TTM
Total Transaction Management

ActiveMQ 5.2
Reference Guide v1.8

Total Transaction Management, LLC

An Open Source Solutions Company

570 Rancheros Drive, Suite 140

San Marcos, CA 92069

760-591-0273

www.ttmsolutions.com

Table of Contents

1 Introduction

The primary goal of this document is to serve as a reference guide on how to use and configure the Apache ActiveMQ message broker. This is a living document that continues to evolve as the product evolves and we uncover more information regarding the product's use.

ActiveMQ is a highly configurable, feature rich messaging system. This release of the guide captures, what we understand to be, the more important aspects of configuring and using ActiveMQ.

The guide supplements the documentation currently available on the Apache ActiveMQ web site, and we hope the guide will help you successfully deploy ActiveMQ.

The document assumes the reader has a basic understanding of message-oriented middleware, the Java programming language, and XML.

Comments and/or suggestions regarding this reference guide would be greatly appreciated. You can email them to the document's primary author: Joe Fernandez .

1.1 What Is ActiveMQ?

ActiveMQ is an open source, message-oriented middleware (MOM) system that is developed and maintained by the Apache Software Foundation (ASF). The next section briefly describes a MOM system; to learn more about MOM in general, visit the Middleware Resource Center.

ActiveMQ provides the following quality of service (QoS) features, which are expected by world-class enterprise deployments: performance, scalability, availability, reliability, transactional integrity, and security. ActiveMQ's clustering and failover technologies provide unprecedented high availability. ActiveMQ also supports a myriad of different low-level transport protocols such as TCP, SSL, HTTP, HTTPS, and XMPP.

ActiveMQ adheres to a plugin architecture that makes it an extendible messaging framework, as well as messaging system. The extendible nature of its architecture allows you to develop custom modules that are included in various parts of the core engine's processing chain. Examples of such modules are core engine plugins, transport connectors, message dispatch policies, persistence adapters, and network services.

ActiveMQ's messaging engine, or message broker, is written in the Java programming language and fully implements version 1.1 of Sun Microsystem's Java Messaging Service (JMS) specification/standard; therefore, it presents a standards-based MOM system through which Java applications can reliably communicate messages to one another. This reference guide assumes the reader has a basic understanding of the Java programming language and JMS 1.1 specification. This guide also advocates developing portable JMS applications and as such, emphasizes adherence to the JMS and the Java Naming and Directory Service (JNDI). There are instances where the guide makes reference to ActiveMQ's native application programming interfaces, and in those instances the reader is warned that making direct reference to such objects will compromise the portability of their JMS client.

ActiveMQ can be deployed on any operating platform (e.g., Windows, UNIX, and Linux) that provides a compatible Java Virtual Machine (JDK 1.5 or higher).

Even though it is written in Java, with the primary goal of implementing the JMS API, ActiveMQ also supports other programming languages (e.g., C#, Ruby, Python, C/C++). This document focuses on JMS clients and does not cover the non-Java clients. Please refer to the ActiveMQ web site for more information on ActiveMQ's support for non-Java programming languages.

You should visit the FAQ on the ActiveMQ web site; it covers a broad range of ActiveMQ topics.

1.2 What is MOM?

MOM is a specific type of messaging middleware that facilitates communications between loosely coupled distributed applications. MOM is more closely identified with providing *asynchronous* communications, via queues, between the loosely coupled applications. So within the MOM framework, messages are sent to and delivered from a message queue. MOM clients send a message to a queue and the message remains in the queue until another MOM client retrieves the message from that queue. One advantage to this asynchronous messaging model is that the client retrieving the message does not have to be available when the message is sent to the queue and can instead retrieve the message at any time. This is sometimes referred to as deferred communications.

All of the above is opposed to a more tightly-coupled synchronous communications model where the sender and receiver of a message must be available at the same time in order to successfully communicate with one another. One example of a tightly coupled distributed system requiring synchronous communications is Remote Method Invocation (RMI). With RMI, the sender requires the receiver to be available when it sends the message; if not, the corresponding remote method invocation fails.

Even though MOM is more closely identified with asynchronous communications, most MOM implementations, including ActiveMQ, can also accommodate the more tightly-coupled synchronous communications paradigm. This is typically performed via the Request-Reply messaging pattern.

1.3 Why Use ActiveMQ

ActiveMQ is the most popular open source software (OSS) MOM today, and is rapidly becoming the de-facto MOM for OSS-based Service Oriented Architecture (SOA) deployments.

The following are advantages of employing OSS and ActiveMQ:

1) OSS advantages:
    a. OSS is more cost effective than proprietary products; there are no royalty license fees.
    b. OSS is now accepted due to the success of Linux and the numerous projects (including ActiveMQ) that comprise the Apache Software Foundation (ASF).
    c. OSS provides customers freedom from being "locked in" to proprietary providers of expensive software products, licenses, and support.
    d. OSS provides more flexibility for support and current status through third party professional services providers and public user forums.
2) AMQ advantages:
    a. ActiveMQ is OSS
    b. ActiveMQ can be integrated with other JMS providers (e.g., IBM's MQ-Series, Progress' SonicMQ, and BEA's MessageQ), and supports clients written in C/C++, C#, Ruby, Perl, and PHP. This provides a path for enterprises to embrace open source ESB for future implementations, regardless of past technology commitments.
    c. ActiveMQ offers proven scalability, availability, and performance that will grow with the customer's requirements.
    d. ActiveMQ is standards based. It supports the open standard JMS 1.1 Application Programming Interface (API), with the ability to communicate between distributed applications using messaging and queuing interfaces.
    e. The ActiveMQ message broker is written in the Java language, and is thus very portable.
    f. ActiveMQ can facilitate a wide range of general purpose solutions via multiple messaging pattern paradigms and numerous attributes.
    g. ActiveMQ makes it easier to leverage grid and virtual server farms with built in load-balancing and optional persistence and performance monitoring through Java Management Extensions (JMX).
    h. ActiveMQ is currently available and is successfully employed in customer production environments.
    i. ActiveMQ is continually supported and updated by the ASF and the user community.

