasons are:

- If the *Persistence Service* is configured to persist its DDS samples to durable storage (a disk or a database) this will further limit the throughput of DDS samples that can be persisted to what the database and/or disk can handle. Depending on computer hardware, the disk or database this limit may be in the order of tens of thousands of DDS samples per second which is far less than what could be communicated system-wide.
- Depending on the CPU there will be limits on the throughput of DDS samples that can be received by a single process.
- The computer running the *Persistence Service* is typically connected to the network via a single network interface so the data that can be persisted will be limited to the throughput that flows through a single interface which is typically far less than the aggregated throughput that can flow on the complete network.

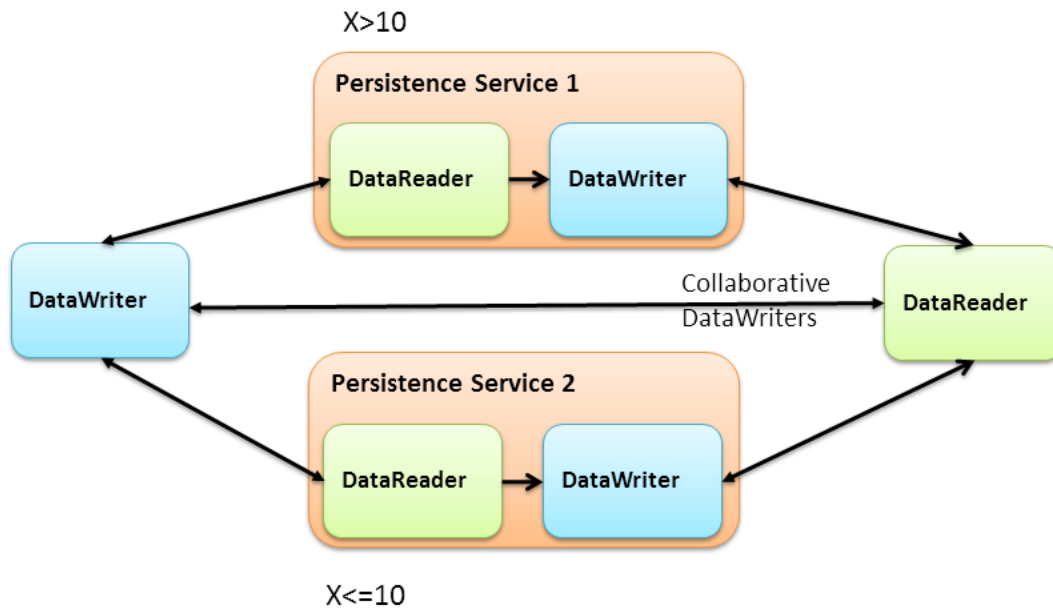
To overcome these limits multiple instances of the RTI *Persistence Service* can be run in parallel. These instances may run in multiple machines and be configured in a “load balancing” fashion such that each *Persistence Service* process is only responsible for persisting a subset of the data published on the DDS domain.

Multiple strategies for partitioning the data stored by each *Persistence Service* instance are possible:

- **Balance *Persistence Services* by Topic name.** This strategy configures each persistence service to persist different Topic names. This is accomplished by associating a filter expression with the declaration of the persistent groups used to configure each *Persistence Service* (see [Creating Persistence Groups \(Section 27.8 on page 947\)](#)). The filter expression is applied to the Topic names, so for example one *Persistence Service* could be configured with the filter “[A-Z]*” filter in the name of the Topics that it will persist and the second with the filter “[a-z]*”. With this configuration the first *Persistence Service* will persist data produced by DataWriters that specify durability TRANSIENT or PERSISTENT and have a Topic name that starts with a capital letter and the second *Persistence Service* will do the same for Topics that start with a lower-case letter.
- **Balance *Persistence Services* by data content.** In some scenarios the data published on a single Topic is too much for a single *Persistence Service* to handle. In this case the *Persistence Services* can also be configured with filter expressions based on the content of the data. This is accomplished by associating a content filter with the declaration of the persistent groups used to configure each *Persistence Service* (see [Creating Persistence Groups \(Section 27.8 on page 947\)](#)).

When multiple instances of *Persistence Service* are used to store data on the same Topic, it becomes possible for DDS samples from the same original *DataWriter* to be stored in separate instances of *Persistence Service*. In this situation, Connex DDS *DataReaders* automatically merge the data from the multiple *Persistence Services* such that the relative order of the DDS samples from the original *DataWriter* is preserved. This Connex DDS capability is called *Collaborative Datawriters* because multiple *DataWriters*, in this case the ones for different *Persistence Services*, collaborate to reconstruct the original stream. (See [Collaborative DataWriters \(Section Chapter 11 on page 670\)](#)).

Figure 30.1 Load-Balanced Persistence Services Scenario



30.2 Scenario: Delegated Reliability

The DDS-RTPS reliability protocol requires the *DataWriter* to periodically send HeartBeat messages to the *DataReaders*, process their ACKs and NACK messages, keep track of the *DataReader* state, and send the necessary repairs. The additional load caused by the reliability protocol increases with the number of reliable *DataReaders* matched with the *DataWriter*. Even if the data is sent via multicast the number of ACKs and NACKs will increase with the number of *DataReaders*.

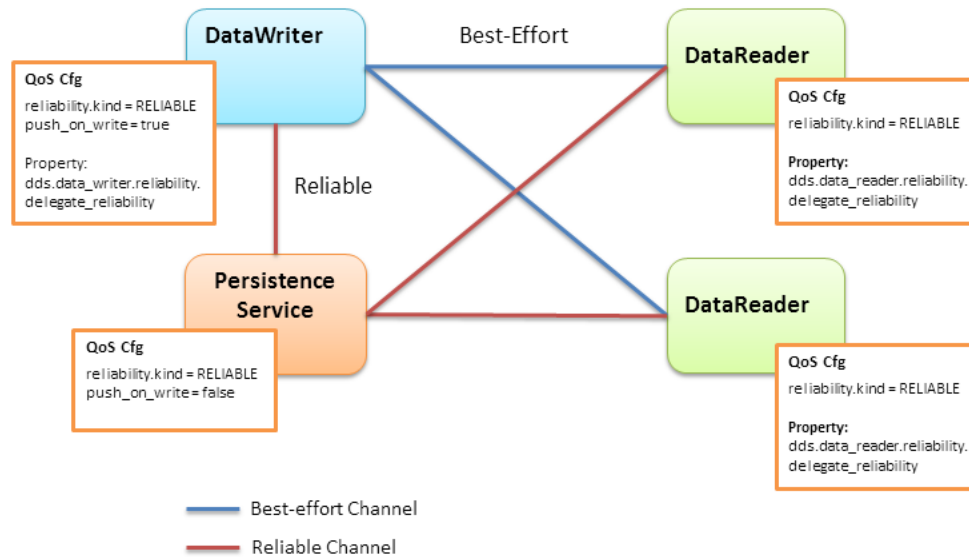
In situations where there many *DataReaders* are subscribing to the same Topic, the reliability and repair traffic may become too much for the *DataWriter* to handle and negatively impact its performance. To address this situation, Connex DDS provides the ability to configure the *DataWriter* so that it delegates the reliability task to a separate service. This approach is known as *delegated reliability*.

To take advantage of *delegated reliability*, both the original *DataWriter* and *DataReader* must be configured to enable an external service to ensure the reliability on their behalf. This is done by setting both the `dds.data_writer.reliability.delegate_reliability` property on the *DataWriter* and the `dds.data_reader.reliability.delegate_reliability` property on the *DataReader* to 1.

With this configuration, the *DataWriter* creates a reliable channel to *Persistence Service*, yet sends data using ‘best-effort’ reliability to the *DataReaders* directly. If a DDS sample is dropped, *Persistence Service* will repair the DDS sample. *Persistence Service* is configured with **push_on_write** (in the [DATA_WRITER_PROTOCOL QosPolicy \(DDS Extension\)](#) (Section 6.5.3 on page 347)) set to false. This way,

DDS samples will only be sent from *Persistence Service* to the *DataReaders* when they are explicitly NACKed by the *DataReader*.

Figure 30.2 Delegated Reliability Scenario

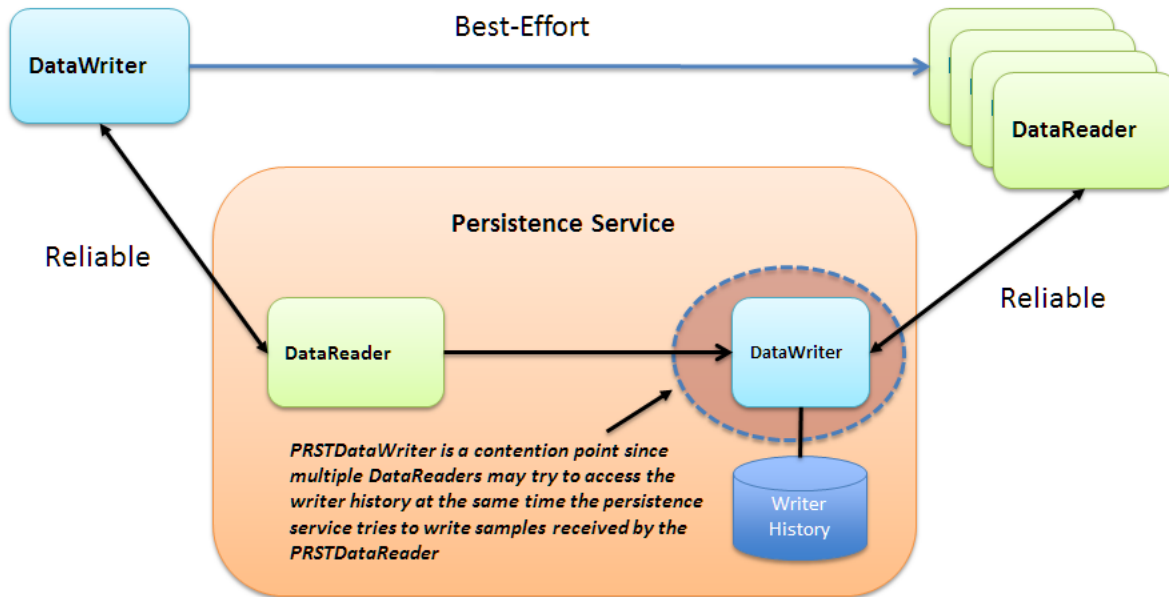


30.3 Scenario: Slow Consumer

Unless special measures are taken, the presence of slow consumers can impact the overall behavior of the system. If a *DataReader* is not keeping up with the DDS samples being sent by the *DataWriter*, it will apply back-pressure to the *DataWriter* to slow the rate at which the *DataWriter* can write DDS samples. With *delegated reliability* (see [Scenario: Delegated Reliability \(Section 30.2 on the previous page\)](#)), the original *DataWriter* can offload the processing of the ACK/NACK messages generated by the *DataReaders* to a *PRSTDataWriter*. However, the original *DataWriter* still has a reliable channel with the *PRSTDataReader* that can slow it down.

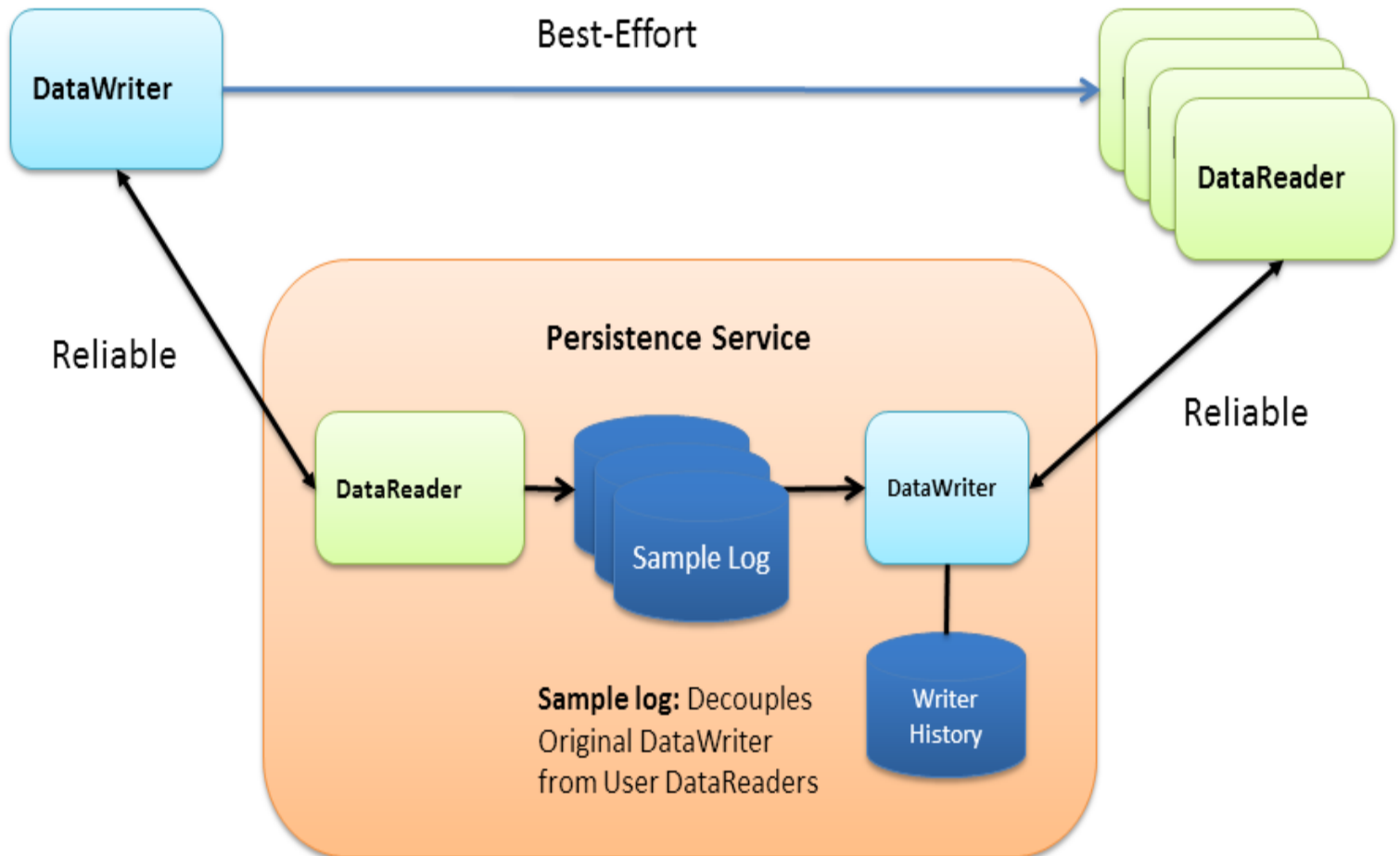
By default, *Persistence Service* uses the Connex DDS receive thread to read DDS samples from the *PRSTDataReaders*, write the DDS samples to the *PRSTDataWriters* history, and send ACKs to the original *DataWriter*. With this configuration, a *PRSTDataReader* does not ACK DDS samples to the original *DataWriter* until they are written into the corresponding *PRSTDataWriter*'s history. Since multiple *DataReaders* may be accessing the *PRSTDataWriter* history at the same time that the persistence service is trying to write new DDS samples, the *PRSTDataWriter* history becomes a contention point that can indirectly slow down the original *DataWriter* (see [Slow-Consumer Scenario with Delegated Reliability \(Section Figure 30.3 on the facing page\)](#)).

Figure 30.3 Slow-Consumer Scenario with Delegated Reliability



To remove this contention point and decouple the slow consumer from the original *DataWriter*, *Persistence Service* supports a mode where DDS samples can be buffered prior to being added to the *PRSTDataWriter*'s queue (see [Slow Consumer Scenario with Delegated Reliability and DDS Sample Log](#) (Section Figure 30.4 on the next page)).

Figure 30.4 Slow Consumer Scenario with Delegated Reliability and DDS Sample Log



If the `PRSTDataWriter` slows down due to the presence of slow consumers, the buffer will hold DDS samples such that the original `DataWriter` and the rest of the system are not impacted. This buffer is called the *Persistence Service sample log*. The persistence service creates a separate DDS sample log per `PRSTDataWriter` in the group. In addition to the DDS sample log, the persistence service creates a thread (write thread) whose main function is to read DDS samples from the log and write them to the associated `PRSTDataWriter`. There is one thread per `PRSTDataWriter`.

Persistence Service currently does not allow multiple DDS sample logs to share the same write thread.

Persistence Service can be configured to enable DDS sample logging per persistence group using the `<sample_logging>` XML tag to specify the log's configuration parameters—see [Table 30.1 Sample Logging Tags](#).

Table 30.1 Sample Logging Tags

Tags within <sample_logging>	Description	Number of Tags Allowed
<enable>	A DDS_Boolean (see Table 27.1 Supported Tag Values) that indicates whether or not DDS sample logging is enabled in the container persistence group. Default: 0	0 or 1
<log_file_size>	Specifies the maximum size of a DDS sample log file in Mbytes. When a log file becomes full, Persistence Service creates a new log file. Default: 60 MB	0 or 1
<log_flush_period>	The period (in milliseconds) at which Persistence Service removes DDS sample log files whose full content have been written into the PRSTDataWriter by the DDS sample log write thread. Default: 10000 milliseconds	0 or 1
<log_read_batch>	Determines how many DDS samples should be read and processed at once by the DDS sample log write thread. Default: 100 DDS samples	0 or 1
<log_bookmark_period>	DDS samples in the DDS sample log are identified by two attributes: <ul style="list-style-type: none"> The file ID The row ID (position within the file) The read bookmark indicates the most recently processed DDS sample. This tag indicates how often (in milliseconds) the read bookmark is persisted into disk. Default: 1000 milliseconds	0 or 1

Enabling DDS sample logging in a persistence group is expensive. For every PRSTDataWriter, *Persistence Service* will create a write thread and an event thread that will be in charge of flushing the log files and storing the read bookmark. Therefore, DDS sample logging should be enabled only for the persistence groups where it is needed based on the potential presence of slow consumers and/or the expected data rate in the persistence group. Small data rates will likely not require a DDS sample log.