3) AMQ features:
    a. ActiveMQ is a fast and feature-rich open source JMS message broker primarily targeted for loosely coupled, distributed application environments. It provides

persistence and once-and-only-once assurance of message delivery, and can be highly scalable through clustering, peer-to-peer and federated network.

b. ActiveMQ fully supports JMS 1.1 with reliable support for non-persistent, persistent, transactional and XA messaging.
c. ActiveMQ supports both point-to-point and publish-and-subscribe messaging models.
d. ActiveMQ provides high availability and high performance clustering and general purpose asynchronous messaging.
e. ActiveMQ can be used from .NET, C/C++ or from scripting languages like Perl, Python, PHP and Ruby via various "Cross Language Clients."
f. ActiveMQ supports secure Internet communications using industry standard Secure Sockets Layer (SSL), and supports IPV6.
g. AJAX (Asynchronous JavaScript and XML) support in ActiveMQ builds on the same basis as the REST (Representational State Transfer) connector for ActiveMQ, allowing web capable applications to send or receive messages over JMS.
h. ActiveMQ can use JDBC for persistence, and when combined with journaling, can provide high performance persistence. Optimal high-performance persistence is provided through the Kaha storage solution, which is being added in Apache ActiveMQ v5.0.
i. ActiveMQ supports a variety of transport protocols such as TCP, SSL, UDP and multicast.
j. ActiveMQ can be embedded into Spring framework applications and configured using Spring's XML configuration mechanism

1.4 When and Where to Use ActiveMQ

1.4.1 Where

ActiveMQ is becoming the preferred OSS MOM for any modern enterprise application architecture, and can be used to implement a SOA framework co-existing and supporting historical client-server, publish and subscribe, and XML-oriented applications.

ActiveMQ can be integrated with a number of platforms/frameworks including: Geronimo, Spring and Apache Tomcat.

1.4.2 When

ActiveMQ provides a clean application agnostic interface when you need applications to communicate by writing and retrieving application-specific data (messages) to/from queues, without having a private, dedicated, logical connection to link them.

ActiveMQ is the preferred Open Source MOM when you need:

- Availability: Transparent load balancing, failover, and recovery
- Interoperability: With many various message stores including JDBC

- Manageability: Can be administered with JMX
- Performance: Approaching or competing with proprietary solutions
- Reliability: Guaranteed once-and-only-once message delivery
- Scalability: Clustering and load balancing features
- Security: SSL, HTTPS, and authentication and authorization.

1.5 Downloading and Installing ActiveMQ

To download and install ActiveMQ on your operating platform, follow the instructions described in the "ActiveMQ Getting Started Guide", which can be found at http://activemq.apache.org/getting-started.html

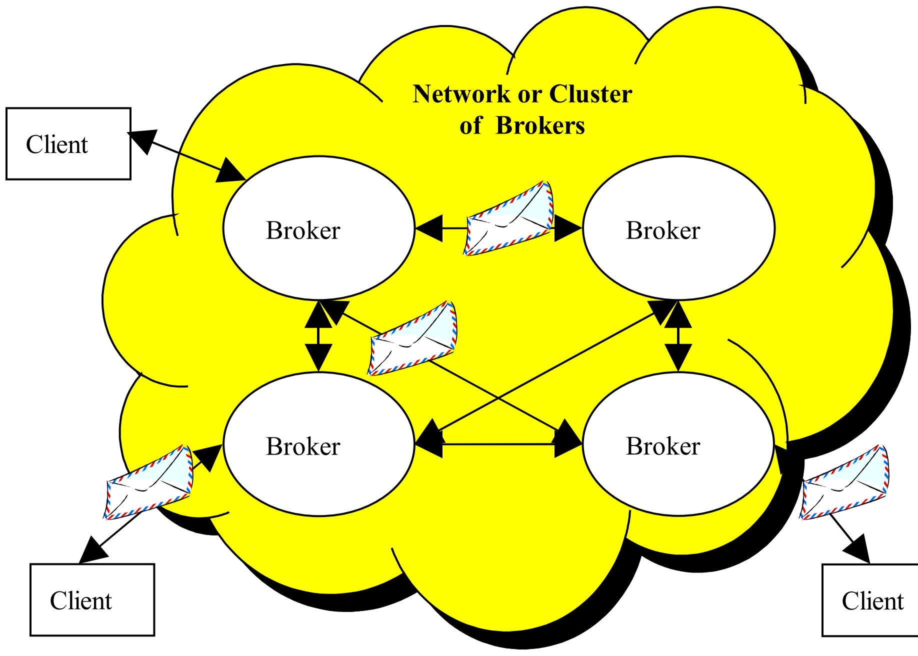
If you have downloaded multiple distributions of ActiveMQ, set the ACTIVEMQ_HOME environment variable to point to the distribution you will be using. For example,

```
> export ACTIVEMQ_HOME=$HOME/apache-activemq-5.0
```

Ensure that, at a minimum, the following jar file is in your client application's CLASSPATH: activemq-all-<*version*>.jar. See 17.9 for a list and description of other jar files that your particular deployment may require.

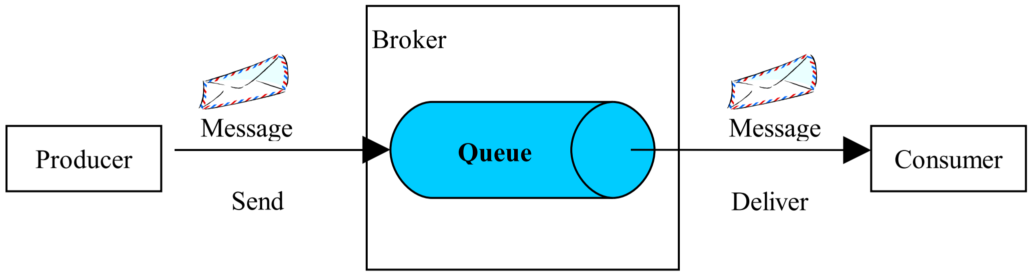
2 ActiveMQ Components

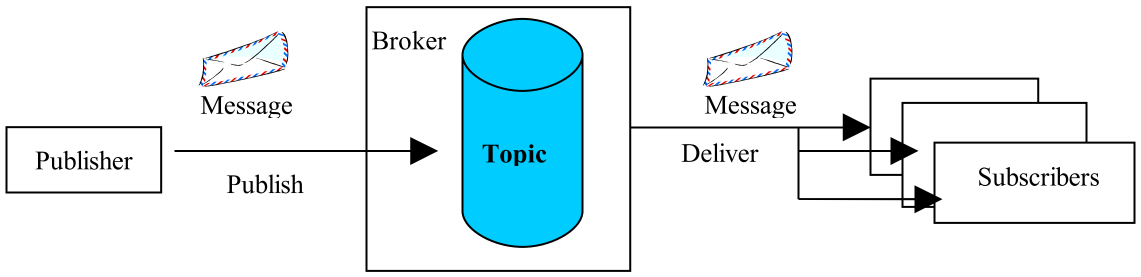
Some of the more important components of the ActiveMQ messaging framework are the client (application), message, destination, and message broker. The client, which is an application component that uses the services provided by the message broker, can further be categorized as either a message *producer* and/or *consumer.* A producer creates a message, which it then gives to the message broker for routing and delivery to a particular destination. Consumers retrieve messages from the destinations to which they have been routed. A destination can therefore be viewed as a logical channel through which clients communicate with one another. It is the responsibility of the ActiveMQ message broker or network of brokers (NoB) to not only route the message to the correct destination, but to also ensure adequate quality of services such as reliability, persistence, security, and high availability. An ActiveMQ NoB can take on different network topologies such as hub-n-spoke, ring, peer-to-peer, etc.



Many times an analogy is drawn between a MOM system and a postal service. That is, the client is a postal customer, the postal system is a broker or NoB, the message is a letter, and the destination is a post office box.

A destination, which can be either a *queue* or a *topic,* is maintained by the message broker (a broker can maintain many destinations). Queues are used by producers to send a message to a consumer, while topics are used by a producer to send a message to one or more consumers. A queue is often-times referred to as a point-to-point messaging channel, while a topic is referred to as a publish-subscribe messaging channel. Thus producers that send messages to topics are more commonly referred to as *publishers* and consumers that retrieve messages from topics are referred to as *subscribers*. So to recap, a message is read from a queue by only one consumer, while one or more consumers (subscribers) can read a message from a topic.

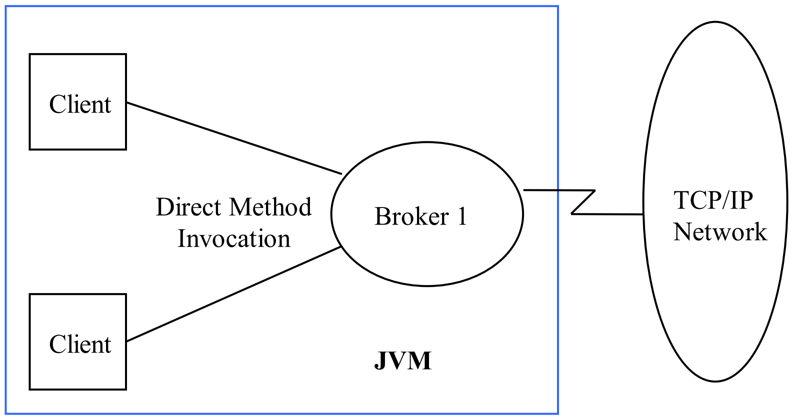




As previously mentioned, the message broker is the ActiveMQ component that accepts messages from producers and delivers those messages to their corresponding target destination (queue or topic). The broker or network of brokers is also responsible for delivering or dispatching messages from the destination to one or more consumers. In ActiveMQ vernacular, the term “dispatching” is more commonly used to describe the delivery of messages from the destination to the consumer. In the process of routing and dispatching messages, the broker provides quality of service features such as guaranteed delivery, high availability, transactional integrity, security, and reliability.

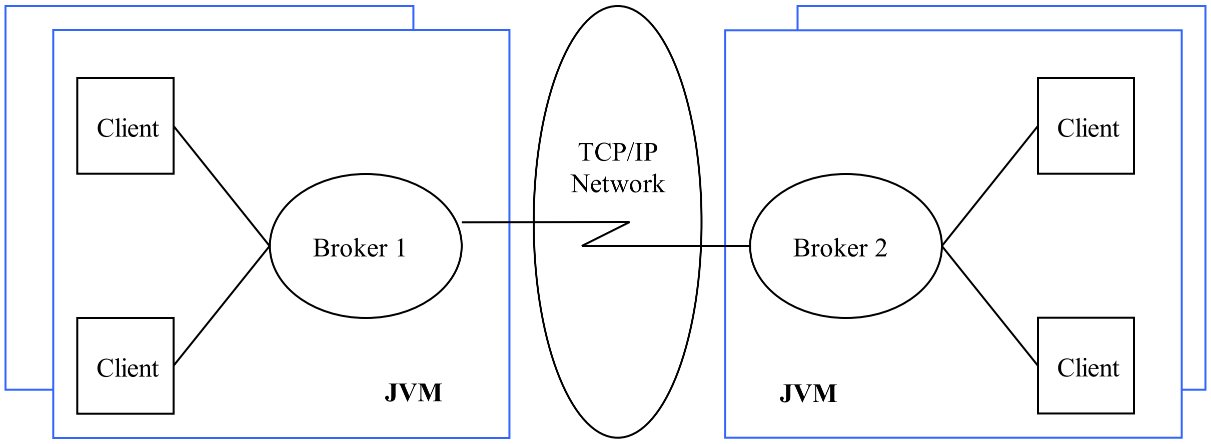
There are two basic types of ActiveMQ brokers: embedded and standalone.

An embedded broker executes within the same JVM process as the Java clients that are using its services. There may be one or more clients residing within a single JVM, each executing within its own thread(s) of execution; all clients access the same embedded broker. The clients communicate with their embedded broker via direct method invocation (DMI) as opposed to serializing command objects across a TCP/IP-based transport connector. One advantage of using an embedded broker is that if the network fails, its embedded clients can still use the services of the broker. For example, a producing client can still send messages to the broker and any messages that need to be forwarded on to another broker will be held and/or persisted by the broker until the network is once-again made available. Another advantage is increased performance, because the broker’s embedded clients communicate with the broker via DMI instead of across a TCP/IP connection.