Part 7: RTI CORBA Compatibility Kit

The material in this part of the manual is only relevant if you have purchased the *CORBA Compatibility Kit*, an optional package that allows Connex DDS's code generator, *RTI Code Generator*, to output type-specific code that is compatible with OCI's distribution of TAO and the JacORB distribution.

This section includes:

- [Introduction to RTI CORBA Compatibility Kit \(Section Chapter 31 on page 980\)](#)
- [Generating CORBA-Compatible Code \(Section Chapter 32 on page 982\)](#)
- [Supported IDL Types \(Section Chapter 33 on page 985\)](#)

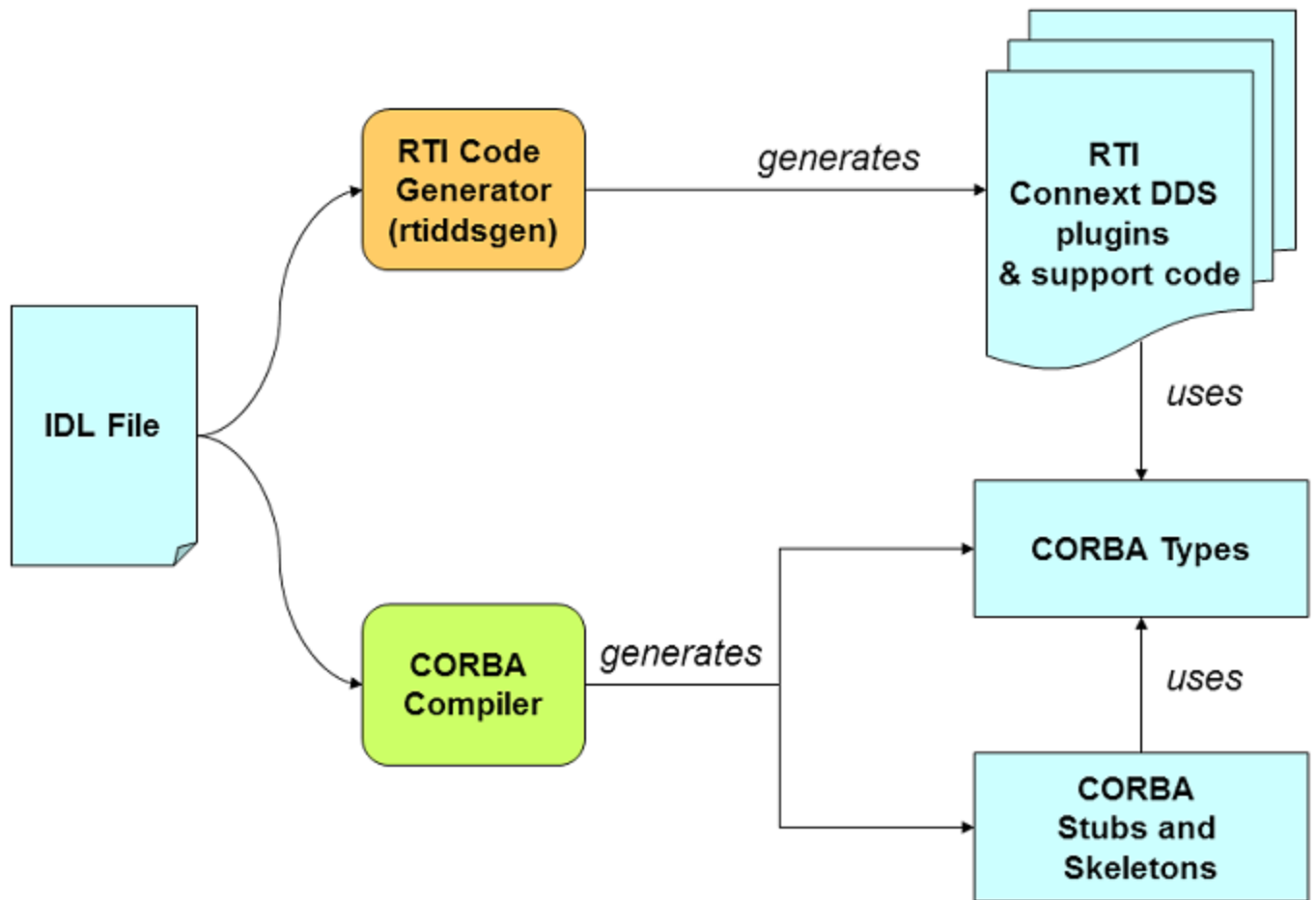
Chapter 31 Introduction to RTI CORBA Compatibility Kit

RTI CORBA Compatibility Kit is an optional package that allows the *RTI Code Generator* to output type-specific code that is compatible with OCI's or DOC's distribution of TAO and the JacORB distribution.

By having compatible data types, your applications can use CORBA and *Connex DDS* APIs, with no type conversions required.

For more information about OCI's or DOC's distribution of TAO and JacORB, please refer to the documentation included with those distributions. Additional information can be found on OCI's TAO website (www.theaceorb.com), DOC's TAO website (www.dre.vanderbilt.edu), and JacORB's website (www.jacorb.org). TAO and JacORB distributions that are compatible with this version of *Connex DDS* are available from the RTI Support Portal, accessible from <https://support.rti.com>.

This figure shows the process of using IDL files and types that are shared with CORBA:



CORBA Compatibility Kit is designed to be installed on top of Connexx DDS; this kit enables *RTI Code Generator* to support these CORBA-specific command-line options:

```

[-corba [CORBA Client header file]]
[-dataReaderSuffix <suffix>]
[-dataWriterSuffix <suffix>]
[-orb <CORBA ORB>]
[-typeSequenceSuffix <suffix>]
  
```

The above options are described in the *RTI Code Generator User's Manual*.

On the wire, the serialized version of the code for types generated using the **-corba** option is identical to the serialized version of the code for types generated without the option. As result, endpoints (*DataReaders* or *DataWriters*) using type-support code generated with **-corba** can fully communicate with endpoints using type-support code generated without **-corba**.

Chapter 32 Generating CORBA- Compatible Code

The CORBA Compatibility Kit enables *RTI Code Generator* to produce type-specific code that is compatible with OCI's distribution of TAO for C++ and with JacORB for Java.

When using *RTI Code Generator*, specify the **-corba** option on the command line to generate compatible code. The **-corba** option enables the use of data structures for both CORBA and Connex DDS API calls without requiring any translation: the IDL-to-language mapping is the same for both.

There are some trade-offs to consider:

- While the **-corba** option provides the benefit of CORBA-compatible type-specific code, it does not provide support for bit fields, pointers and ValueTypes.
- For complex types such as sequences and strings, the memory management is different when the **-corba** option is used. When code is generated without the option, the memory needed for the type is pre-allocated at system initialization. When code is generated with the option, the memory is allocated when it is needed, so memory allocation system calls may occur while the system is in steady state.
- Without the **-corba** option, access to data fields within types may be faster under some circumstances. CORBA-compatible types require the use of accessor methods. When **-corba** is *not* used, while the accessor methods are provided for convenience, they can be bypassed and the data can be accessed directly. This direct access is available to the user as well as to the Connex DDS internal implementation code. As a result, depending on the complexity of the types used, overall system latency could be lower when using non-compatible types (that is, when **-corba** is not used).

The following sections describe how to use the CORBA Compatibility Kit. In addition to these instructions, a simple example is available.

By default, examples are copied into your home directory the first time you run *RTI Launcher* or any script in `<NDDSHOME>/bin`. This document refers to the location of the copied examples as `<path to examples>`.

Wherever you see `<path to examples>`, replace it with the appropriate path.

Default path to the examples:

- Mac OS X systems: `/Users/your user name/rti_workspace/5.2.3/examples`
- UNIX-based systems: `/home/your user name/rti_workspace/5.2.3/examples`
- Windows systems: `your Windows documents folder\rti_workspace\5.2.3\examples`

Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 7, the folder is `C:\Users\your user name\Documents`; on Windows Server 2003, the folder is `C:\Documents and Settings\your user name\Documents`.

Note: You can specify a different location for `rti_workspace`. You can also specify that you do not want the examples copied to the workspace. For details, see *Controlling Location for RTI Workspace and Copying of Examples* in the *Connex DDS Core Libraries Getting Started Guide*.

- **C++ using TAO:**
 - [Generating Java Code \(Section 32.2 on the facing page\)](#)
 - See the example in `<path to examples>/corba/c++` and read **Instructions.pdf**.
- **Java using JacORB:**
 - [Generating Java Code \(Section 32.2 on the facing page\)](#)
 - See the example in `<path to examples>/corba/java` and read **Instructions.pdf**.

32.1 Generating C++ Code

To generate CORBA-compatible type-specific code, first run TAO's code generator, `tao_idl`, on the IDL file containing your data types. If you followed the TAO distribution compilation instructions contained in this document, the `tao_idl` compiler executable will be in the TAO install directory under `<ACE_ROOT>/bin`.

```
<ACE_ROOT>/bin/tao_idl <IDL file name>.idl
```

This will generate CORBA support files for your data types. The generated file will have a name matching the pattern `<IDL file name>C.h` and will contain the type definitions. Pass this header file as a parameter to `rtiddsgen` to generate the Connex DDS support code for the data types.

```
rtiddsgen -language C++ -corba <IDL file name>C.h -example \  
<architecture> <IDL file name>.idl
```

The optional **-example <architecture>** flag will generate code for a publisher and a subscriber. It will also generate an **.mpc** file (and an **.mwc** file for Windows) that can be used with TAO's Makefile, Project and Workspace Creator (MPC) to generate a makefile or a Visual Studio project file for your DDS-CORBA application. The **.mpc** file is meant to work out-of-the-box with the DDS-CORBA C++ Message example only, so you will have to modify it to compile your custom application. Please refer to the DDS-CORBA C++ example for more information about using MPC (see the Instructions document).

32.2 Generating Java Code

To generate Java CORBA-compatible type specific code, first run the JacORB code generator on the IDL file containing your data types.

```
<JacORB install dir>/bin/idl <IDL file name>.idl
```

After generating the CORBA code for the IDL types run *rtiddsgen* as follows:

```
rtiddsgen -language Java -corba -example <architecture> \  
<IDL file name>.idl
```

The optional **-example <architecture>** flag will generate code for a DDS publisher and a DDS subscriber. It will also generate a makefile specific to your architecture that can be used to compile the example using the publisher and subscriber code generated.

To form a complete code set, use the type class generated by the CORBA IDL compiler and the files generated by *RTI Code Generator*.

Chapter 33 Supported IDL Types

Table 33.1 Supported IDL Types when Using `rtiddsgen -corba` lists the IDL types supported when using the `-corba` option.

Table 33.1 Supported IDL Types when Using `rtiddsgen -corba`

IDL Construct	Support
Modules	Supported
Interfaces	Ignored
Constants	Supported
Basic Data Types	Supported
Enums	Supported
String Types	Supported
Wide String Types	Supported
Struct Types	Supported Note: In-line nested structures are not supported (whether using <code>-corba</code> or not). See Note 1 (Section on the next page) .
Fixed Types	Ignored
Union Types	Supported
Sequence Types	Supported Note: Sequences of anonymous sequences are not supported. See Note 2 (Section on the next page) .

Table 33.1 Supported IDL Types when Using rtiddsgen -corba

IDL Construct	Support
Array Types	Supported
Typedefs	Supported
Any	Not Supported. Note that <i>RTI Code Generator</i> does not ignore them. This construct cannot be in the IDL file.
Value Types	Ignored
Exception Types	Ignored
Type Code	Supported <i>RTI Code Generator</i> generates Connex DDS TypeCodes CORBA TypeCodes are generated by the CORBA IDL compiler

Note 1

Inline nested structures, such as the following example, are *not* supported.

```

struct Outer {
    short outer_short;
    struct Inner {
        char inner_char;
        short inner_short;
    } outer_nested_inner;
};

```

Note 2

Sequences of anonymous Sequences are *not* supported. This kind of type will be banned in future revisions of CORBA. For example, the following is *not* supported:

```

sequence<sequence<short, 4>, 4> MySequence;

```

Instead, sequences of sequences can be supported using typedef definitions. For example, this *is* supported:

```

typedef sequence<short, 4> MyShortSequence;
sequence<MyShortSequence, 4> MySequence;

```

Part 8: RTI TCP Transport

RTI TCP Transport is only available on specific architectures. See the [RTI Connex DDS Core Libraries Platform Notes](#) for details.

Out of the box, Connex DDS uses the UDPv4 and Shared Memory transport to communicate with other DDS applications. This configuration is appropriate for systems running within a single LAN. However, using UDPv4 introduces some problems when Connex DDS applications in different LANs need to communicate:

- UDPv4 traffic is usually filtered out by the LAN firewalls for security reasons.
- Forwarded ports are usually TCP ports.
- Each LAN may run in its own private IP address space and use NAT (Network Address Translation) to communicate with other networks.

TCP Transport enables participant discovery and data exchange using the TCP protocol (either on a local LAN, or over the public WAN). *TCP Transport* allows Connex DDS to address the challenges of using TCP as a low-level communication mechanism between peers and limits the number of ports exposed to one. (When using the default UDP transport, a Connex DDS application uses multiple UDP ports for communication, which may make it unsuitable for deployment across firewalled networks).

Chapter 34 TCP Communication Scenarios

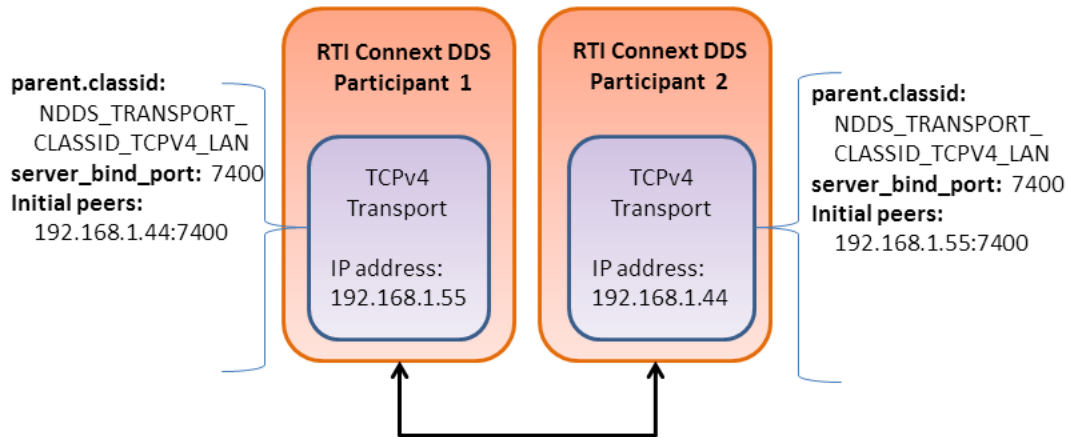
TCP Transport can be used to address multiple communication scenarios—from simple communication within a single LAN, to complex communication scenarios across LANs where NATs and firewalls may be involved. This section describes these scenarios:

- [Communication Within a Single LAN \(Section 34.1 below\)](#)
- [Symmetric Communication Across NATs \(Section 34.2 on the next page\)](#)
- [Asymmetric Communication Across NATs \(Section 34.3 on page 990\)](#)

34.1 Communication Within a Single LAN

TCP Transport can be used as an alternative to UDPv4 to communicate Connex DDS applications running inside the same LAN. [Figure 34.1 Communication within a Single LAN on the next page](#) shows how to configure the TCP transport in this scenario.

Figure 34.1 Communication within a Single LAN



- [parent.classid](#) (Section on page 1004) and [server_bind_port](#) (Section on page 1009) are transport properties configured using the PropertyQosPolicy of the participant. (Note: When the TCP transport is instantiated, by default it is configured to work in a LAN environment using symmetric communication and binding to port 7400 for incoming connections.) For additional information about these properties, see [Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t](#).
- Initial Peers represents the peers to which the participant will be announced to. Usually, these peers are configured using the DiscoveryQosPolicy of the participant or the environment variable `NDDS_DISCOVERY_PEERS`. For information on the format of initial peers, see [Choosing a Transport Mode](#) (Section 35.1.1 on page 993).