Embedded Broker

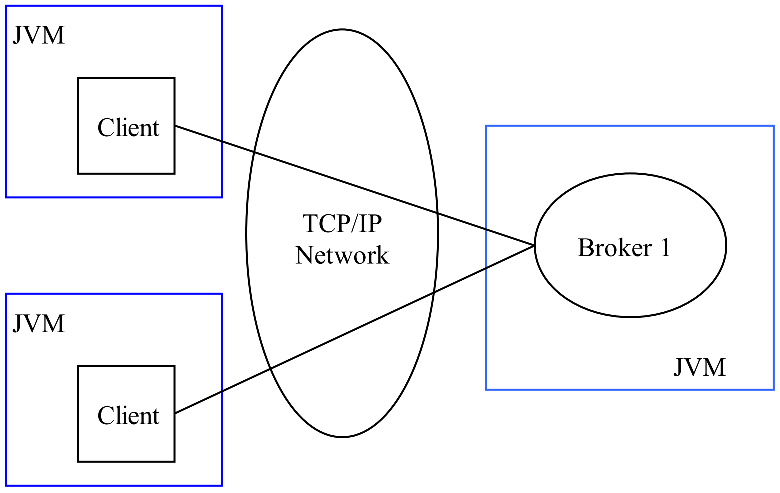
Embedded brokers can connect to other embedded brokers to form what is referred to as a “peer” network. Peer networks provide better performance because there is only one network hop involved when sending a message from a producer/publisher to a consumer/subscriber.



Peer Network

A client starts an embedded broker via the vm or peer transport connectors (see sections 3.2.1 and 3.2.2).

Embedded brokers can also listen for and initiate connections to standalone or non-embedded brokers. Unlike an embedded broker, a *standalone* broker is one that does not have its clients co-residing in its JVM and communicates with its clients through network-based transport connectors, which are covered in the next section.



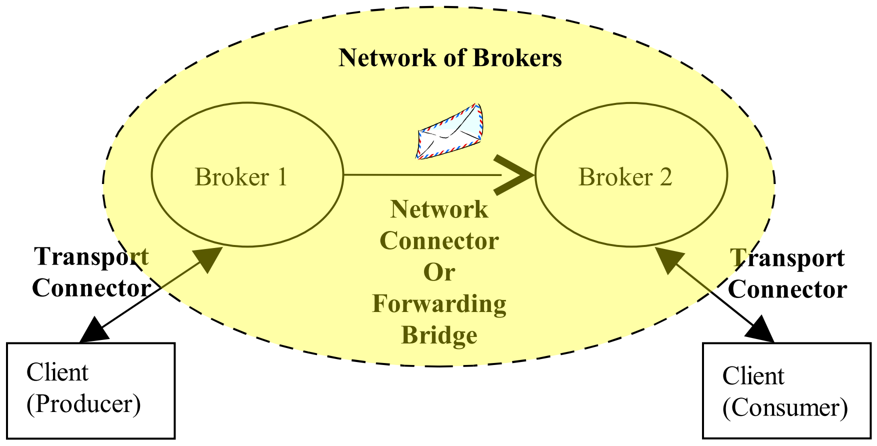
Standalone Broker

3 Connectors

Within the ActiveMQ nomenclature, the terms '*transport connector*' and '*network connector*' are significant. These connectors represent network communication channels through which the clients communicate with their respective *standalone* brokers and brokers communicate with one another. The underlying wire format protocols used through these communication channels are called, "OpenWire" and "STOMP" (see section 7); the default protocol is OpenWire. So when a Java client invokes a JMS operation (e.g., message send), the ActiveMQ client libraries will wrap that operation into an OpenWire command object and then sends or serializes the command object, via the underlying 'transport connector', to the broker.

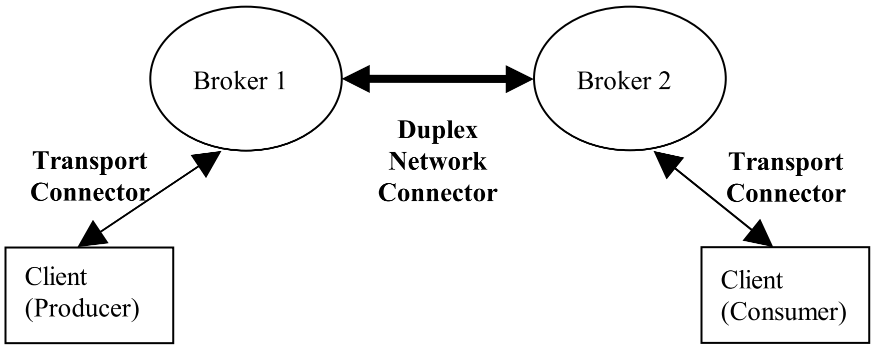
A 'transport connector' is used by a client to establish a bidirectional communication channel with a broker. It is also used by a broker to listen for and accept network connection requests from clients and other brokers.

A 'network connector' is used by a broker to establish a communications channel to another broker. Transport and network connectors are specified through the client and broker's external configuration files.



When a broker (call it broker 1) establishes or initiates a network connection with another broker (broker 2), the resulting connection serves as a unidirectional "*forwarding bridge*" that is used by broker 1 to forward messages on to broker 2. In this case, broker 1 is referred to as the producing broker, whilst broker 2 is referred to as the consuming broker. For example, if a broker has producers, but no consumers, it may use one or more forwarding bridges to forward messages on to brokers that have appropriate consumers. If a broker has multiple forwarding bridges, with appropriate consumers at the other ends of the bridges, it will load balance messages across the bridges.