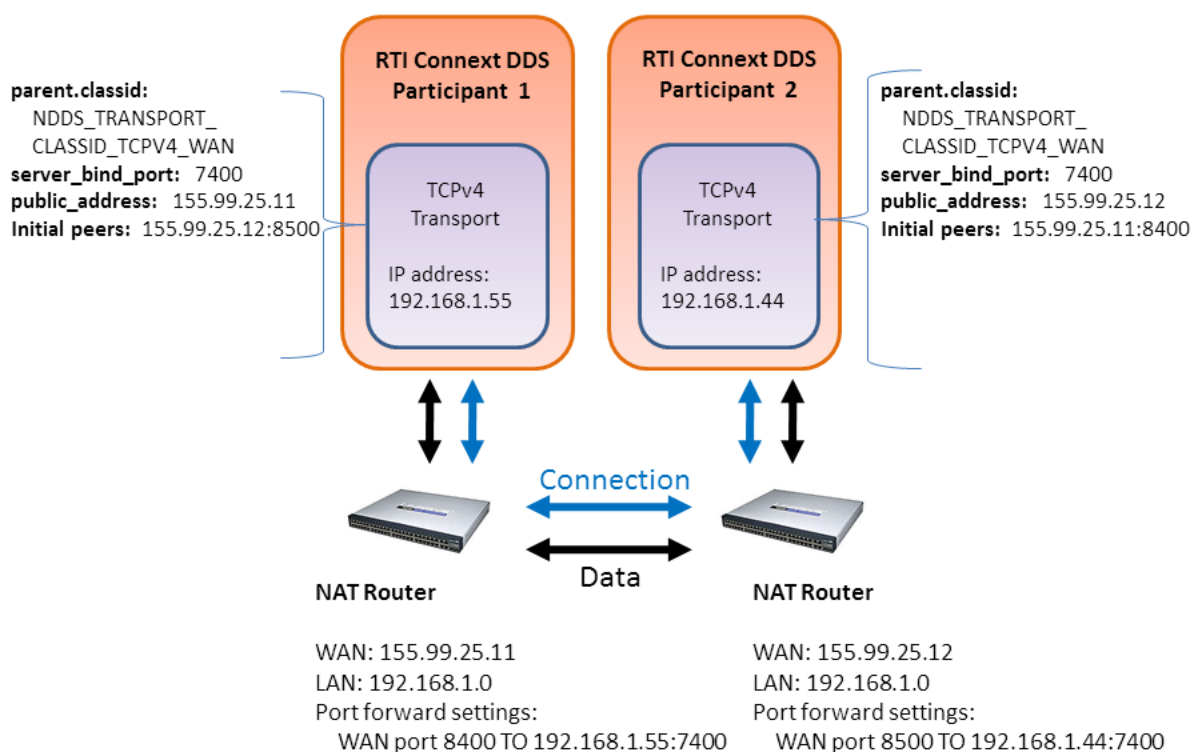
Unlike the UDPv4 transport, **you must specify the initial peers**, because multicast cannot be used with TCP.

34.2 Symmetric Communication Across NATs

In NAT communication scenarios, each one of the LANs has a private IP address space. The communication with other LANs is done through NAT routers that translate private IP addresses and ports into public IP addresses and ports.

In symmetric communication scenarios, any Connex DDS application can initiate TCP connections with other applications. [Figure 34.2 Symmetric Communication Across NATs on the facing page](#) shows how to configure the TCP transport in this scenario.

Figure 34.2 Symmetric Communication Across NATs



Notice that initial peers refer to the public address of the remote LAN where the Connex DDS application is deployed and not the private address of the node where the application is running. In addition, the transport associated with a Connex DDS instance will have to be configured with its public address ([public_address \(Section on page 1008\)](#)) so that this information can be propagated as part of the discovery process.

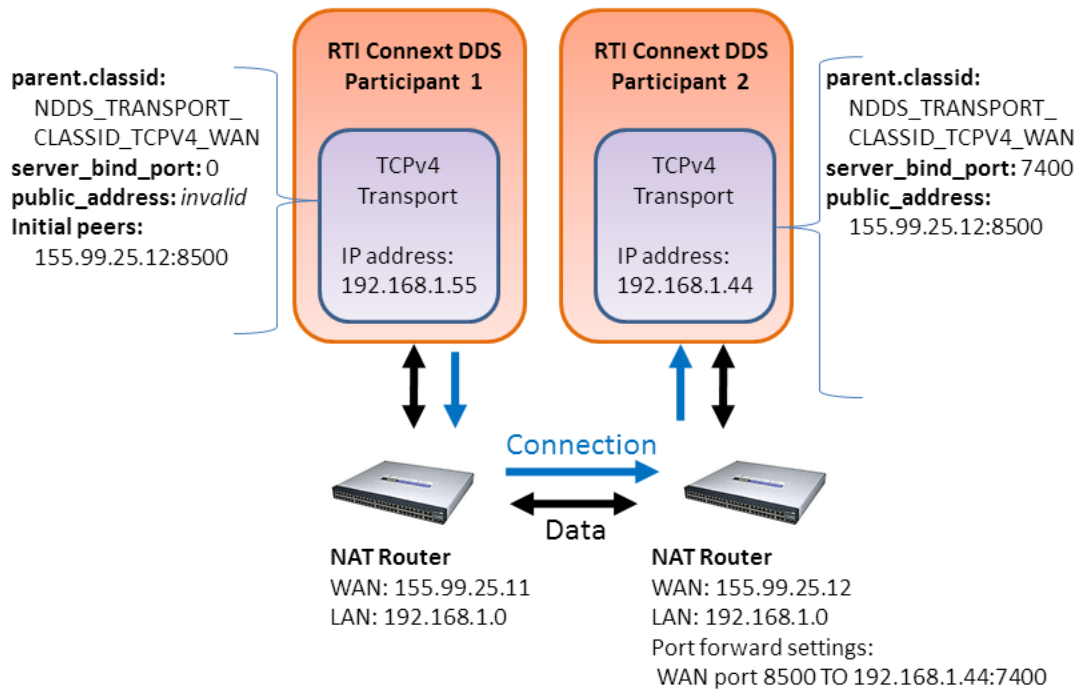
Because the public address and port of the Connex DDS instances must be known before the communication is established, the NAT Routers will have to be configured statically to translate (forward) the private [server_bind_port \(Section on page 1009\)](#) into a public port. This process is known as static NAT or port forwarding; it allows traffic originating in outer networks to reach designated peers in the LAN behind the NAT router. You will need to refer to your router's configuration manual to understand how to correctly set up port forwarding.

34.3 Asymmetric Communication Across NATs

This scenario is similar to the previous one, except in this case the TCP connections can be initiated only by the Connex DDS instance in LAN1. For security reasons, incoming connections to LAN1 are not allowed. In this case, the peer in LAN1 is considered 'unreachable.' Unreachable peers can publish and subscribe just like any other peer, but communication can occur only to a 'reachable' peer.

Figure 34.3 [Asymmetric Communication Across NATs below](#) shows how to configure the TCP transport in this scenario. Notice that the transport property `server_bind_port` is set to 0 to configure the node as unreachable.

Figure 34.3 Asymmetric Communication Across NATs



In an asymmetric configuration, an unreachable peer (that is behind a firewall or NAT without port forwarding) can still publish and subscribe like a reachable peer, but with some important limitations:

- An unreachable peer can only communicate with reachable peers: two unreachable peers cannot establish a direct communication since they are both behind a firewall and/or NAT.

Note that since Connex DDS always relies on a direct connection between peers (even if there is a third node that can be reachable by both unreachable peers), **communication can never occur between unreachable peers**. For example, suppose Peers A and B are unreachable and Peer C is reachable. Communication can take place between A and C, and between B and C, but not between A and B. For this configuration, you should consider using *RTI Federation Service* (available for purchase as a separate product).

- It can take longer to discover unreachable peers than reachable ones. This is because a reachable peer has to wait for the unreachable peer to establish the communication first.

For example, suppose Peer A (unreachable) starts before Peer B (reachable). The discovery mechanism of A attempts to connect to the (not-yet existing) Peer B. Since it fails, it will retry after n

seconds. Right after that, B starts. If A would be reachable (and in B's peer list), the discovery mechanism will immediately contact A. In this case, since A cannot be reached, B needs to wait until the discovery process of A decides to retry.

This effect can be minimized by modifying the QoS that controls the discovery mechanism used by A. In particular, you should set the *DomainParticipant's* DiscoveryConfig QoS policy's **min_initial_participant_announcement_period** to a small value.

Note that the concept of symmetric/asymmetric configuration is a local concept that only describes the communication mechanism between two peers. A reachable peer can be involved in symmetric communication with another reachable peer, and at the same time have asymmetric communication with a unreachable peer. When a peer attempts to communicate with a remote peer, it knows if the remote peer is reachable or not by looking at the transport address provided.

35.1 Configuring the TCP Transport

TCP Transport is distributed as a both shared and static library in `<NDDSHOME>/lib/<architecture>`. The library is called `nddstransporttcp`.

Mechanisms for Configuring the Transport:

By explicitly instantiating a new transport (see [Explicitly Instantiating the TCP Transport Plugin \(Section 35.1.2 on the next page\)](#)) and then registering it with the *DomainParticipant* (see [Installing Additional Builtin Transport Plugins with register_transport\(\) \(Section 15.7 on page 765\)](#)). (Not available in the Java and .NET APIs.)

Through the Property QoS policy of the *DomainParticipant* (on UNIX, Solaris and Windows systems only). This process is described in [Configuring the TCP Transport with the Property QoS Policy \(Section 35.1.3 on page 996\)](#).

This section describes:

[Choosing a Transport Mode \(Section 35.1.1 below\)](#)

[Explicitly Instantiating the TCP Transport Plugin \(Section 35.1.2 on the next page\)](#)

[Configuring the TCP Transport with the Property QoS Policy \(Section 35.1.3 on page 996\)](#)

[Setting the Initial Peers \(Section 35.1.4 on page 999\)](#)

[Support for External Hardware Load Balancers in TCP Transport Plugin \(Section 35.1.5 on page 1000\)](#)

[TCP/TLS Transport Properties \(Section 35.1.6 on page 1002\)](#)

35.1.1 Choosing a Transport Mode

When you configure the TCP transport, you must choose one of the following types of communication:

TCP over LAN — Communication between the two peers is not encrypted (data is written directly to a TCP socket). Each node can use all the possible interfaces available on that machine to receive connections. The node can only receive connections from machines that are on a local LAN.

TCP over WAN — Communication is not encrypted (data is written directly to a TCP socket). The node can only receive connections from a specific port, which must be configured in the public router of the local network (WAN mode).

TLS over LAN — This is similar to the TCP over LAN, where the node can use all the available network interfaces to TX/RX data (LAN nodes only), but in this mode, the data being written on the physical socket is encrypted first (through the **openssl** library). Performance (throughput and latency) may be less than TCP over LAN since the data needs to be encrypted before going on the wire. Discovery time may be longer with this mode because when the first connection is established, the two peers exchange handshake information to ensure line protection. For more general information on TLS, see [Datagram Transport-Layer Security \(DTLS\) \(Section 24.3 on page 908\)](#).

TLS over WAN — The data is encrypted just like TLS over LAN, but it can be sent and received only from a specific port of the router.

Note: To use either TLS mode, you also need *RTI* TLS Support, which is available for purchase as a separate package.

An instance of the transport can only communicate with other nodes that use the same transport mode.

You can specify the transport mode in either the `NDDS_Transport_TCPv4_Property_t` structure (see [TCP/TLS Transport Properties \(Section 35.1.6 on page 1002\)](#)) or in the `parent.classid` ([Section on page 1004](#)) field of the `Properties QoS` (see [Configuring the TCP Transport with the Property QoSPolicy \(Section 35.1.3 on page 996\)](#)). Your choice of transport mode will also be reflected in the prefix you use for setting the initial peers (see [Setting the Initial Peers \(Section 35.1.4 on page 999\)](#)).

35.1.2 Explicitly Instantiating the TCP Transport Plugin

As described on [Page 993](#), there are two ways to configure a transport plugin. This section describes the way that includes explicitly instantiating and registering a new transport. (The other way is to use the Property QoS mechanism, described in [Configuring the TCP Transport with the Property QoSPolicy \(Section 35.1.3 on page 996\)](#)).

Notes:

This way of instantiating a transport is not supported in the Java and .NET APIs. If you are using Java or .NET, use the Property QoS mechanism described in [Configuring the TCP Transport with the Property QoSPolicy \(Section 35.1.3 on page 996\)](#).

To use this mechanism, there are **extra libraries that you must link into your program and an additional header file** that you must include. Please see [Additional Header Files and Include Directories \(Sec-](#)

tion [35.1.2.1 on the facing page](#)) and [Additional Libraries and Compiler Flags \(Section 35.1.2.2 below\)](#) for details.

To instantiate a TCP transport:

Include the extra header file described in [Additional Header Files and Include Directories \(Section 35.1.2.1 below\)](#).

Instantiate a new transport by calling `NDDS_Transport_TCPv4_new()`:

```
NDDS_Transport_Plugin* NDDS_Transport_TCPv4_new (
    const struct NDDS_Transport_TCPv4_Property_t * property_in)
```

Register the transport by calling `NDDSTransportSupport::register_transport()`.

See the API Reference HTML documentation for details on these functions and the contents of the `NDDS_Transport_TCPv4_Property_t` structure.

35.1.2.1 Additional Header Files and Include Directories

To use the TCP Transport API, you must include an extra header file (in addition to those in [Table 9.1 Header Files to Include for Connex DDS \(All Architectures\)](#)):

```
#include "ndds/transport_tcp/transport_tcp_tcpv4.h"
```

Since TCP Transport is in the same directory as Connex DDS (see [Table 9.2 Include Paths for Compilation \(All Architectures\)](#)), no additional include paths need to be added for the TCP Transport API. If this is not the case, you will need to specify the appropriate include path.

35.1.2.2 Additional Libraries and Compiler Flags

To use the TCP Transport, you must add the `nddstransporttcp` library to the link phase of your application. There are four different kind of libraries, depending on if you want a debug or release version, and static or dynamic linking with Connex DDS.

For UNIX- based systems, the libraries are:

- `libnddstransporttcp.a` — Release version, dynamic libraries
- `libnddstransporttcpd.a` — Debug version, dynamic libraries
- `libnddstransporttcpz.a` — Release version, static libraries
- `libnddstransporttcpzd.a` — Debug version, static libraries

For Windows-based systems, the libraries are:

- **NDDSTRANSPORTTCP.LIB** — Release version, dynamic libraries
- **NDDSTRANSPORTTCPD.LIB** — Debug version, dynamic libraries
- **NDDSTRANSPORTTCPZ.LIB** — Release version, static libraries
- **NDDSTRANSPORTTCPZD.LIB** — Debug version, static libraries

Notes for using TLS:

To use either TLS mode (see [Choosing a Transport Mode \(Section 35.1.1 on page 993\)](#)), you also need *RTI* TLS Support, which is available for purchase as a separate package. The TLS library (**libnddstls.so** or **NDDSTLS.LIB**, depending on your platform) must be in your library search path (pointed to by the environment variable `LD_LIBRARY_PATH` on UNIX/Solaris systems, `Path` on Windows systems, `LIBPATH` on AIX systems, `DYLD_LIBRARY_PATH` on Mac OS systems).

If you already have `$NDDSHOME/lib/<architecture>` in your library search path, no extra steps are needed to use TLS once TLS Support is installed.

Even if you link everything statically, you must make sure that the location for `$NDDSHOME/lib/<architecture>` (or wherever the TLS library is located) is in your search path. When the TCP Transport Plugin is explicitly instantiated, the TLS library is loaded dynamically, even if you use static linking for everything else. To load TLS libraries statically, please see [Configuring the TCP Transport with the Property QoSPolicy \(Section 35.1.3 below\)](#).

Your search path must also include the location for the OpenSSL library, which is used by the TLS library.

35.1.3 Configuring the TCP Transport with the Property QoSPolicy

The [PROPERTY QoSPolicy \(DDS Extension\) \(Section 6.5.17 on page 394\)](#) allows you to set up name/value pairs of data and attach them to an entity, such as a *DomainParticipant*.

Like all QoS policies, there are two ways to specify the Property QoS policy:

Programmatically, as described in this section and [Getting, Setting, and Comparing QoS Policies \(Section 4.1.7 on page 158\)](#). This includes using the `add_property()` operation to attach name/value pairs to the Property QoSPolicy and then configuring the *DomainParticipant* to use that QoSPolicy (by calling `set_qos()` or specifying QoS values when the *DomainParticipant* is created).

With an XML QoS Profile, as described in [Configuring QoS with XML \(Section Chapter 17 on page 791\)](#). This causes Connex DDS to dynamically load the TCP transport library at run time and then implicitly create and register the transport plugin.

To add name/value pairs to the Property QoS policy, use the `add_property()` operation:

```
DDS_ReturnCode_t DDSPropertyQoSPolicyHelper::add_property
(DDS_PropertyQoSPolicy policy, const char * name,
 const char * value, DDS_Boolean propagate)
```

For more information on `add_property()` and the other operations in the `DDSPROPERTYQOSPolicyHelper` class, see [Table 6.57 PropertyQoS Policy Helper Operations](#), as well as the API Reference HTML documentation.

The ‘name’ part of the name/value pairs is a predefined string. The property names for the TCP Transport are described in [Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t](#).

Here are the basic steps, taken from the example Hello World application (for details, please see the example application.)

Get the default *DomainParticipant* QoS from the `DomainParticipantFactory`.

```
DDSDomainParticipantFactory::get_instance()->
    get_default_participant_qos(participant_qos);
```

Disable the builtin transports.

```
participant_qos.transport_builtin.mask =
    DDS_TRANSPORTBUILTIN_MASK_NONE;
```

Set up the *DomainParticipant*’s Property QoS.