As described above, the *default* behavior of a network connector is to construct a unidirectional forwarding bridge between two message brokers. So if broker 1 initiates a network connection with broker 2, then the resulting connection can only accommodate messages flowing from broker 1 to broker 2. The connection cannot accommodate messages flowing from broker 2 to broker 1. In effect, broker 1 serves as the message producer and broker 2 the message consumer. This default behavior could not, for example, accommodate two clients implementing a request-reply messaging pattern or protocol, because the reply from the consumer could not flow back to the producer. However, starting with version 5.0 of ActiveMQ, you can override this default behavior so that the network connection can accommodate messages flowing in either direction. In other words, brokers at either end of the channel can both produce and consume messages to and from one another. This type of bidirectional network connection is referred to as a 'duplex' connection and is configured through the broker's external XML configuration file. For more information on how to configure a broker's network connections, see section 6.2.3.



CAUTION: Ensure that only one broker establishes a "duplex" connector with another broker. If both brokers try and establish duplex connectors with one another, one of the brokers will throw an InvalidClientIdException as follows:

javax.jms.InvalidClientIDException: Broker: **broker1** - Client: NC_**broker2**_inbound**broker1** already connected from vm://**broker1**#6

The above exception, which is being thrown by broker1, is stating that broker2 tried to establish a demanding forwarding bridge (network connector) with broker1, but broker1 is disallowing it, because it has already established a demand forwarding bridge with broker2. Broker2 will continuously keep trying and will fill the log files with such an exception.

CAUTION: When reading through the documentation on the ActiveMQ web site, you may run across a broker boolean attribute called, "advisorySupport". The documentation states that setting this attribute to false (default is true) will improve performance a little because it disables the support for advisory messages (see section 9); however, setting this option to 'false' precludes the broker from forwarding messages to other brokers.

3.1 Connector URIs

A connector, whether it is a transport or network connector, is specified through a URI (Uniform Resource Identifier) entry in both the client and broker's configuration files.

In general, URI's are written as follows: *<scheme>:<scheme-specific-part>*

For example, the URI below is used by a client to connect, via the TCP protocol, to a standalone broker that resides on the same machine and is listening on port 61616. This same transport connector URI can also be found in the broker's configuration file to specify that the broker listen on port 61616 for incoming transport and network connection requests.

```
tcp://localhost:61616
```

More information on the URI syntax can be found here: http://www.ietf.org/rfc/rfc1738.txt

The client-side transport connector URIs can be specified either directly by the application code (e.g., when creating an AMQ connection or connection factory object) or indirectly either through the Java Naming and Directory Interface (JNDI) or corresponding framework's (e.g., Spring) configuration file. For those JMS clients that are not integrated within a framework, such as a J2EE container or Spring, the JNDI name space can be configured through the *jndi.properties* file; therefore, you can treat the jndi.properties file as the client's configuration file. It is good practice to have the client rely on the JNDI to indirectly acquire JMS administered objects such as connection factories and destinations (queues and topics). Using the JNDI will ensure that the application remains isolated from any provider-specific objects and thus can be more easily ported across different JMS providers. If a JMS client is embedded within another framework (e.g., Tomcat, Geronimo, Spring), then you should rely on that framework's respective JNDI implementation or configuration file and not the jndi.properties file.

More information on the jndi.properties file can be found in section 5, "Client Configuration".

In the broker's XML configuration file, you'll note a *transportConnector* element. This element is used to listen for and accept network connection requests from other brokers, as well as from clients. The sample below is a snippet of a broker's XML configuration file. The sample defines a broker that establishes one transport and two network connectors. This broker will create networkConnectors to the brokers on the machines called linux01 and linux02. It will also setup a transportConnector to listen for network connection requests that arrive from the brokers on linux01 and linux02, or from clients.

```
<broker brokerName="mybroker" useJmx="true"
xmlns="http://activemq.org/config/1.0">
```

```
    <transportConnectors>
     <transportConnector
              name="openwire"
              uri="tcp://localhost:61616"
     />
  </transportConnectors>

  <networkConnectors>
    <networkConnector
            name="linux01 and linux02"
           uri="static://(tcp://linux01:61616,tcp://linux02:61616)"
    />
  </networkConnectors>

</broker>
```

The transportConnector and networkConnector element's 'name' attribute can be assigned any name, but should be unique. In the example above, the transportConnector's name was assigned "openwire" to simply identify the transportConnector as one supporting clients that use ActiveMQ's OpenWire protocol. Brokers typically use OpenWire to communicate with one another while clients can use OpenWire, STOMP, and XMPP to communicate with the broker. For more information on ActiveMQ's wire protocols see section 7.

More information on configuring the broker can be found in section 6, "Broker Configuration".

There are two types of ActiveMQ connector URIs: low-level and composite. The scheme (prefix) portion of a low-level connector URI identifies the underlying network protocol that will be used for communications on the corresponding connection. ActiveMQ supports the TCP, SSL, HTTP, and HTTPS protocols for reliable communications, and Multicast and jmDNS protocols for broker/client discovery. There are also two additional prefixes for low-level connector URIs that do not specify a protocol, but are instead used by a client to launch and connect to an embedded broker. These two are referred to as the 'vm' and 'peer' transports.

A composite connector URI wraps one or more low-level connector URIs and is used to layer additional logic or functionality on top of the low-level connector. The following sections cover low-level and composite connectors.

CAUTION*: ActiveMQ uses the java.net.URI class, which does not allow any white space to be used within the URI. So for example, if you are using the failover: or static: composite connectors URIs, which are described later, do not put any white spaces around the* **','** *characters.*

3.2 Low-Level Connectors

3.2.1 VM

The VM connector is used by Java clients to launch and connect to an embedded broker. The corresponding VM transport connector between the client and broker is not a network-based connection (e.g., socket) and instead relies on direct method invocations, which provides for a high performance embedded messaging system.

The first client (thread) to establish a VM transport connector will start the embedded or intra-VM broker. Any subsequent threads within the same JVM that reference the same VM transport connector URI will attach themselves to the already active embedded broker. Once **all** of the broker's connectors (transport and network) have been closed, the embedded broker will automatically shut down. If all clients (threads) close their respective VM transport connectors, but the broker still has active network connectors to other brokers and/or active transport connectors to external clients, then it will remain active and not automatically shutdown.