Load the plugin.

```
DDSPROPERTYQOSPolicyHelper::add_property (
    participant_qos.property,
    "dds.transport.load_plugins",
    "dds.transport.TCPv4.tcp1",
    DDS_BOOLEAN_FALSE);
```

Specify the transport plugin library.

```
DDSPROPERTYQOSPolicyHelper::add_property (
    participant_qos.property,
    "dds.transport.TCPv4.tcp1.library",
    "nddstransporttcp",
    DDS_BOOLEAN_FALSE);
```

Specify the transport’s ‘create’ function.

```
DDSPROPERTYQOSPolicyHelper::add_property (
    participant_qos.property,
    "dds.transport.TCPv4.tcp1.create_function",
    "NDDS_Transport_TCPv4_create", DDS_BOOLEAN_FALSE);
```

Set the transport to work in a WAN configuration with a public address:

```

DDSPropertyQoSPolicyHelper::add_property (
    participant_qos.property,
    "dds.transport.TCPv4.tcp1.parent.classid",
    "NDDS_TRANSPORT_CLASSID_TCPV4_WAN", DDS_BOOLEAN_FALSE);

DDSPropertyQoSPolicyHelper::add_property (
    participant_qos.property,
    "dds.transport.TCPv4.public_address",
    "182.181.2.31",
    DDS_BOOLEAN_FALSE);

```

Specify any other properties, as needed.

Create the *DomainParticipant* using the modified QoS.

```

participant =
    DDSTheParticipantFactory->create_participant (
        domainId,
        participant_qos,
        NULL /* listener */,
        DDS_STATUS_MASK_NONE);

```

Property changes should be made before the transport is loaded—either before the *DomainParticipant* is enabled, before the first *DataWriter/DataReader* is created, or before the builtin topic reader is looked up, whichever one happens first.

35.1.3.1 Configuring the TCP Transport to be Loaded Statically

Similar to the previous example, here are the basic steps to load the TCP Transport Plugin statically.

1. Get the default *DomainParticipant* QoS from the *DomainParticipantFactory*.

```

DDSDomainParticipantFactory::get_instance()->
    get_default_participant_qos(participant_qos);

```

2. Disable the builtin transports.

```

participant_qos.transport_builtin.mask =
    DDS_TRANSPORTBUILTIN_MASK_NONE;

```

3. Set up the *DomainParticipant's* Property QoS.

- a. Load the plugin.

```

DDSPropertyQoSPolicyHelper::add_property
(participant_qos.property,
    "dds.transport.load_plugins",
    "dds.transport.TCPv4.tcp1", DDS_BOOLEAN_FALSE);

```

- b. Specify the transport's 'create' function pointer.

```
DDSPROPERTYQOSPolicyHelper::add_pointer_property
(participant_qos.property,
 "dds.transport.TCPv4.tcpl.create_function_ptr",
 (void*)NDDS_Transport_TCPv4_create);
```

- c. Set the transport to work in a WAN configuration with a public address:

```
DDSPROPERTYQOSPolicyHelper::add_property
(participant_qos.property,
 "dds.transport.TCPv4.tcpl.parent.classid",
 "NDDS_TRANSPORT_CLASSID_TCPV4_WAN",
 DDS_BOOLEAN_FALSE);
DDSPROPERTYQOSPolicyHelper::add_property
(participant_qos.property,
 "dds.transport.TCPv4.tcpl.public_address",
 "182.181.2.31",
 DDS_BOOLEAN_FALSE);
```

- d. Specify any other properties, as needed.

4. Create the *DomainParticipant* using the modified QoS.

```
participant = DDSTheParticipantFactory->create_participant
(domainId, participant_qos,
 NULL /* listener */, DDS_STATUS_MASK_NONE);
```

35.1.3.2 Loading TLS Support Libraries Statically

The process to load TLS Support library statically is similar, but in this case both the `tls_create_function_ptr` and `tls_delete_function_ptr` properties need to be set.

```
DDSPROPERTYQOSPolicyHelper::add_pointer_property
(participant_qos.property,
 "dds.transport.TCPv4.tcpl.tls_create_function_ptr",
 (void*)RTITLS_ConnectionEndpointFactoryTLSv4_create);
DDSPROPERTYQOSPolicyHelper::add_pointer_property
(participant_qos.property,
 "dds.transport.TCPv4.tcpl.tls_delete_function_ptr",
 (void*)RTITLS_ConnectionEndpointFactoryTLSv4_delete);
```

35.1.4 Setting the Initial Peers

Note: You must specify the initial peers (you cannot use the defaults because multicast cannot be used with TCP).

For TCP Transport, the addresses of the initial peers (NDDS_DISCOVERY_PEERS) that will be contacted during the discovery process have the following format:

- For WAN communication using TCP: **tcpv4_wan://<IP address or hostname>:<port>**
- For WAN communication using TLS: **tlsv4_wan://<IP address or hostname>:<port>**
- For LAN communication using TCP: **tcpv4_lan://<IP address or hostname>:<port>**
- For LAN communication using TLS: **tlsv4_lan://<IP address or hostname>:<port>**

For example:

```
setenv NDDS_DISCOVERY_PEERS tcpv4_wan://10.10.1.165:7400,  
tcpv4_wan://10.10.1.111:7400,tcpv4_lan://192.168.1.1:7500
```

When the TCP transport is configured for LAN communication (with the **parent.classid** (Section on [page 1004](#)) property), the IP address is the LAN address of the peer and the port is the server port used by the transport (the **server_bind_port** (Section on [page 1009](#)) property).

When the TCP transport is configured for WAN communication (with the **parent.classid** (Section on [page 1004](#)) property), the IP address is the WAN or public address of the peer and the port is the public port that is used to forward traffic to the server port in the TCP transport.

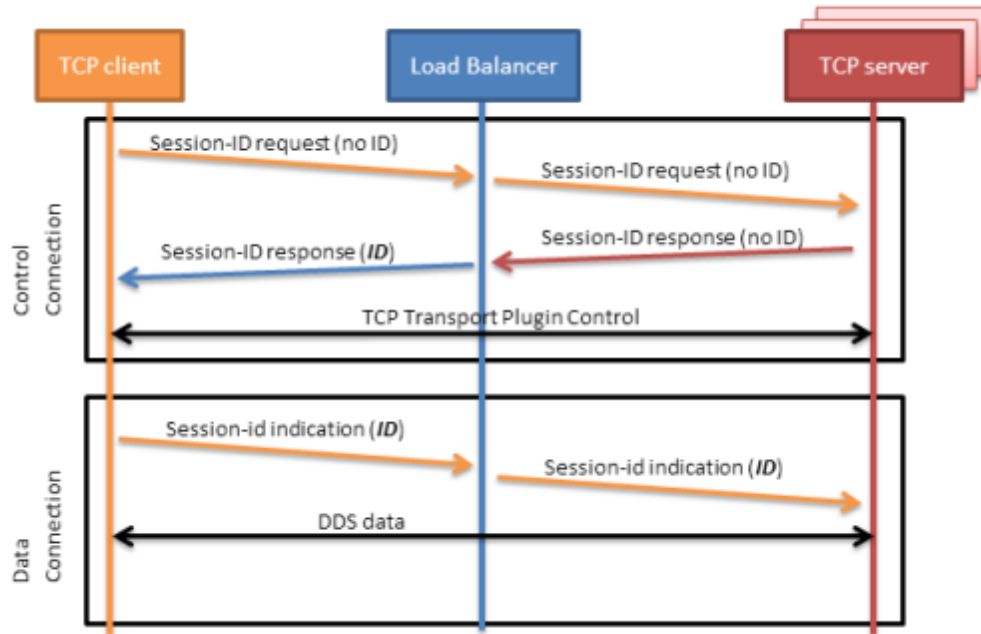
35.1.5 Support for External Hardware Load Balancers in TCP Transport Plugin

For two Connex DDS applications to communicate, the TCP Transport Plugin needs to establish 4-6 connections between the two communicating applications. The plugin uses these connections to exchange DDS data (discovery or user data) and TCP Transport Plugin control messages.

With the default configuration, the TCP Transport Plugin does not support external load balancers. This is because external load balancers do not forward the traffic to a unique TCP Transport Plugin server, but they divide the connections among multiple servers. Because of this behavior, when an application running a TCP Transport Plugin client tries to establish all the connections to an application running a TCP Transport Plugin server, the server may not receive all the required connections.

In order to support external load balancers, the TCP Transport Plugin provides a session-ID negotiation feature. When session-ID negotiation is enabled (by setting the **negotiate_session_id** property to true), the TCP Transport Plugin will perform the negotiation depicted in [Session-ID Negotiation \(Section Figure 35.1 on the facing page\)](#).

Figure 35.1 Session-ID Negotiation



During the session-ID negotiation, the TCP Transport Plugin exchanges three types of messages:

Session-ID Request: This message is sent from the client to the server. The server must respond with a session-ID response.

Session-ID Response: This message is sent from the server to the client as a response to a session-ID request. The client will store the session ID contained in this message.

Session-ID Indication: This message is sent from the client to the server; it does not require a response from the server.

The negotiation consists of the following steps:

1. The TCP client sends a session-ID request with the session ID set to zero.
2. The TCP server sends back a session-ID response with the session ID set to zero.
3. The external load balancer modifies the session-ID response, setting the session ID with a value that is meaningful to the load balancer and identifies the session.
4. The TCP client receives the session-ID response and stores the received session ID.

- For each new connection, the TCP client sends a session-ID indication containing the stored session ID. This will allow the load balancer to redirect to the same server all the connections with the same session ID.

35.1.5.1 Session-ID Messages

[TCP Payload for Session-ID Message \(Section below\)](#) depicts the TCP payload of a session-ID message. The payload consists of 48 bytes. In particular, your load balancer needs to read/modify the following two fields:

CTRLTYPE: This field allows a load balancer to identify session-ID messages. Its value (two bytes) varies according to the session-ID message type: 0x0c05 for a request, 0x0d05 for a response, or 0x0c15 for an indication.

SESSION-ID: This field consists of 16 bytes that the load balancer can freely modify according to its requirements.

TCP Payload for Session-ID Message

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
RTI reserved				0xDD	0x54	0xDD	0x55	CTRLTYPE		RTI reserved					
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RTI reserved															
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
SESSION-ID															

To ensure all the TCP connections within the same session are directed to the same server, you must configure your load balancer to perform the two following actions:

Modify the SESSION-ID field in the *session-id response* with a value that identifies the session within the load balancer.

Make the load-balancing decision according to the value of the SESSION-ID field in the session-ID indication.

35.1.6 TCP/TLS Transport Properties

[Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t](#) describes the TCP and TLS transport properties.

Note: To use TLS, you also need RTI TLS Support, which is a separate component.

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
dds.transport.load_plugins (Note: this does not take a prefix)	<p>Required</p> <p>Comma-separated strings indicating the prefix names of all plugins that will be loaded by Connex DDS. For example: "dds.transport.TCPv4.tcp1". You will use this string as the prefix to the property names.</p> <p>Note: you can load up to 8 plugins.</p>
library	<p>Only required if linking dynamically</p> <p>Must be "nddstransporttcp".</p> <p>This library must be in your library search path (pointed to by the environment variable LD_LIBRARY_PATH on UNIX/Solaris systems, Path on Windows systems, LIBPATH on AIX systems, DYLD_LIBRARY_PATH on OS X systems).</p>
create_function	<p>Only required if linking dynamically</p> <p>Must be "NDDS_Transport_TCPv4_create".</p>
create_function_ptr	<p>Only required if linking statically</p> <p>Defines the function pointer to the TCP Transport Plugin creation function. Used for loading TCP Transport Plugin statically.</p> <p>Must be set to the NDDS_Transport_TCPv4_create function pointer.</p>
tls_create_function_ptr	<p>Defines the function pointer to the TLS Support creation function. Used for loading TLS Support libraries statically.</p> <p>Must be set to the RTITLS_ConnectionEndpointFactoryTLsv4_create function pointer.</p> <p>Note: In order to have effect, the tls_delete_function_ptr property must also be set.</p>
tls_delete_function_ptr	<p>Defines the function pointer to the TLS Support deletion function. Used for loading TLS Support libraries statically.</p> <p>Must be set to the RTITLS_ConnectionEndpointFactoryTLsv4_delete function pointer.</p> <p>Note: In order to have effect, the tls_create_function_ptr property must also be set.</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for `NDDS_Transport_TCPv4_Property_t`

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
aliases	<p>Used to register the transport plugin returned by <code>NDDS_Transport_TCPv4_create()</code> (as specified by <code><TCP_prefix>.create_function</code>) to the <i>DomainParticipant</i>. Aliases should be specified as a comma-separated string, with each comma delimiting an alias.</p> <p>Default: the transport prefix</p>
parent.classid	<p>Must be set to one of the following values:</p> <p>NDDS_TRANSPORT_CLASSID_TCPV4_LAN for TCP communication within a LAN</p> <p>NDDS_TRANSPORT_CLASSID_TLSV4_LAN for TLS communication within a LAN</p> <p>NDDS_TRANSPORT_CLASSID_TCPV4_WAN for TCP communication across LANs and firewalls</p> <p>NDDS_TRANSPORT_CLASSID_TLSV4_WAN for TLS communication across LAN and firewalls</p> <p>Default: NDDS_TRANSPORT_CLASSID_TCPV4_LAN</p> <p>Note: To use either TLS mode, you also need RTI TLS Support which is available for purchase as a separate package.</p>
parent.gather_send_buffer_count_max	<p>Specifies the maximum number of buffers that Connex DDS can pass to the <code>send()</code> function of the transport plugin.</p> <p>The transport plugin <code>send()</code> API supports a gather-send concept, where the <code>send()</code> call can take several discontinuous buffers, assemble and send them in a single message. This enables Connex DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.</p> <p>However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connex DDS from trying to gather too many buffers into a send call for the transport plugin.</p> <p>Connex DDS requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is defined as NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.</p> <p>Default: 128</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for `NDDS_Transport_TCPv4_Property_t`

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
parent.message_size_max	<p>The maximum size of a message in bytes that can be sent or received by the transport plugin.</p> <p>Default: 65536</p>
parent.allow_interfaces_list	<p>A list of strings, each identifying a range of interface addresses that can be used by the transport.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>For example: 10.10.*, 10.15.*</p> <p>If the list is non-empty, this "white" list is applied before parent.deny_interfaces_list (Section below).</p> <p>Default: All available interfaces are used.</p>
parent.deny_interfaces_list	<p>A list of strings, each identifying a range of interface addresses that will not be used by the transport.</p> <p>If the list is non-empty, deny the use of these interfaces.</p> <p>Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.</p> <p>For example: 10.10.*</p> <p>This "black" list is applied after parent.allow_interfaces_list (Section above) and filters out the interfaces that should not be used.</p> <p>Default: No interfaces are denied</p>
send_socket_buffer_size	<p>Size, in bytes, of the send buffer of a socket used for sending. On most operating systems, <code>setsockopt()</code> will be called to set the <code>SENDBUF</code> to the value of this parameter.</p> <p>This value must be greater than or equal to parent.message_size_max (Section above), or -1.</p> <p>When set to -1, <code>setsockopt()</code> (or equivalent) will not be called to size the send buffer of the socket.</p> <p>The maximum value is operating system-dependent.</p> <p>Default: 131072</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for `NDDS_Transport_TCPv4_Property_t`

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
recv_socket_buffer_size	<p>Size, in bytes, of the receive buffer of a socket used for receiving.</p> <p>On most operating systems, <code>setsockopt()</code> will be called to set the <code>RCVBUF</code> to the value of this parameter.</p> <p>This value must be greater than or equal to <code>parent.message_size_max</code> (Section on the previous page), or <code>-1</code>.</p> <p>When set to <code>-1</code>, <code>setsockopt()</code> (or equivalent) will not be called to size the receive buffer of the socket.</p> <p>The maximum value is operating-system dependent.</p> <p>Default: 131072</p>
ignore_loopback_interface	<p>Prevents the transport plugin from using the IP loopback interface.</p> <p>This property is ignored when <code>parent.classid</code> (Section on page 1004) is <code>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</code> or <code>NDDS_TRANSPORT_CLASSID_TLsv4_WAN</code>.</p> <p>Two values are allowed:</p> <p>0: Enable local traffic via this plugin. The plugin will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system.</p> <p>1: Disable local traffic via this plugin. This means “do not use the IP loopback interface, even if no NICs are discovered.” This setting is useful when you want applications running on the same node to use a more efficient plugin like shared memory instead of the IP loopback.</p> <p>Default: 1</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for `NDDS_Transport_TCPv4_Property_t`

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
ignore_nonrunning_interfaces	<p>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.</p> <p>The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated" and may be off if no link is detected (e.g., the network cable is unplugged).</p> <p>Two values are allowed:</p> <p>0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.</p> <p>1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.</p> <p>Default: 1</p>
transport_priority_mask	<p>Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low (Section below)/transport_priority_mapping_high (Section below) to define the mapping from DDS transport priority to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket.</p> <p>For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 -0xff00 in this case) to the range specified by low and high.</p> <p>If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.</p> <p>Default: 0</p>
transport_priority_mapping_low	<p>Sets the low and high values of the output range to IPv4 TOS.</p> <p>These values are used in conjunction with transport_priority_mask (Section above) to define the mapping from DDS transport priority to the IPv4 TOS field. Defines the low and high values of the output range for scaling.</p>
transport_priority_mapping_high	<p>Note that IPv4 TOS is generally an 8-bit value.</p> <p>Default transport_priority_mapping_low: 0</p> <p>Default transport_priority_mapping_high: 0xFF</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
server_socket_backlog	<p>The backlog parameter determines what is the maximum length of the queue of pending connections.</p> <p>Default: 5</p>
public_address	<p>Required for WAN communication (see note below)</p> <p>Public IP address and port (WAN address and port) (separated with ':') associated with the transport instantiation.</p> <p>For example: 10.10.9.10:4567</p> <p>This field is used only when parent.classid (Section on page 1004) is NDDS_TRANSPORT_CLASSID_TCPV4_WAN or NDDS_TRANSPORT_CLASSID_TLsv4_WAN.</p> <p>The public address and port are necessary to support communication over WAN that involves Network Address Translators (NATs). Typically, the address is the public address of the IP router that provides access to the WAN. The port is the IP router port that is used to reach the private server_bind_port (Section on the facing page) inside the LAN from the outside. This value is expressed as a string in the form: ip[:port], where ip represents the IPv4 address and port is the external port number of the router.</p> <p>Host names are not allowed in the public_address because they may resolve to an internet address that is not what you want (i.e., 'localhost' may map to your local IP or to 127.0.0.1).</p> <p>Note: If you are using an asymmetric configuration, public_address does not have to be set for the non-public peer.</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
server_bind_port	<p>Private IP port (inside the LAN) used by the transport to accept TCP connections.</p> <p>If this property is set to zero, the transport will disable the internal server socket, making it impossible for external peers to connect to this node. In this case, the node is considered unreachable and will communicate only using the asymmetric mode with other (reachable) peers.</p> <p>For WAN communication, this port must be forwarded to a public port in the NAT-enabled router that connects to the outer network.</p> <p>The server_bind_port cannot be shared among multiple participants on a common host. On most operating systems, attempting to reuse the same server_bind_port for multiple participants on a common host will result in a "port already in use" error. However, Windows systems will not recognize if the server_bind_port is already in use; therefore care must be taken to properly configure Windows systems.</p> <p>Default: 7400</p>
read_buffer_allocation	<p>Allocation settings applied to read buffers.</p> <p>These settings configure the initial number of buffers, the maximum number of buffers and the buffers to be allocated when more buffers are needed.</p> <p>Default:</p> <p>read_buffer_allocation.initial_count = 2</p> <p>read_buffer_allocation.max_count = -1 (unlimited)</p> <p>read_buffer_allocation.incremental_count = -1 (number of buffers will keep doubling on each allocation until it reaches max_count)</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
write_buffer_allocation	<p>Allocation settings applied to buffers used for asynchronous (non-blocking) write.</p> <p>These settings configure the initial number of buffers, the maximum number of buffers and the buffers to be allocated when more buffers are needed.</p> <p>Default:</p> <p>write_buffer_allocation.initial_count = 4</p> <p>write_buffer_allocation.max_count = 1000</p> <p>write_buffer_allocation.incremental_count = 10</p> <p>Note that for the write buffer pool, the max_count is not set to unlimited. This is to avoid having a fast writer quickly exhaust all the available system memory, in case of a temporary network slowdown. When this write buffer pool reaches the maximum, the low-level send command of the transport will fail; at that point Connex DDS will take the appropriate action (retry to send or drop it), according to the application's QoS (if the transport is used for reliable communication, the data will still be sent eventually).</p>
control_buffer_allocation	<p>Allocation settings applied to buffers used to serialize and send control messages.</p> <p>These settings configure the initial number of buffers, the maximum number of buffers and the buffers to be allocated when more buffers are needed.</p> <p>Default:</p> <p>control_buffer_allocation.initial_count = 2</p> <p>control_buffer_allocation.max_count = -1 (unlimited)</p> <p>control_buffer_allocation.incremental_count = -1 (number of buffers will keep doubling on each allocation until it reaches max_count)</p>
control_message_allocation	<p>Allocation settings applied to control messages.</p> <p>These settings configure the initial number of messages, the maximum number of messages and the messages to be allocated when more messages are needed.</p> <p>Default:</p> <p>control_message_allocation.initial_count = 2</p> <p>control_message_allocation.max_count = -1 (unlimited)</p> <p>control_message_allocation.incremental_count = -1 (number of messages will keep doubling on each allocation until it reaches max_count)</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for `NDDS_Transport_TCPv4_Property_t`

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
control_attribute_allocation	<p>Allocation settings applied to control messages attributes.</p> <p>These settings configure the initial number of attributes, the maximum number of attributes and the attributes to be allocated when more attributes are needed.</p> <p>Default:</p> <p>control_attribute_allocation.initial_count = 2</p> <p>control_attribute_allocation.max_count = -1 (unlimited)</p> <p>control_attribute_allocation.incremental_count = -1 (number of attributes will keep doubling on each allocation until it reaches max_count)</p>
force_asynchronous_send	<p>Forces asynchronous send. When this parameter is set to 0, the TCP transport will attempt to send data as soon as the internal send() function is called. When it is set to 1, the transport will make a copy of the data to send and enqueue it in an internal send buffer. Data will be sent as soon as the low-level socket buffer has space.</p> <p>Normally setting it to 1 delivers better throughput in a fast network, but will result in a longer time to recover from various TCP error conditions. Setting it to 0 may cause the low-level send() function to block until the data is physically delivered to the lower socket buffer. For an application writing data at a very fast rate, it may cause the caller thread to block if the send socket buffer is full. This could produce lower throughput in those conditions (the caller thread could prepare the next packet while waiting for the send socket buffer to become available).</p> <p>Default: 0</p>
max_packet_size	<p>The maximum size of a TCP segment.</p> <p>This parameter is only supported on Linux architectures.</p> <p>By default, the maximum size of a TCP segment is based on the network MTU for destinations on a local network, or on a default 576 for destinations on non-local networks. This behavior can be changed by setting this parameter to a value between 1 and 65535.</p> <p>Default: -1 (default behavior)</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
enable_keep_alive	<p>Configures the sending of KEEP_ALIVE messages in TCP.</p> <p>Setting this value to 1, causes a KEEP_ALIVE packet to be sent to the remote peer if a long time passes with no other data sent or received.</p> <p>This feature is implemented only on architectures that provide a low-level implementation of the TCP keep-alive feature.</p> <p>On Windows systems, the TCP keep-alive feature can be globally enabled through the system's registry: <code>\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Tcpip\Parameters</code>.</p> <p>Refer to MSDN documentation for more details.</p> <p>On Solaris systems, most of the TCP keep-alive parameters can be changed through the kernel properties.</p> <p>Default: 0</p>
keep_alive_time	<p>Specifies the interval of inactivity in seconds that causes TCP to generate a KEEP_ALIVE message.</p> <p>This parameter is only supported on Linux and Mac architectures.</p> <p>Default: -1 (OS default value)</p>
keep_alive_interval	<p>Specifies the interval in seconds between KEEP_ALIVE retries.</p> <p>This parameter is only supported on Linux architectures.</p> <p>Default: -1 (OS default value)</p>
keep_alive_retry_count	<p>The maximum number of KEEP_ALIVE retries before dropping the connection.</p> <p>This parameter is only supported on Linux architectures.</p> <p>Default: -1 (OS default value)</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
user_timeout	<p>Changes the default OS TCP User Timeout configuration. If set to a value greater than 0, it specifies the maximum amount of time in seconds that transmitted data may remain unacknowledged before TCP will forcibly close the corresponding connection and return ETIMEDOUT to the application.</p> <p>If set to 0, TCP Transport plugin will use the system default.</p> <p>Currently this feature is supported only on Linux 2.6.37 and higher platforms.</p> <p>Default: 0 (use system's default).</p>
connection_liveliness	<p>Configures the connection liveliness feature. See Connection Liveliness (Section 35.1.6.1 on page 1020).</p> <p>Defaults:</p> <p>connection_liveliness.enable: 0</p> <p>connection_liveliness.lease_duration: 10</p> <p>connection_liveliness.assertions_per_lease_duration: 3</p>
event_thread	<p>Configures the event thread used by the TCP Transport plugin for providing some features.</p> <p>Defaults:</p> <p>event_thread.priority: THREAD_PRIORITY_DEFAULT</p> <p>event_thread.stack_size: THREAD_STACK_SIZE_DEFAULT</p> <p>event_thread.mask: PRIORITY_ENFORCE STDIO</p>
disable_nagle	<p>Disables the TCP nagle algorithm.</p> <p>When this property is set to 1, TCP segments are always sent as soon as possible, which may result in poor network utilization.</p> <p>Default: 0</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
logging_verbosity_bitmap	<p>Bitmap that specifies the verbosity of log messages from the transport.</p> <p>Logging values:</p> <ul style="list-style-type: none"> -1 (0xffffffff): do not change the current verbosity 0x00: silence 0x01: errors 0x02: warnings 0x04: local 0x08: remote 0x10: period 0x80: other (used for control protocol tracing) 0x9F: all (errors, warnings, local, remote, period, and other) <p>You can combine these values by logically ORing them together.</p> <p>Default: -1</p> <p>Note: the logging verbosity is a global property shared across multiple instances of the TCP transport. If you create a new TCP Transport instance with logging_verbosity_bitmap different than -1, the change will affect all the other instances as well.</p> <p>The default TCP transport verbosity is errors and warnings.</p> <p>Note: The option of 0x80 (other) is used only for tracing the internal control protocol. Since the output is very verbose, this feature is enabled only in the debug version of the TCP Transport library (libnddstransporttcpd.so / LIBNDDSTRANSPORTD.LIB).</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
outstanding_connection_cookies	<p>Maximum number of outstanding connection cookies allowed by the transport when acting as server.</p> <p>A connection cookie is a token provided by a server to a client; it is used to establish a data connection. Until the data connection is established, the cookie cannot be reused by the server.</p> <p>To avoid wasting memory, it is good practice to set a cap to the maximum number of connection cookies (pending connections).</p> <p>When the maximum value is reached, a client will not be able to connect to the server until new cookies become available.</p> <p>Range: 1 or higher, or -1 (which means an unlimited number).</p> <p>Default: 100</p>
outstanding_connection_cookies_life_span	<p>Maximum lifespan (in seconds) of the cookies associated with pending connections.</p> <p>If a client does not connect to the server before the lifespan of its cookie expires, it will have to request a new cookie.</p> <p>Range: 1 second or higher, or -1</p> <p>Default: -1, which means an unlimited amount of time (effectively disabling the feature).</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
send_max_wait_sec	<p>Controls the maximum time (in seconds) the low-level sendto() function is allowed to block the caller thread when the TCP send buffer becomes full.</p> <p>If the bandwidth used by the transport is limited, and the sender thread tries to push data faster than the OS can handle, the low-level sendto() function will block the caller until there is some room available in the queue. Limiting this delay eliminates the possibility of deadlock and increases the response time of the internal DDS thread.</p> <p>This property affects both CONTROL and DATA streams. It only affects SYNCHRONOUS send operations. Asynchronous sends never block a send operation.</p> <p>For synchronous send() calls, this property limits the time the DDS sender thread can block for a full send buffer. If it is set too large, Connex DDS not only won't be able to send more data, it also won't be able to receive any more data because of an internal resource mutex.</p> <p>Setting this property to 0 causes the low-level function to report an immediate failure if the TCP send buffer is full.</p> <p>Setting this property to -1 causes the low-level function to block forever until space becomes available in the TCP buffer.</p> <p>Default: 3 seconds.</p>
socket_monitoring_kind	<p>Configures the socket monitoring API used by the transport. This property can have the following values:</p> <p>SELECT: The transport uses the POSIX select API to monitor sockets.</p> <p>WINDOWS_IOCP: The transport uses Windows I/O completion ports to monitor sockets. This value only applies to Windows systems.</p> <p>WINDOWS_WAITFORMULTIPLEOBJECTS: The transport uses the API WaitForMultipleObjects to monitor sockets. This value only applies to Windows systems.</p> <p>Default: SELECT</p> <p>Note: The value selected for this property may affect transport performance and scalability. On Windows systems, using WINDOWS_IOCP provides the best performance and scalability.</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
windows_iocp	<p>Configures I/O completion ports when socket_monitoring_kind (Section on the previous page) is set to WINDOWS_IOCP.</p> <p>This setting configures the number of threads the plugin creates to process I/O completion packets (<code>thread_pool_size</code>) and the number of those threads that the operating system can allow to concurrently run (<code>concurrency_value</code>).</p> <p>Defaults:</p> <p><code>windows_iocp.thread_pool_size: 2</code></p> <p><code>windows_iocp.concurrency_value: 1</code></p>
negotiate_session_id	<p>When set to 1, the TCP Transport Plugin will perform a session negotiation that will help external load balancers identify all the connections associated with a particular session between two <i>Connex DDS</i> applications. This keeps the connections from being divided among multiple servers and ensures proper communication.</p> <p>For more information about this property, see Support for External Hardware Load Balancers in TCP Transport Plugin (Section 35.1.5 on page 1000).</p> <p>Default: 0</p> <p>Note: The value of this property must be consistent among all the applications running the TCP Transport Plugin. If two applications have a different value for this property, they may not communicate.</p>
server_connection_negotiation_timeout	<p>Specifies a timeout for the negotiation of a new connection accepted by the server.</p> <p>When the TCP Transport plugin accepts a new connection, some TCP Transport plugin-specific negotiation is exchanged between the client and the server.</p> <p>This property controls the maximum time (in seconds) the negotiation for a server connection can remain in progress. If the negotiation has not completed after the specified timeout, the connection will be closed. Then the TCP Transport plugin can restart the process of establishing that connection.</p> <p>Range: 1 second or higher.</p> <p>Default: 10 seconds</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
initial_handshake_timeout :	<p>Specifies a timeout for the initial handshake for a connection.</p> <p>Some of the TCP Transport plugin configurations (e.g., when using TLS over TCP) require an initial handshake for each established connection.</p> <p>This property controls the maximum time (in seconds) the initial handshake for a connection can remain in progress. If the handshake has not completed after the specified timeout, the connection will be closed. Then the TCP Transport plugin can restart the process of establishing and handshaking that connection.</p> <p>Range: 1 second or higher.</p> <p>Default: 10 seconds</p>
tls.verify.ca_file	<p>A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information.</p> <p>To enable TLS, ca_file or ca_path is required; both may be specified (at least one is required).</p>
tls.verify.ca_path	<p>A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_load_verify_locations for more information.</p> <p>To enable TLS, ca_file or ca_path is required; both may be specified (at least one is required).</p>
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.crl_file	<p>Name of the file containing the Certificate Revocation List.</p> <p>File should be in PEM format.</p>
tls.identity.certificate_chain	<p>String containing an identifying certificate (in PEM format) or certificate chain (appending intermediate CA certs in order).</p> <p>An identifying certificate is required for secure communication. The string must be sorted starting with the certificate to the highest level (root CA). If this is specified, certificate_chain_file must be empty.</p>

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t

Property Name (prefix with 'dds.transport.TCPv4.tcp1.' ¹)	Description
tls.identity.certificate_chain_file	<p>File containing identifying certificate (in PEM format) or certificate chain (appending intermediate CA certs in order).</p> <p>An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If this is specified, certificate_chain must be empty.</p> <p>Optionally, a private key may be appended to this file. If no private key option is specified, this file will be used to load a private key.</p>
tls.identity.private_key_password	A string that specifies the password for private key.
tls.identity.private_key	<p>String containing private key (in PEM format).</p> <p>At most one of private_key and private_key_file may be specified. If no private key is specified (all values are NULL), the private key will be read from the certificate chain file.</p>
tls.identity.private_key_file	<p>File containing private key (in PEM format).</p> <p>At most one of private_key and private_key_file may be specified. If no private key is specified (all values are NULL), the private key will be read from the certificate chain file.</p>
tls.identity.rsa_private_key	<p>String containing additional RSA private key (in PEM format).</p> <p>For use if both an RSA and non-RSA key are required for the selected cipher. At most one of rsa_private_key and rsa_private_key_file may be specified.</p> <p>At most one of rsa_private_key and rsa_private_key_file may be specified.</p>
tls.identity.rsa_private_key_file	<p>File containing additional RSA private key (in PEM format).</p> <p>For use if both an RSA and non-RSA key are required for the selected cipher. At most one of rsa_private_key and rsa_private_key_file may be specified.</p> <p>At most one of rsa_private_key and rsa_private_key_file may be specified.</p>
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for <code>SSL_set_cipher_list</code> for more information on the format of this string.

¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Table 35.1 Properties for `NDDS_Transport_TCPv4_Property_t`

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')	Description
tls.cipher.dh_param_files	List of available Diffie-Hellman (DH) key files. For example: "foo.h:2048,bar.h:1024" means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 2048, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 1024
tls.cipher.engine_id	ID of OpenSSL cipher engine to request.

35.1.6.1 Connection Liveliness

The **connection_liveliness** property configures the connection liveliness feature. When enabled, the TCP Transport plugin will periodically exchange some additional control traffic (liveliness requests/responses) over one of the connections between the TCP Client and Server. This traffic allows determining if a that connection is not alive anymore, and thus proceed to its close. This avoids depending on the OS notification about the status of the connection, potentially decreasing the time to reestablish lost connections.

The following parameters can be configured:

- **connection_liveliness.enable**: Enables or disables the feature.
- **connection_liveliness.lease_duration**: In seconds, the timeout by which the connection liveliness must be asserted or the connection will be considered not alive. It is also used also as the period between connection liveliness checks. Therefore, the maximum time before a connection is marked as not alive is $2 * \text{connection_liveliness.lease_duration}$.
- **connection_liveliness.assertions_per_lease_duration**: The number of liveliness requests send per each lease duration. Increasing this value will increase the overhead send into the network, but it will also make the connection liveliness mechanism more robust.

This feature relies on the creation on an additional thread in the TCP Transport Plugin (the event thread). For more information about how to configure this thread, see the **event_thread** in [Table 35.1 Properties for NDDS_Transport_TCPv4_Property_t](#).