VM Connector URI Syntax (required portion in **bold**)

```
vm://brokerName?transportOptions
```

VM Transport Options

Option Name	Default Value	Description
broker.*		All the options with this prefix are used to configure the broker. See table below.
create	true	Create the broker on demand if it does not already exist.
marshal	false	If 'true', it will force all command objects, sent between client and broker, to be marshaled and un-marshaled using the default wire format.
trace	false	Causes all commands that are sent over the transport to be logged
wireFormat	"default"	The name of the wire format to use. The default wire format is OpenWire.
wireFormat.*		All the options with this prefix are used to configure the wire format.

The options in this table are used with the broker.* prefix.

Option	Default	Description
useJmx	true	If true, enables JMX. Default is true.
persistent	true	If true, the broker uses persistent storage.

Option	Default	Description
populateJMSXUserID	false	If true, the broker populates the JMSXUserID message property with the sender's authenticated username.
useShutdownHook	true	If true, the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill.
brokerName	"localhost"	Specifies the broker name. Default is localhost.
deleteAllMessagesOnStartup	false	If true, deletes all the messages in the persistent store as the broker starts up.
enableStatistics	true	If true, enables statistics gathering in the broker.
waitForSlave	false	If true, this marks the broker as being a master broker in a pure master/slave configuration and thus the master will not fully initialize until the server has connected to it. This was introduce in ActiveMQ 5.2

This is an example of the most basic VM transport connector URI.

```
vm://mylocalbroker
```

The above example is used by a client to launch and connect to an embedded broker; the resulting broker will be called, "mylocalbroker". In this particular example, the broker is used only for intra-JVM messaging because it has not been configured to connect to other brokers or listen for connections from clients or other brokers. Therefore, the primary purpose of the broker is to service clients (threads) that are running within its same JVM. An example application for such a configuration is one that implements a message processing chain (pipeline). Each stage of the chain is defined by a queue and a thread dedicated to reading off the queue and performing some operation on the messages it reads off the queue. After performing its operation, the thread can then pass the message to the next queue in the chain. Recall that the broker will be launched only by the first client to connect to the embedded broker. Subsequent clients will simply attach to the running broker.

This example illustrates how to configure the broker with the 'broker.*' prefix.

```
vm://mylocalbroker?marshal=false&broker.persistent=false
```

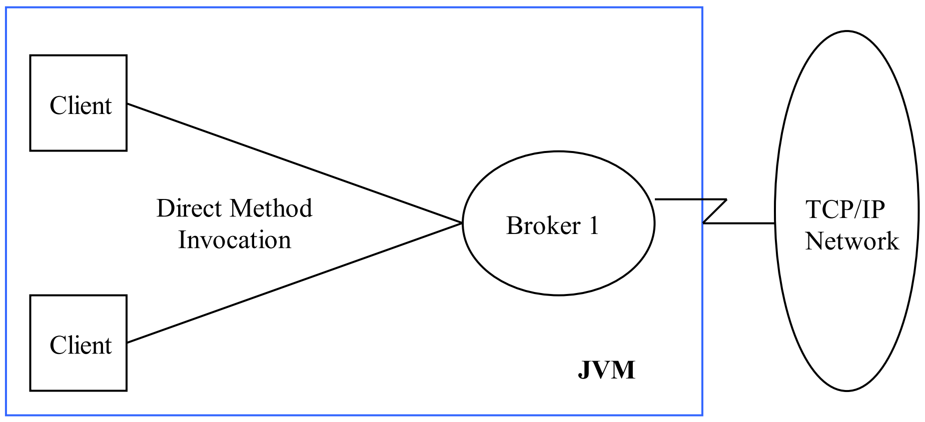
As an alternative, you can configure the broker via the following syntax, which allows you to drop the 'broker.*' prefix.

```
vm:broker:(tcp://host:port)?brokeroptions
```

Here's an example client VM connector URI that starts an embedded broker that is called "embeddedbroker". The broker listens on the local port 6000 for incoming connection requests, and has the persistence feature turned off.

```
vm:broker:(tcp://localhost:6000)?brokerName=embeddedbroker&persistent=false
```

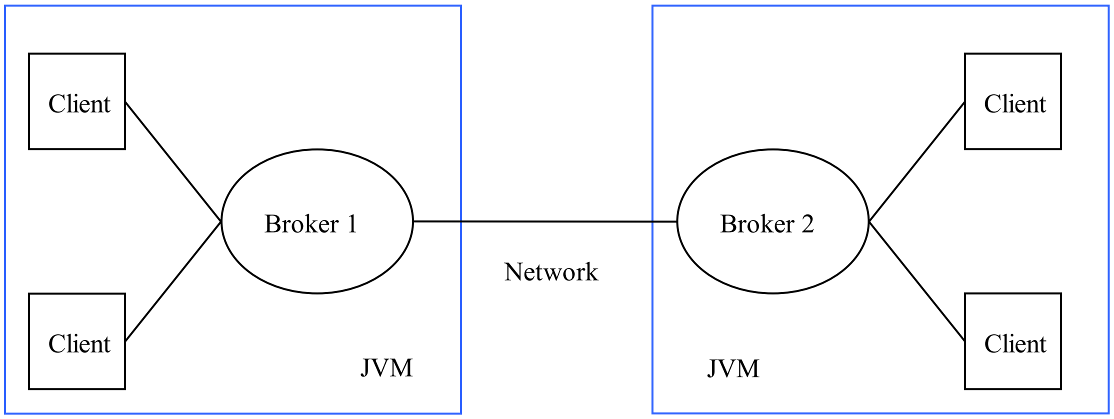
The diagram below illustrates the network topology that results from the above vm connector.



The broker has many properties and using the above syntax to configure the broker can easily result in a very complex URI. The most flexible and powerful method of configuring the embedded broker is through its external XML configuration file. The following URI starts an embedded broker and configures it using an external configuration file called, "activemq.xml", which must be located in the CLASSPATH.

```
vm://localbroker?brokerConfig=xbean:activemq.xml
```

Through the external XML configuration file, you can, for example, instruct the embedded broker to listen on transport connectors and initiate network connections to other embedded and/or standalone brokers. Setting up a network or cluster of brokers that is comprised of nothing but embedded brokers provides for added performance because a message will only travel across one network hop to reach its destination.