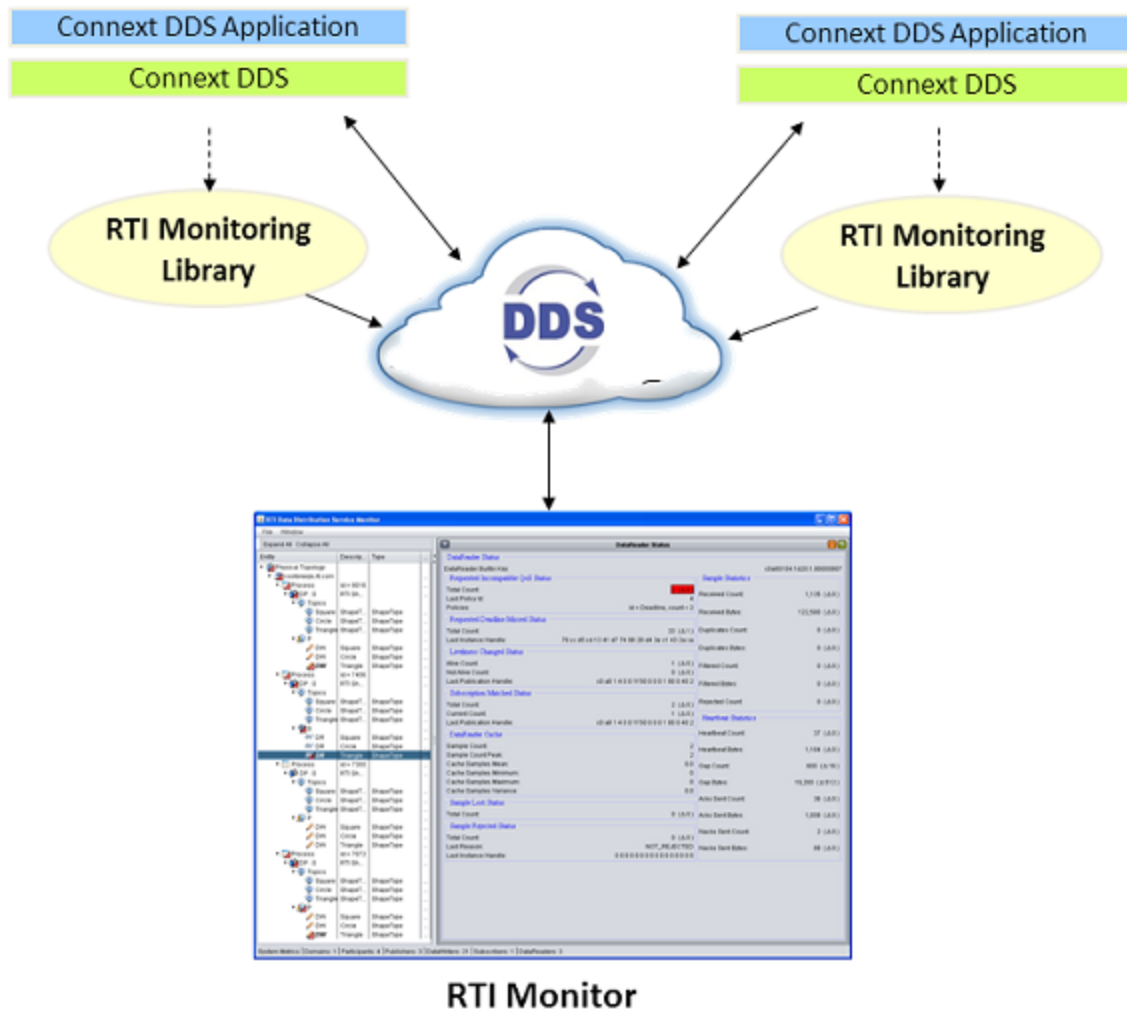
¹Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with 'dds.transport.'

Enabling this feature breaks backwards compatibility with TCP Transport plugins that do not include this feature.

Part 9: RTI Monitoring Library

RTI Monitoring Library is a plug-in that enables *RTI Connex DDS™* applications to provide monitoring data. The monitoring data can be visualized with *RTI Monitor*, a separate GUI application that can run on the same host as *Monitoring Library* or on a different host.

Connex DDS notifies *Monitoring Library* every time an entity is created/deleted or a QoS is changed. *Monitoring Library* periodically queries the status of all Connex DDS entities. You can enable/disable monitoring by setting values in the *DomainParticipant's* *PropertyQoSPolicy* (programmatically or through an XML QoS profile).



This part of the *User's Manual* includes:

- Using Monitoring Library in Your Application (Section Chapter 36 on page 1024)
- Configuring Monitoring Library (Section Chapter 37 on page 1034)

Chapter 36 Using Monitoring Library in Your Application

36.1 Enabling Monitoring

There are two ways to enable monitoring in your application:

- [Method 1—Change the Participant QoS to Automatically Load the Dynamic Monitoring Library \(Section 36.1.1 on the next page\)](#)
- [Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library \(Section 36.1.2 on the next page\)](#)

Notes:

- The libraries that you will need for Monitoring are listed in the [RTI Connex DDS Core Libraries Platform Notes](#).
- If your original application has made modifications to either the ParticipantQoS **resource_limits.type_code_max_serialized_length** or any of the transport's default settings to enable large type code or large data, refer to [What Monitoring Topics are Published? \(Section 36.3 on page 1031\)](#) for additional QoS modifications that may be needed.
- *Monitoring Library* creates internal *DataWriters* to publish monitoring data by making modifications based on the default *DataWriter* QoS settings. If you have made changes to the default *DataWriter* QoS, especially if you have increased/decreased the initial or maximum DDS sample/instance values, *Monitoring Library* may have trouble creating *DataWriters* to publish monitoring data, or it may limit the number of statistics that you can publish through the internal monitoring writers. If this is true for your case, you may want to specify the **qos_library** and **qos_profile** that will be used to create these internal writers for publishing monitoring data, to avoid being impacted by default *DataWriter* QoS settings. See [Configuring Monitoring Library \(Section Chapter 37 on page 1034\)](#) for details.

36.1.1 Method 1—Change the Participant QoS to Automatically Load the Dynamic Monitoring Library

If all of the following are true, you can enable monitoring simply by changing your participant QoS (otherwise, use [Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library](#) (Section 36.1.2 below)):

- Your application is linked to *dynamic* Connex DDS libraries, or you are using Java or .Net, and
- You will run your application on a Linux, Windows, Solaris, AIX or Mac OS platform, and
- You are NOT linking in an additional monitoring library into your application at link time (you let the middleware load the monitoring library for you automatically as needed).

If you change the QoS in an XML file as shown below, you can enable/disable monitoring without recompiling. If you change the QoS in your source code, you may need to recompile every time you enable/disable monitoring.

If you need to change the participant QoS by hand, refer to the definition of **Built-inQoSLib::Generic.Monitoring.Common** in `<NDDSHOME>/resource/xml/BuiltinProfiles.documentationONLY.xml` for the values you should set.

Example XML to enable monitoring:

```
<participant_qos>
  <property>
    <value>
      <element>
        <name>rti.monitor.library</name>
        <value>rtimonitoring</value>
      </element>
      <element>
        <name>rti.monitor.create_function</name>
        <value>RTIDefaultMonitor_create</value>
      </element>
    </value>
  </property>
</participant_qos>
```

36.1.2 Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library

If any of the following are true, you must change the Participant QoS to enable monitoring and explicitly load the correct version of *Monitoring Library* at compile time:

- Your application is linked to the static version of Connex DDS libraries.
- You are NOT running your application on Linux, Windows, Solaris, AIX or Mac OS platforms.
- You want to explicitly link in the monitoring library (static or dynamic) into your application.

There are two ways to do this:

- [Method 2-A: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in Source Code \(Section 36.1.2.1 below\)](#): Applies to most users who cannot use Method 1 and do not mind changing/recompiling source code every time you enable/disable monitoring, or whose system does not support setting environment variables programmatically. Participant QoS must be defined in source code with this approach.
- [Method 2-B: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in an Environment Variable \(Section 36.1.2.2 on page 1029\)](#): Applies to users who cannot use Method 1 *and* want to specify the create function pointer via an environment variable. This approach allows the Participant QoS to be defined in an XML file or in source code.

36.1.2.1 Method 2-A: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in Source Code

1. Modify your Connex DDS application based on the following examples.

Traditional C++ Example:

```
#include "ndds/ndds_cpp.h"
#include "monitor/monitor_common.h"
extern "C" int publisher_main(int domainId, int sample_count)
{
    ...
    DDSDomainParticipant *participant = NULL;
    DDS_DomainParticipantQos participant_qos;
    char valueBuffer[17];
    /* Get default QoS */
    retcode =
    DDSTheParticipantFactory->get_default_participant_qos(
        participant_qos);
    if (retcode != DDS_RETCODE_OK) {
        /*Error*/
    }
    /* This property indicates that the DomainParticipant
       has monitoring turned on. The property name MUST be
       "rti.monitor.library". The value can be anything.*/
    retcode = DDSPropertyQoSPolicyHelper::add_property(
        participant_qos.property,
        "rti.monitor.library", "rtimonitoring", DDS_BOOLEAN_FALSE);
}
```

```

if (retcode != DDS_RETCODE_OK) {
    /*Error*/
}
/* The property name "rti.monitor.create_function"
   indicates the entry point for the monitoring library.
   The value MUST be the value of the function pointer of
   RTIDefaultMonitor_create */
sprintf(valueBuffer, "%p", RTIDefaultMonitor_create);
retcode = DDSPropertyQoSPolicyHelper::add_property(
    participant_qos.property,
    "rti.monitor.create_function_ptr",
    valueBuffer, DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Error */
}
/* Create DomainParticipant with participant_qos */
participant = DDSTheParticipantFactory->create_participant(
    domainId, participant_qos, NULL /* listener */,
    DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    /* Error */
}
...

```

Modern C++ Example:

```

#include "rti/rti.hpp" // include all the modern C++ API
#include "monitor/monitor_common.h" // for RTIDefaultMonitor_create
//...

using rti::core::policy::Property;

// Get the property policy from the default DomainParticipantQoS
auto participant_qos = dds::core::QoSProvider::Default().participant_qos();
auto property_policy = participant_qos.policy<Property>();

// This property turns monitoring on
property_policy.set(Property::Entry("rti.monitor.library",
    "rtimonitoring"));

// This property specifies the entry point (function pointer) for the
// monitoring library.
std::ostringstream monitor_function_to_str;
monitor_function_to_str << RTIDefaultMonitor_create;
property_policy.set(Property::Entry(
    "rti.monitor.create_function_ptr", monitor_function_to_str.str()));

```


C Example:

```
#include "ndds/ndds_c.h"
#include "monitor/monitor_common.h"
...
extern "C" int publisher_main(int domainId, int sample_count)
{
    DDS_DomainParticipantFactory *factory = NULL;
    struct DDS_DomainParticipantQos participantQos =
        DDS_DomainParticipantQos_INITIALIZER;
    char valueBuffer[17];
    DDS_DomainParticipant *participant = NULL;
    factory = DDS_DomainParticipantFactory_get_instance();
    if (factory == NULL) {
        /* error */
    }
    if (DDS_DomainParticipantFactory_get_default_participant_qos(
        factory, &participantQos) != DDS_RETCODE_OK) {
        /* error */
    }
    /* This property indicates that the DomainParticipant has
       monitoring turned on. The property name MUST be
       "rti.monitor.library". The value can be anything.*/
    if (DDS_PropertyQosPolicyHelper_add_property(
        &participantQos.property,
        "rti.monitor.library", "rtimonitoring",
        DDS_BOOLEAN_FALSE) != DDS_RETCODE_OK) {
        /* error */
    }
    /* The property name "rti.monitor.create_function_ptr"
       indicates the entry point for the monitoring library.
       The value MUST be the value of the function pointer
       of RTIDefaultMonitor_create */
    sprintf(valueBuffer, "%p", RTIDefaultMonitor_create);
    if (DDS_PropertyQosPolicyHelper_add_property(
        &participantQos.property,
        "rti.monitor.create_function_ptr", valueBuffer,
        DDS_BOOLEAN_FALSE) != DDS_RETCODE_OK) {
        /* error */
    }
    /* create DomainParticipant with participantQos */
    participant=
        DDS_DomainParticipantFactory_create_participant(
            factory, domainId, &participantQos,
            NULL /* listener */,
            DDS_STATUS_MASK_NONE);
    if (participant == NULL) {
        /* error */
    }
}
```

```
}  
DDS_DomainParticipantQos_finalize(&participantQos);  
...
```

Note:

- In the above code, you may notice that valueBuffer is initialized to 17 characters. This is because a pointer (RTIDefaultMonitor_create) is at most 8 bytes (on a 64-bit system) and it takes two characters to represent a byte in hex. So the total size must be:

```
(2 * 8 characters) + 1 null-termination character = 17 characters.
```

2. Link the *Monitoring Library* for your platform into your application at compile time (the Monitoring libraries are listed in the [RTI Connex DDS Core Libraries Platform Notes](#)).

The kind of monitoring library that you link into your application at compile time must be consistent with the kind of Connex DDS libraries that you are linking into your application (static/dynamic, release/debug version of the libraries).

On Windows systems: If you are linking a static monitoring library, you will also need to link in **Psapi.lib** at compile time.

36.1.2.2 Method 2-B: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in an Environment Variable

This is similar to Method 2-A, but if you specify the function pointer value for **rti.monitor.create_function_ptr** in an environment variable that is set programmatically, you can specify your QoS either in an XML file or in source code. If you specify the QoS in an XML file, you can enable/disable monitoring without recompiling. If you change the QoS in your source code, you may need to recompile every time you enable/disable monitoring.

1. In XML, enable monitoring by setting the **rti.monitor.create_function_ptr** property to an environment variable. In our example, the variable is named RTIMONITORFUNCPTR.

```

<participant_qos>
  <property>
    <value>
      <element>
        <name>rti.monitor.library</name>
        <value>rtimonitoring</value>
      </element>
      <element>
        <name>rti.monitor.create_function_ptr</name>
        <value>$(RTIMONITORFUNCPTR)</value>
      </element>
    </value>
  </property>
</participant_qos>

```

2. In the DDS application that links in the monitoring library, get the function pointer of **RTIDefaultMonitor_create** and write it to the same environment variable you named in Step 1 and create a *DomainParticipant* by using the XML profile specified in Step 1. (Setting of the environment variable must appear in the application *before* it creates the *DomainParticipant* using the profile from Step 1.)

Here is an example in C:

```

#include <stdio.h>
#include <stdlib.h>
#include "monitor/monitor_common.h"
...
char putenvBuffer[34];
int putenvReturn;
putenvBuffer[0] = '\0';
sprintf(putenvBuffer, "RTIMONITORFUNCPTR=%p",
        RTIDefaultMonitor_create);
putenvReturn = putenv(putenvBuffer);
if (putenvReturn) {
    printf(
        "Error: couldn't set env variable for RTIMONITORFUNCPTR. "
        "error code: %d\n", putenvReturn );
}
...
/* create DomainParticipant using XML profile from Step 1 */
...

```

Note: In the above code, you may notice that **putenvBuffer** is initialized to 34 characters. This is because a pointer (**RTIDefaultMonitor_create**) is at most 8 bytes (on a 64-bit system) and it takes 2 characters to represent a byte in hex. So the total size must be: $\text{strlen}(\text{RTIMONITORFUNCPTR}) + (2 * 8 \text{ characters}) + 1 \text{ null-termination character} = 17 + 16 + 1 = 34 \text{ characters}$

3. Link the *Monitoring Library* for your platform into your application at compile time (the Monitoring libraries are listed in the [RTI Connex DDS Core Libraries Platform Notes](#)).

The kind of monitoring library that you link into your application at compile time must be consistent with the kind of Connex DDS libraries that you are linking into your application (static/dynamic, release/debug version of the libraries).

On Windows systems: If you are linking a static monitoring library, you will also need to link in **Psapi.lib** at compile time.

36.2 How does Monitoring Library Work?

Monitoring Library works by creating DDS Topics that publish information about the other DDS entities contained in the same operating system process. The Topics can be created inside of the first *DomainParticipant* that enables the library (the default). Or they may be created in a separate *DomainParticipant* if the **rti.monitor.config.new_participant_domain_id** property is used. Use cases for this latter configuration include controlling the domain ID on which this information is exchanged (for example to ensure that this data does not interfere with production topics) as well as the ability to specify the QoS that is used for the *DomainParticipant* (through the **rti.monitor.config.qos_library** and **rti.monitor.config.qos_profile** properties). It may be desirable to specify the QoS for *Monitoring Library's DomainParticipant* if the information will be consumed on a different transport or simply to enable the feature but keep it as isolated from the production system as possible.

36.3 What Monitoring Topics are Published?

Two categories of predefined monitoring topics are sent out:

- *Descriptions* are published when an entity is created or deleted, or there are QoS changes (see [Table 36.1 Descriptions \(QoS and Other Static System Information\)](#)).
- *Entity Statistics* are published periodically (see [Table 36.2 Entity Statistics \(Statuses, Aggregated Statuses, CPU and Memory Usage\)](#)).

Table 36.1 Descriptions (QoS and Other Static System Information)

Topic Name	Topic Contents
rti/dds/monitoring/domainParticipantDescription	<i>DomainParticipant</i> QoS and other static information
rti/dds/monitoring/topicDescription	<i>Topic</i> QoS and other static information
rti/dds/monitoring/publisherDescription	<i>Publisher</i> QoS and other static information
rti/dds/monitoring/subscriberDescription	<i>Subscriber</i> QoS and other static information
rti/dds/monitoring/dataReaderDescription	<i>DataReader</i> QoS and other static information
rti/dds/monitoring/dataWriterDescription	<i>DataWriter</i> QoS and other static information

Table 36.2 Entity Statistics (Statuses, Aggregated Statuses, CPU and Memory Usage)

Topic Name	Topic Contents
rti/dds/monitoring/domainParticipantEntityStatistics	Number of entities discovered in the system, CPU and memory usage of the process
rti/dds/monitoring/dataReaderEntityStatistics	<i>DataReader</i> statuses
rti/dds/monitoring/dataWriterEntityStatistics	<i>DataWriter</i> statuses
rti/dds/monitoring/topicEntityStatistics	<i>Topic</i> statuses
rti/dds/monitoring/ dataReaderEntityMatchedPublicationStatistics	<i>DataReader</i> statuses calculated on a per discovered matching writer basis
rti/dds/monitoring/ dataWriterEntityMatchedSubscriptionStatistics	<i>DataWriter</i> statuses calculated on a per discovered matching reader basis
rti/dds/monitoring/ dataWriterEntityMatchedSubscriptionWithLocatorStatistics	<i>DataWriter</i> statuses calculated on a per sending destination basis

All monitoring data are sent out using specially created *DataWriters* with the above topics.

You can configure some aspects of *Monitoring Library*'s behavior, such as which monitoring topics to turn on, which user topics to monitor, how often to publish the statistics topics, and whether to publish monitoring data using (a) the participant created in the user's application that has monitoring turned on or (b) a separate participant created just for publishing monitoring data. See [Configuring Monitoring Library \(Section Chapter 37 on page 1034\)](#).

36.4 Enabling Support for Large Type-Code (Optional)

Some monitoring topics have large type-code (larger than the default maximum type code serialized size setting). If you use *Monitor* to display all the monitoring data, it already has all the monitoring types built-in and therefore it uses the default maximum type-code serialized size in the Connex DDS application and there is no problem. However, if you are using any other tools to display monitoring data (such as *RTI Spreadsheet Add-in for Microsoft Excel*, *rtiddsspy*, or writing your own application to subscribe to monitoring data), or if your user data-type has large type-code, you may need to increase the maximum type-object serialized size setting in the *DomainParticipantResourceLimitsQosPolicy*.

36.5 Troubleshooting Monitoring

36.5.1 Buffer Allocation Error

Monitoring Library obtains the default *DataWriter* QoS from the Connex DDS application's *DomainParticipant*. If the application has changed the default QoS Profile, either through application code or in an XML file, *Monitoring Library* will use this new default QoS. In specific scenarios, the new default QoS may cause your Connex DDS application to run out of memory and report error messages similar to these:

```
REDAFastBufferPool_growEmptyPoolEA: !allocate buffer of 1210632000 bytes
[D0012|ENABLE]REDAFastBufferPool_newWithNotification:!create fast buffer pool buffers
[D0012|ENABLE]PRESTypePluginDefaultEndpointData_createWriterPool:!create writer buffer pool
[D0012|ENABLE]WriterHistorySessionManager_new:!create newAllocator
[D0012|ENABLE]WriterHistoryMemoryPlugin_createHistory:!create sessionManager
[D0012|ENABLE]PRESWriterHistoryDriver_new:!create _whHnd
[D0012|ENABLE]PRESPsService_enableLocalEndpointWithCursor:!create WriterHistoryDriver
[D0012|ENABLE]PRESPsService_enableAllLocalEndpointsInGroupWithCursor:!enable endpoint
[D0012|ENABLE]PRESPsService_enableGroupWithCursor:!enableAllLocalEndpointsInGroupWithCursor
[D0012|ENABLE]PRESPsService_enableGroup:!enableGroupWithCursor
[D0012|ENABLE]RTIDefaultMonitorPublisher_enableEntitiesAndStartThreadI:!create enable
publisher
[D0012|ENABLE]RTIDefaultMonitorPublisher_onEventNotify:!create enable entities
```

To resolve this problem, either:

- Configure *Monitoring Library* to use a non-default QoS Profile. For details, see [Configuring Monitoring Library \(Section Chapter 37 on page 1034\)](#).
- Change the default QoS to have a lower value for *DataWriter*'s **initial_samples**; this field is part of the ResourceLimitsQoSPolicy.

Chapter 37 Configuring Monitoring Library

You can control some aspects of *Monitoring Library*'s behavior by setting the `PropertyQoSPolicy` of the *DomainParticipant*, either via an XML QoS profile or in your application's code prior to creating the *DomainParticipant*.

Two example QoS profiles are provided in

`<path to examplesa>/connext_dds/qos/MONITORING_LIBRARY_QOS_PROFILES.xml`:

- `CustomerExampleMonitoringLibrary::CustomerExampleMonitoringProfile`

This is an example of how to enable *Monitoring Library* for your applications. It can be used as a guide to enabling *Monitoring Library* quickly in your applications.

- `RTIMonitoringQoSLibrary::RTIMonitoringQoSProfile`

This profile documents the QoS used by *Monitoring Library*. It can also be used as a starting point if you want to tune QoS for *Monitoring Library* (normally not necessary). Use cases for this include customizing *DomainParticipant* QoS (often the transports) to accommodate preferences or environment. This same profile can also be used to subscribe to the *Monitoring Library* Topics. This is useful in situations where the *Monitoring Library* information can be used directly by system components or it is not possible to use the *RTI Monitor* tool.

See the `qos_library` (Section on page 1036) and `qos_profile` (Section on page 1036) properties in [Table 37.1 Configuration Properties for Monitoring Library](#) for further information on when to use the example profiles in `MONITORING_LIBRARY_QOS_PROFILES.xml`.

[Table 37.1 Configuration Properties for Monitoring Library](#) lists the configuration properties that you can set for *Monitoring Library*. These properties are immutable; they cannot be changed after the *DomainParticipant* is created.

^aSee [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

Table 37.1 Configuration Properties for Monitoring Library

Property Name (all must be prepended with “rti.monitor.config.”)	Property Value
get_process_statistics	<p>This boolean value specifies whether or not <i>Monitoring Library</i> should collect CPU and memory usage statistics for the process in the topic <code>rti/dds/monitoring/domainParticipantDescription</code>.</p> <p>This property is only applicable to Linux and Windows systems—obtaining CPU and memory usage on other architectures is not supported.</p> <p>CPU usage is reported in terms of time spent since the process has been started. It can be longer than the actual running time of the process on a multi-core machine.</p> <p>Default: true if unspecified</p>
new_participant_domain_id	<p>To create a separate participant that will be used to publish monitoring information in the application, set this to the domain ID that you want to use for the newly created participant.</p> <p>This property can be used with the <code>qos_library</code> (Section on the facing page) and <code>qos_profile</code> (Section on the facing page) properties to specify the QoS that will be used to create a new participant.</p> <p>Default: Not set (means you want to reuse the participant in your application that has monitoring turned on to publish statistics information for that participant)</p>
publish_period	<p>Period of time to sample and publish all monitoring topics, in units of seconds.</p> <p>Default: 5 if unspecified</p>
publish_thread_priority	<p>Priority of the thread used to sample and publish monitoring data.</p> <p>This value is architecture dependent.</p> <p>Default if unspecified: same as the default used in Connex DDS for the event thread:</p> <p>Windows systems: -2</p> <p>Linux systems: -999999 (meaning use OS-default priority)</p>
publish_thread_stacksize	<p>Stack size used for the thread that samples and publishes monitoring data. This value is architecture dependent.</p> <p>Default if unspecified: same as the default used in Connex DDS for the event thread:</p> <p>Windows systems: 0 (meaning use the default size for the executable).</p> <p>Linux systems: -1 (meaning use OS’s default value).</p>

Table 37.1 Configuration Properties for Monitoring Library

Property Name (all must be prepended with “rti.monitor.config.”)	Property Value
publish_thread_options	<p>Describes the type of thread.</p> <p>Supported values (may be combined with by OR’ing with ‘ ’ as seen in the default below):</p> <ul style="list-style-type: none"> • FLOATING_POINT: Code executed within the thread may perform floating point operations • STDIO: Code executed within the thread may access standard • I/O REALTIME_PRIORITY: The thread will be scheduled on a real-time basis • PRIORITY_ENFORCE: Strictly enforce this thread's priority <p>Default: FLOATING_POINT STDIO (same as the default used in Connex DDS for the event thread)</p>
qos_library	<p>Specifies the name of the QoS library that you want to use for creating entities in the monitoring library (if you do not want to use default QoS values as set by the monitoring library).</p> <p>The QoS values used for internally created entities can be found in the library RTIMonitoringQoSLibrary in <path to examples>/connex_dds/qos/MONITORING_LIBRARY_QOS_PROFILES.xml.</p> <p>Default: Not set (means you want to use default <i>Monitoring Library</i> QoS values)</p>
qos_profile	<p>Specifies the name of the QoS profile that you want to use for creating entities in the monitoring library (if you do not want to use the default QoS values).</p> <p>The QoS values used for internally created entities can be found in the profile RTIMonitoringPublishingQoSProfile in <path to examples>/connex_dds/qos/MONITORING_LIBRARY_QOS_PROFILES.xml.</p> <p>Default: Not set (means you want to use default <i>Monitoring Library</i> QoS values)</p>
reset_status_change_counts	<p><i>Monitoring Library</i> obtains all statuses of all entities in the Connex DDS application. This boolean value controls whether or not the change counts in those statuses are reset by <i>Monitoring Library</i>.</p> <p>If set to true, the change counts are reset each time <i>Monitoring Library</i> is done accessing them.</p> <p>If set to false, the change counts truly reflect what users will see in their application and are unaffected by the access of the monitoring library.</p> <p>Default: false</p>
skip_monitor_entities	<p>This boolean value controls whether or not the entities created internally by <i>Monitoring Library</i> should be included in the entity counts published by the participant entity statistics topic.</p> <p>If set to true, the internal monitoring entities will not be included in the count. (Thirteen internal writers are created by the monitoring library by default.)</p> <p>Default: true</p>

Table 37.1 Configuration Properties for Monitoring Library

Property Name (all must be prepended with “rti.monitor.config.”)	Property Value
skip_participant_properties	<p>If set to true, <i>DomainParticipant</i> PropertyQosPolicy name and value pairs will not be sent out through the domainParticipantDescriptionTopic. This is necessary if you are linking with <i>Monitoring Library</i> and any of these conditions occur:</p> <ul style="list-style-type: none"> • The PropertyQosPolicy of a <i>DomainParticipant</i> has more than 32 properties. • Any of the properties in PropertyQosPolicy of a <i>DomainParticipant</i> has a name longer than 127 characters or a value longer than 511 characters. <p>Default: false if unspecified</p>
skip_reader_properties	<p>If set to true, <i>DataReader</i> PropertyQosPolicy name and value pairs will not be sent out through the dataReaderDescriptionTopic. This is necessary if you are linking with <i>Monitoring Library</i> and any of these conditions occur:</p> <ul style="list-style-type: none"> • The PropertyQosPolicy of a <i>DataReader</i> has more than 32 properties. • Any of the properties in PropertyQosPolicy of a <i>DataReader</i> has a name longer than 127 characters or a value longer than 511 characters. <p>Default: false if unspecified</p>
skip_writer_properties	<p>If set to true, <i>DataWriter</i> PropertyQosPolicy name and value pairs will not be sent out through the dataWriterDescriptionTopic. This is necessary if you are linking with <i>Monitoring Library</i> and any of these conditions occur:</p> <ul style="list-style-type: none"> • The PropertyQosPolicy of a <i>DataWriter</i> has more than 32 properties. • Any of the properties in PropertyQosPolicy of a <i>DataWriter</i> has a name longer than 127 characters or a value longer than 511 characters. <p>Default: false if unspecified</p>
topics	<p>Filter for monitoring topics, with regular expression matching syntax as specified in the Connex DDS documentation (similar to the POSIX fnmatch syntax). For example, if you only want to send description topics and the entity statistics topics, but NOT the matching statistics topics, you can specify “**Description,*EntityStatistics”.</p> <p>Default: * if unspecified</p>
usertopics	<p>Filter for user topics, with regular expression matching syntax as specified in the Connex DDS documentation (similar to the POSIX fnmatch syntax). For example, if you only want to send monitoring information for reader/writer/topic entities for topics that start with Foo or Bar, you can specify “Foo*,Bar*”.</p> <p>Default: * if unspecified</p>

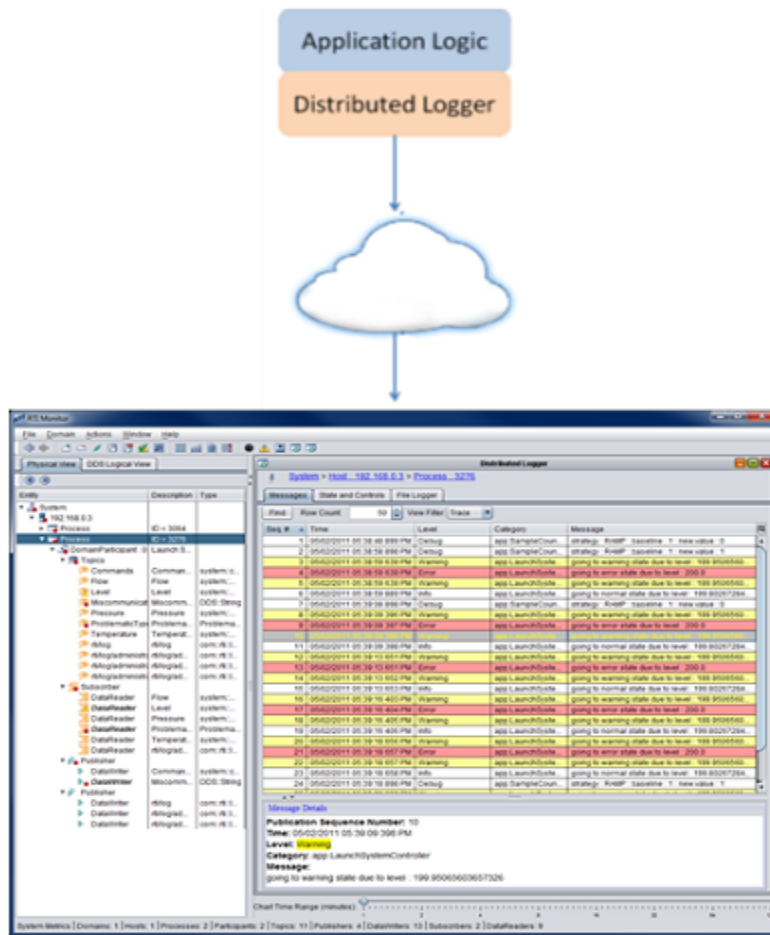
Table 37.1 Configuration Properties for Monitoring Library

Property Name (all must be prepended with “rti.monitor.config.”)	Property Value
verbosity	<p>Sets the verbosity on the monitoring library for debugging purposes (does not affect the topic/data that is sent out).</p> <ul style="list-style-type: none"> • -1: Silent • 0: Exceptions only • 1: Warnings • 2 and up: Higher verbosity level <p>Default: 1 if unspecified</p>
writer_pool_buffer_max_size	<p>Controls the threshold at which dynamic memory allocation is used, expressed as a number of bytes.</p> <p>If the serialized size of the data to be sent is smaller than this size, a pre-allocated writer buffer pool is used to obtain the memory.</p> <p>If the serialized size of the data is larger than this value, the memory is allocated dynamically.</p> <p>This setting can be used to control memory consumption of the monitoring library, at the cost of performance, when the maximum serialized size of the data type is large (which is the case for some description topics' data types) or if you have several participants on the same machine.</p> <p>The default setting is -1, meaning memory is always obtained from the writer buffer pool, whose size is determined by the maximum serialized size.</p>

Part 10: RTI Distributed Logger

RTI® Distributed Logger is a library that enables applications to publish their log messages to Connext DDS. The log message data can be visualized with *RTI Monitor*, a separate GUI application that can run on the same host as your application or on a different host. Since the data is provided in a *Topic*, you can also use *rtiddspy* or even write your own visualization tool.

Distributed Logger can send Connext DDS errors, warnings and other internal messages as a DDS *Topic*. In fact, *Distributed Logger* also provides a remote command topic so that its behavior can be remotely controlled at run time.



This part of the *User's Manual* includes:

- Using Distributed Logger in a Connex DDS Application (Section Chapter 38 on page 1041)
- Enabling Distributed Logger in RTI Services (Section Chapter 39 on page 1049)

Chapter 38 Using Distributed Logger in a Connex DDS Application

There are two ways to use *Distributed Logger*: directly through its API or by attaching it to an existing logging framework as an ‘appender’ or a ‘handler.’ Using the API directly is straightforward, but keep in mind that *Distributed Logger* is not intended to be a full-featured logging library. In particular, it does *not* contain the ability to log messages to standard out/error. Rather, it is primarily intended to be integrated into third-party logging infrastructures.

The libraries that you will need for *Distributed Logger* are listed in the [RTI Connex DDS Core Libraries Platform Notes](#).

Distributed Logger comes with third-party integrations for the open-source project log4j (<http://logging.apache.org/log4j/>) as well as Java’s built-in logging library (`java.util.logging`). Please see [Examples \(Section 38.2 on the next page\)](#) for examples that illustrate these integrations.

Distributed Logger captures and forwards Connex DDS internal information, warning, and error messages using a DDS topic. It monitors these messages using the same mechanism as user log messages.

These *Connex DDS* log messages are sent over DDS automatically as soon as you initialize *Distributed Logger* (by calling `RTI_DL_DistLogger_getInstance()` in C or C++, or `Logger.getLogger(...)` in Java; see the API Reference HTML documentation for details).