A Simple Cluster or Network of Embedded Brokers

More information on configuring the broker can be found in section 6.

When starting an embedded broker, make sure the following jar file is in your CLASSPATH: apache-activemq-<*version*>.jar. If you configure the embedded broker through its external XML configuration file, make sure these jar files are in your client's CLASSPATH: xbean-spring-<*version*>.jar and spring-<*version*>.jar.

CAUTION*: When using an embedded broker, you must use caution that you do not send a message immediately after starting the corresponding Connection object. It is very likely that the embedded broker may not have been fully activated and/or establlished its network connectors by the time you send the message. Our testing has shown that a 1 second delay immediately after the Connection.start() should suffice. A more elegant approach for addressing this situation should be implemented in upcoming releases of ActiveMQ.*

3.2.2 Peer

The peer connector is a superset of the VM connector. The peer connector uses the VM connector to launch and connect to an embedded broker, but it also configures the embedded broker to establish network connectors to other embedded brokers within the LAN subnet that have the same peer group name. So you can consider the peer connector to be a *convenience* connector for setting up a peer-to-peer network of embedded brokers. In other words, a cluster that is comprised solely of embedded brokers.

Peer Connector URI Syntax (required portion in **bold**)

peer://peergroup/brokerName?brokerOptions

The example below will start an embedded broker called "broker1" that will join the "groupa" cluster of embedded brokers.

```
peer://groupa/broker1
```

As a simpler example, you can specify this URI below for all clients in the peer network (each client's broker will be assigned a unique broker id).

```
peer://groupa
```

The peer connector uses multicast to connect to other brokers on the LAN subnet. If any of the following apply, you may encounter issues when setting up a peer network of brokers.

- You have firewall software running on any of the machines involved
- You do not have DNS lookup properly configured for each of the machines involved
- All of the machines involved are not on the same subnet

In the case that you cannot use the peer connector for one of these reasons, you can instead use the VM connector with appropriate broker configuration to set up a cluster of embedded brokers.

3.2.3 TCP

The TCP connector, which is by far the most frequently used, is used by:

- Clients to establish transport connectors to brokers
- Brokers to accept transport connections from clients
- Brokers to establish network connectors to other brokers
- Brokers to accept network connections from other brokers

TCP Connector URI Syntax (required portion in **bold**)

```
tcp://hostname:port?transportOptions
```

Transport Options

Option Name	Default Value	Description

trace	false	Causes all commands that are sent over the transport to be logged
useLocalHost	true	When true, it causes the local machines name to resolve to "localhost".
socketBufferSize	64 * 1024	Sets the socket buffer size in bytes
soTimeout	0	Sets the socket timeout in milliseconds. With this option set to a non-zero timeout, a read() call on the InputStream associated with this TCP Socket will block for only this amount of time. If the timeout expires, a **java.net.SocketTimeoutException** is raised, though the Socket is still valid.
connectionTimeout	30000	A non-zero value specifies the connection timeout in milliseconds. A zero value means wait forever for the connection to be established. Negative values are ignored.
wireFormat	Openwire	The name of the wireFormat to use
wireFormat.*		All the properties with this prefix are used to configure the wireFormat. See section 7, "Wire Protocols".

Examples of using TCP connectors were given in previous sections. The snippet below, which is taken from a broker's XML configuration file, uses a TCP connector within a transportConnector element. In this particular case, the element is used to instruct the broker to listen on the local port 61616 for incoming connection requests from both clients and other brokers. Tracing has been enabled for the connector.

```
<broker brokerName="mybroker" useJmx="true"
     xmlns="http://activemq.org/config/1.0">

    <transportConnectors>
       <transportConnector
                 name="openwire"
                 uri="tcp://localhost:61616?trace=true"
       />
    </transportConnectors>

  …
  </broker>
```

3.2.4 NIO

Same as the TCP connector, but the new I/O (NIO) Java package is used. This may provide better performance than TCP. Our labs have not yet run performance tests between the TCP and NIO connectors.

This is a good article on the Java NIO package:
http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html?page=1

The Java NIO package should not be confused with IBM's AIO4J package:
http://java.sys-con.com/read/46658.htm

NIO Connector URI Syntax (required portion in **bold**)

```
nio://hostname:port?transportOptions
```

3.2.5 SSL

The SSL connector URI is used to establish transport and network connectors using SSL over a TCP socket connection. SSL is short for ***S**ecure **S**ockets **L**ayer;* it is a protocol used for securing communications across a TCP/IP network. It uses a cryptographic system based on two keys to encrypt data that is transmitted across the network.

Connector Syntax

```
ssl://hostname:port?transportOptions
```

A broker XML configuration entry example follows:

```
<transportConnector name="ssl"     uri="ssl://localhost:61410"/>
```

A client jndi.properties configuration entry example follows:

```
connection.local.brokerURL = ssl://linux01:61410
```

The transport options are the same as those for the TCP connector.

3.2.5.1 Setting up the SSL Key and Trust Stores

The following steps describe how to set up the broker and client to use a SSL connector. The 'keytool' is a command line utility that is provided with the Java runtime.

Using the keytool, create a new keystore and self-signed certificate with corresponding public/private keys for the broker.

```
> keytool -genkey -alias amqbroker -keyalg RSA -keystore brokerkeystore
```

Examine the keystore and notice the entry type is PrivateKeyEntry, which means that this entry has a private key associated with it.

```
> keytool -list -v -keystore brokerkeystore
```

Export the broker's certificate so it can be shared with clients:

```
> keytool -export -alias amqbroker -keystore brokerkeystore -rfc -file broker_cert
```

Certificates are typically stored using the printable encoding format defined by the Internet RFC 1421 standard, instead of their binary encoding. This certificate format, also known as "Base 64 encoding", facilitates exporting certificates to other applications by email or through some other mechanism. The `-export` command by default outputs a certificate in binary encoding, but will instead output a certificate in the printable encoding format, if the `-rfc` option is specified.