38.1 Using the API Directly

Details on using the *Distributed Logger* APIs are provided in the API Reference HTML documentation: `<NDDSHOME1>/doc/api/connex_dds/distributed_logger/<language>`. Start by opening `index.html`.

¹See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

If you plan to use the *Distributed Logger*'s API directly, please be aware of the following notes. To configure the options, create an options object and update its fields. Once your updates are complete, set the options on *Distributed Logger*. It is important that this be done **before** *Distributed Logger* is instantiated. *Distributed Logger* acts as a singleton and there is no way to change the options after it has been created.

When your application is ready to exit, use the 'delete' method. This will delete all Entities and threads associated with *Distributed Logger*.

38.2 Examples

Distributed Logger includes several examples in `<path to examples>/distributed_logger`:

- **c/hello_distributed_logger**

This is a simple example of how to use the API directly and does not publish or subscribe to any Topics except the ones related to *Distributed Logger*.

- **c++/hello_distributed_logger**

This is a simple example of how to use the API directly and does not publish or subscribe any Topics except the ones related to *Distributed Logger*.

- **java/hello_direct_usage**

This is a simple example of how to use the API directly and does not publish or subscribe any Topics except the ones related to *Distributed Logger*.

- **java/hello_file_logger**

This example shows how an application can use the information provided by *Distributed Logger*. As the name suggests, this example subscribes to log messages and writes them to a file. Multiple DDS domains can be subscribed to simultaneously if desired. The example is meant to strike a balance between simplicity and function. Certainly more features could be added to make it a production-ready application but that would obscure the goal of the example.

- **java/hello_java_util_logging**

This is an adaptation of the Hello_idl example which replaces all `System.{out/err}` invocations with Java logging library equivalents. It adds *Distributed Logger* through a configuration file.

¹See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

- **java/hello_log4j_logging**

This is an adaptation of the Hello_idl example which replaces all System.{out/err} invocations with log4j library equivalents. It adds *Distributed Logger* through a configuration file.

Each example has a **READ_ME.txt** file which explains how to build and run it.

38.3 Data Type Resource

You can find the data types used by *Distributed Logger* in `<NDDSHOME1>/resource/idl/distog.idl`.

If you want to generate code and interact with *Distributed Logger* through Topics, you can use this file to do so. You will need to provide extra command-line arguments to *RTI Code Generator* (*rtiddsgen*). (This allows us to accommodate multiple language bindings within the same file. As a consequence, we've used preprocessor definitions to achieve this functionality.) The command-line options which must be added to *rtiddsgen* are as follows:

- For C or C++: **-D LANGUAGE_C**
- For Java: **-D LANGUAGE_JAVA**
- For .Net: **-D LANGUAGE_DOTNET**

If you plan to use the generated code in your application (to subscribe to log messages, for instance) be aware that the type names used might not match the default ones. *Do not* use the generated type names obtained when calling `get_type_name()` or found in `distlogSupport.h`. Use the variables in [Table 38.1 Registration Names for each Distributed Logger Type](#) instead.

Table 38.1 Registration Names for each Distributed Logger Type

Type	Registered Typename	Variable
Log Message	com::rti::dl::LogMessage	C/C++: RTI_DL_LOG_MESSAGE_TYPE_NAME Java: LOG_MESSAGE_TYPE_NAME.VALUE
Administration State	com::rti::dl::admin::State	C/C++: RTI_DL_STATE_TYPE_NAME Java: STATE_TYPE_NAME.VALUE

¹See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

Table 38.1 Registration Names for each Distributed Logger Type

Type	Registered Typename	Variable
Administration Command Request	com::rti::dl::admin::CommandRequest	C/C++: RTI_DL_COMMAND_REQUEST_TYPE_NAME Java: COMMAND_REQUEST_TYPE_NAME.VALUE
Administration Command Response	com::rti::dl::admin::CommandResponse	C/C++: RTI_DL_COMMAND_RESPONSE_TYPE_NAME Java: COMMAND_RESPONSE_TYPE_NAME.VALUE

For instance, to subscribe to log messages in C you will need to do the following:

```
retcode = RTI_DL_LogMessageTypeSupport_register_type(
    participant, RTI_DL_LOG_MESSAGE_TYPE_NAME);
```

38.4 Distributed Logger Topics

Distributed Logger uses four Topics to publish log messages, state, and command responses and one topic to subscribe to command requests. These are detailed in [Table 38.2 Topics Used by Distributed Logger](#).

Table 38.2 Topics Used by Distributed Logger

Topic	Type Name	Quality of Service
rti/distlog	com::rti::dl::LogMessage	Reliable Transient Local
rti/distlog/administration/state	com::rti::dl::admin::State	Reliable Transient Local
rti/distlog/administration/command_request	com::rti::dl::admin::CommandRequest	Reliable
rti/distlog/administration/command_response	com::rti::dl::admin::CommandResponse	Reliable

38.5 Distributed Logger IDL

The IDL describing the types used for Topics created by *Distributed Logger* are in `<NDDSHOME>/resource/idl/distlog.idl`. You can use this IDL to create custom applications that use the data provided by

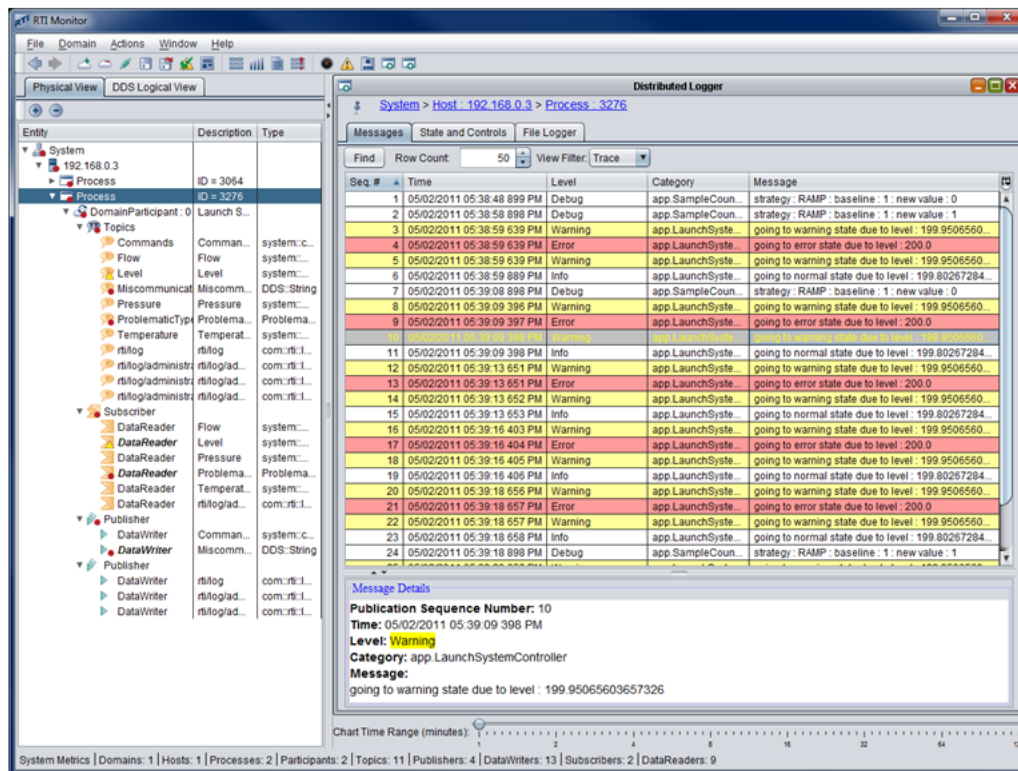
¹See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

Distributed Logger and/or to remotely control any *Distributed Logger* instances that are running in your system. The IDL has been designed to take advantage of the latest type-support features in Connex DDS.

38.6 Viewing Log Messages

One way to see the messages from *Distributed Logger* is to use *RTI Monitor*.

Figure 38.1 Viewing Log Messages with RTI Monitor



Other ways to see the log messages include using *rtiddsspy* or writing your own visualization tool. If you want to write your own application that interacts with *Distributed Logger*, you can find the IDL in `<NDDSHOME>1/resource/idl/distlog.idl`.

38.7 Logging Levels

Log levels in *Distributed Logger* are organized as follows (ordered by importance). This table also shows the mapping between logging levels in the Connex DDS middleware and *Distributed Logger*.

¹See [Paths Mentioned in Documentation \(Section on page xxxviii\)](#)

Connex DDS Logger Log Level	Distributed Logger Log Level
NDDS_CONFIG_LOG_LEVEL_ERROR	RTI_DL_ERROR_LEVEL
NDDS_CONFIG_LOG_LEVEL_WARNING	RTI_DL_WARNING_LEVEL
NDDS_CONFIG_LOG_LEVEL_STATUS_LOCAL	RTI_DL_NOTICE_LEVEL
NDDS_CONFIG_LOG_LEVEL_STATUS_REMOTE	RTI_DL_INFO_LEVEL
NDDS_CONFIG_LOG_LEVEL_DEBUG	RTI_DL_DEBUG_LEVEL

38.8 Distributed Logger Quality of Service Settings

To ensure that *Distributed Logger* works correctly with other RTI tools, some QoS settings are hard-coded and cannot be modified by customized profiles. [Table 38.3 QoS Values Used by Distributed Logger](#) lists the QoS values that are set in *Distributed Logger*. Values in bold are hard-coded; therefore even if they appear in an XML profile, they remain as noted in the table.

Table 38.3 QoS Values Used by Distributed Logger

Entity	Property	Value
Subscriber	Presentation.access_scope	PRES_INSTANCE_PRESENTATION_QOS
	Presentation.coherent_access	false
	Presentation.ordered_access	false
Publisher	Presentation.access_scope	PRES_INSTANCE_PRESENTATION_QOS
	Presentation.coherent_access	false
	Presentation.ordered_access	false
Log Message Topic	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
	Durability.kind	DDS_TRANSIENT_LOCAL_DURABILITY_QOS
Administration State Topic	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
	Durability.kind	DDS_TRANSIENT_LOCAL_DURABILITY_QOS
Administration Command Request Topic	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
Administration Command Response Topic	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS

Table 38.3 QoS Values Used by Distributed Logger

Entity	Property	Value
Log Message DataWriter	Ownership.kind	DDS_SHARED_OWNERSHIP_QOS
	Latency_budget.duration.sec	0
	Latency_budget.duration.nanosec	0
	Liveliness.kind	DDS_AUTOMATIC_LIVELINESS_QOS
	Destination_order.kind	DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
	Durability.kind	DDS_TRANSIENT_LOCAL_DURABILITY_QOS
	History.kind	DDS_KEEP_LAST_HISTORY_QOS
	History.depth	10
Administration State DataWriter	Ownership.kind	DDS_SHARED_OWNERSHIP_QOS
	Latency_budget.duration.sec	0
	Latency_budget.duration.nanosec	0
	Liveliness.kind	DDS_AUTOMATIC_LIVELINESS_QOS
	Destination_order.kind	DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
	Durability.kind	DDS_TRANSIENT_LOCAL_DURABILITY_QOS
	History.kind	DDS_KEEP_LAST_HISTORY_QOS
	History.depth	1

Table 38.3 QoS Values Used by Distributed Logger

Entity	Property	Value
Administration Command Response DataWriter	Ownership.kind	DDS_SHARED_OWNERSHIP_QOS
	Latency_budget.duration.sec	0
	Latency_budget.duration.nanosec	0
	Liveliness.kind	DDS_AUTOMATIC_LIVELINESS_QOS
	Destination_order.kind	DDS_BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
	History.kind	DDS_KEEP_LAST_HISTORY_QOS
	History.depth	10
Administration Command Request DataReader	Ownership.kind	DDS_SHARED_OWNERSHIP_QOS
	Latency_budget.duration.sec	DDS_DURATION_INFINITE_SEC
	Latency_budget.duration.nanosec	DDS_DURATION_INFINITE_NSEC
	Deadline.period.sec	DDS_DURATION_INFINITE_SEC
	Deadline.period.nanosec	DDS_DURATION_INFINITE_NSEC
	Liveliness.kind	DDS_AUTOMATIC_LIVELINESS_QOS
	Destination_order.kind	DDS_BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
	Reliability.kind	DDS_RELIABLE_RELIABILITY_QOS
	History.kind	DDS_KEEP_LAST_HISTORY_QOS
	History.depth	10

Chapter 39 Enabling Distributed Logger in RTI Services

Many RTI components provide integrated support for *Distributed Logger* (check the component's *Release Notes*) and include the *Distributed Logger* library in their distribution. To enable *Distributed Logger* in these components, modify their XML configuration file. In the `<administration>` section, add the `<distributed_logger>` tag as shown in this example:

```
<persistence_service name="default">
  <administration>
    <domain_id>10</domain_id>
    <distributed_logger>
      <enabled>true</enabled>
      <filter_level>DEBUG</filter_level>
      <queue_size>2048</queue_size>
      <thread>
        <priority>
          THREAD_PRIORITY_BELOW_NORMAL
        </priority>
        <stack_size>8192</stack_size>
        <cpu_list>
          <element>0</element>
          <element>1</element>
        </cpu_list>
        <cpu_rotation>
          THREAD_SETTINGS_CPU_NO_ROTATION
        </cpu_rotation>
      </thread>
    </distributed_logger>
  </administration>
  ...
</persistence_service>
```

The tags supported within the `<distributed_logger>` tag are described in [Table 39.1 Distributed Logger Tags](#).

Table 39.1 Distributed Logger Tags

Tags within <distributed_ logger>	Description	Number of Tags Allowed
<enabled>	Controls whether or not <i>Distributed Logger</i> should be enabled at start up. This field is required. Allowed values: TRUE or FALSE	1 (required)
<filter_level>	<p>The filter level for the log messages to be sent. <i>Distributed Logger</i> uses the filter level to discard log messages before they can be sent from the application/service. This is the minimum log level that will be sent out over the network. For example, when using the NOTICE level, any INFO, DEBUG and TRACE-level log messages will be filtered out and not sent from the application/service to Connex DDSS.</p> <p>See important information in Relationship Between Service Verbosity and Filter Level (Section 39.1 on page 1052).</p> <p>Can be set to these values:</p> <ul style="list-style-type: none"> • SILENT • FATAL • SEVERE • ERROR • WARNING • NOTICE • INFO • DEBUG • TRACE (most verbose level, default) 	0 or 1
<queue_size>	The size of an internal message queue used to store log messages before they are written to DDS. Default, 128 log messages.	0 or 1
<thread>	See Table 39.2 Distributed Logger Thread Tags .	0 or 1

Table 39.2 Distributed Logger Thread Tags

Tags within <distributed_ logger>/ <thread>	Description	Number of Tags Allowed
<cpu_list>	<p>Each <element> specifies a processor on which the Distributed Logger thread may run.</p> <pre><cpu_list> <element>value</element> </cpu_list></pre> <p>Only applies to platforms that support controlling CPU core affinity (see the RTI Connex DDS Core Libraries Platform Notes).</p>	0 or 1
<cpu_rotation>	<p>Determines how the CPUs in <cpu_list> will be used by the Distributed Logger thread. The value can be either:</p> <ul style="list-style-type: none"> • THREAD_SETTINGS_CPU_NO_ROTATION The thread can run on any listed processor, as determined by OS scheduling. • THREAD_SETTINGS_CPU_RR_ROTATION The thread will be assigned a CPU from the list in round-robin order. <p>Only applies to platforms that support controlling CPU core affinity (see the RTI Connex DDS Core Libraries Platform Notes).</p>	0 or 1
<mask>	<p>A collection of flags used to configure threads of execution. Not all of these options may be relevant for all operating systems. May include these bits:</p> <ul style="list-style-type: none"> • STDIO • FLOATING_POINT • REALTIME_PRIORITY • PRIORITY_ENFORCE <p>It can also be set to a combination of the above bits by using the “or” symbol (), such as STDIO FLOATING_POINT.</p> <p>Default: MASK_DEFAULT</p>	0 or 1

Table 39.2 Distributed Logger Thread Tags

Tags within <distributed_ logger>/ <thread>	Description	Number of Tags Allowed
<priority>	<p>Thread priority. The value can be specified as an unsigned integer or one of the following strings.</p> <ul style="list-style-type: none"> • THREAD_PRIORITY_DEFAULT • THREAD_PRIORITY_HIGH • THREAD_PRIORITY_ABOVE_NORMAL • THREAD_PRIORITY_NORMAL • THREAD_PRIORITY_BELOW_NORMAL • THREAD_PRIORITY_LOW <p>When using an unsigned integer, the allowed range is platform-dependent.</p>	0 or 1
<stack_size>	Thread stack size, specified as an unsigned integer or set to the string THREAD_STACK_SIZE_DEFAULT. The allowed range is platform-dependent.	0 or 1

39.1 Relationship Between Service Verbosity and Filter Level

A service's verbosity influences the way the log messages reach *Distributed Logger* and their quantity. If a service (such as *RTI Persistence Service*, *RTI Routing Service*, or another service that is integrated with *Distributed Logger*) is configured with a low verbosity, it will not pass a lot of messages to *Distributed Logger*, even if the *Distributed Logger* filter level is set to a very verbose one (such as TRACE). On the contrary, a high verbosity will work better, because it will pass more messages to *Distributed Logger*; in this case the filter level will have more effect.

Note: Since *Distributed Logger* uses a separate thread to send log messages, there is little impact on performance with more verbose filter levels. However, there is some performance penalty in services that use a higher verbosity.