Using the keytool, create a new keystore and self-signed certificate with corresponding public/private keys for the client.

```
> keytool -genkey -alias amqclient -keyalg RSA -keystore clientkeystore
```

Create a truststore for the client, and import the broker's certificate. This establishes that the client "trusts" the broker:

```
> keytool -import -alias amqbroker -keystore clientkeystore -file broker_cert
```

3.2.5.2 Starting the SSL-enabled Broker

Before starting the broker, you must set up the SSL_OPTS environment variable as follows so that the JVM will know how to use the broker keystore.

```
> export SSL_OPTS = -Djavax.net.ssl.keyStore=/path/to/brokerkeystore
       -Djavax.net.ssl.keyStorePassword=password
```

3.2.5.3 Starting the SSL-enabled Client

When starting the client's JVM, specify the following system properties:

```
javax.net.ssl.keyStore=/path/to/clientkeystore
javax.net.ssl.trustStore=/path/to/clientkeystore
javax.net.ssl.keyStorePassword=password
```

The system properties can be entered as client command arguments, as follows:

```
> java -Djavax.net.ssl.keyStore=/path/to/clientkeystore –D(other properties)
             MyConsumer $@ &
```

Detailed SSL specific information can be obtained by specifying the following argument on the client command line:

```
-Djavax.net.debug=ssl
```

In the above example, the broker will be authenticated by the client. If the broker is to authenticate the client, then you will need to provide a corresponding truststore for the broker.

For more information on the keytool see Sun's JSSE Reference Guide.

This is also a good reference article on the topic. http://docs.codehaus.org/display/JETTY/How+to+configure+SSL

3.2.6 HTTP/HTTPS

These transport connector URIs allow the client and broker to tunnel their communications through the HTTP and HTTPS protocols.

HTTP/HTTPS Connector URI Syntax (required portion in **bold**)

```
http://host:port

https://host:port
```

Client jndi.properties file example:

```
connection.local.brokerURL = http://linux01:61410
```

Broker XML file example:

```
<transportConnector name="http"     uri="http://linux01:61410"/>
```

The following jar files are required in the CLASSPATH for clients:

$ACTIVEMQ_HOME/activemq-<*version*>.jar
$ACTIVEMQ_HOME/lib/optional/commons-httpclient-<*version*>.jar
$ACTIVEMQ_HOME/lib/optional/xstream-<*version*>.jar
$ACTIVEMQ_HOME/lib/optional/xmlpull-<*version*>.jar

Note: See the following JIRA regarding the use of HTTPS.

https://issues.apache.org/activemq/browse/AMQ-1098

There is also this thread, which concerns ActiveMQ's use of Jetty for HTTPS support. Note that a misconfiguration with respect to the keys and/or certificates can lead to an infinite loop.

http://www.nabble.com/SslSocketConnector-loops-forever-during-initialization-to14621825.html#a17535467

3.2.7 Multicast

The multicast low-level connector provides a means for brokers and clients located on the same LAN subnet to find each other and establish connections.

Multicast Connector URI Syntax (required portion in **bold**)

```
multicast://default?transportOptions

or

multicast://ipaddress:port?transportOptions
```

The use of the multicast connector for brokers to discover each other is described here. Client use of the multicast connector URI to find a broker requires the Discovery composite connector, described in section 3.3.3

The multicast connector employs a MulticastSocket to join a multicast group. A multicast group is defined by a class D IP address and port, which can fall in the range 224.0.0.0 to 239.255.255.255, inclusive; the address 224.0.0.0 is reserved and should not be used. When you send a message to a multicast group, via a multicast socket, all subscribing recipients to that IP address and port receive the message. The multicast socket does not have to be a member of the multicast group to send messages to it. When a multicast socket subscribes to a multicast group/ port, it receives datagrams sent by other hosts to the group/port, as do all other members of the group and port.

The example below, which is a snippet from a broker XML configuration file, illustrates how to set up multicast discovery for a broker. The <transportConnector> element's discoveryURI attribute tells the broker to join a multicast group to receive discovery advertisements from other brokers and clients via the broker's well-known default multicast group. The default multicast group's IP address is 239.255.2.3. The <transportConnector> element's uri attribute is telling the broker to listen for subsequent connection requests from other brokers and clients on port 61616. The <networkConnector> element's uri attribute is telling the broker to advertise its discovery packet out through the default multicast group. The discovery advertisement packet essentially tells other brokers that this broker is available and carries information on how to connect to it (i.e., send a network connect request to localhost:61616).

```
<transportConnectors>
    <transportConnector
        name="default"
        uri="tcp://localhost:61616"
        discoveryURI="multicast://default"
    />
</transportConnectors>

<networkConnectors>
   <networkConnector
      name="default"
      uri="multicast://default"
   />
</networkConnectors>
```

This example illustrates how to set up your own unique or non-default multicast group that a network of brokers will subscribe to. Notice how the discoveryURI and uri attributes for the <transportConnector> and <networkConnector> elements have been modified from the default value. Also note that a multicast port number is not required.

```
<transportConnectors>
    <transportConnector
        name="default"
        uri="tcp://localhost:61616"
        discoveryURI="multicast://239.255.2.5"
    />
</transportConnectors>

<networkConnectors>
   <networkConnector
      name="default"
      uri="multicast://239.255.2.5"
   />
</networkConnectors>
```

CAUTION: See AMQ-1489

https://issues.apache.org/activemq/browse/AMQ-1489

3.2.8 Rendezvous

The ActiveMQ rendezvous low-level connector is an alternative to multicast as a means to discover clients or brokers. The rendezvous connector uses jmDNS, the Java implementation of multi-cast DNS. JmDNS is fully compatible with Apple's Bonjour (a.k.a. Rendezvous) zero-configuration protocol. The rendezvous connector can be used by a client to find brokers, and by brokers to find other brokers.

Rendezvous Connector URI Syntax (required portion in **bold**)

```
rendezvous://groupname
```

The use of the rendezvous connector for brokers to discover each other is described here. Client use of the rendezvous connector URI to find a broker requires the Discovery composite connector, described in section 3.3.3

To configure brokers to discover each other using the rendezvous connector, specify matching rendezvous connector URIs in the uri attribute of the <networkConnector> element and the discoveryURI attribute of the <transportConnector> element in the brokers' configuration files. Here is an example.

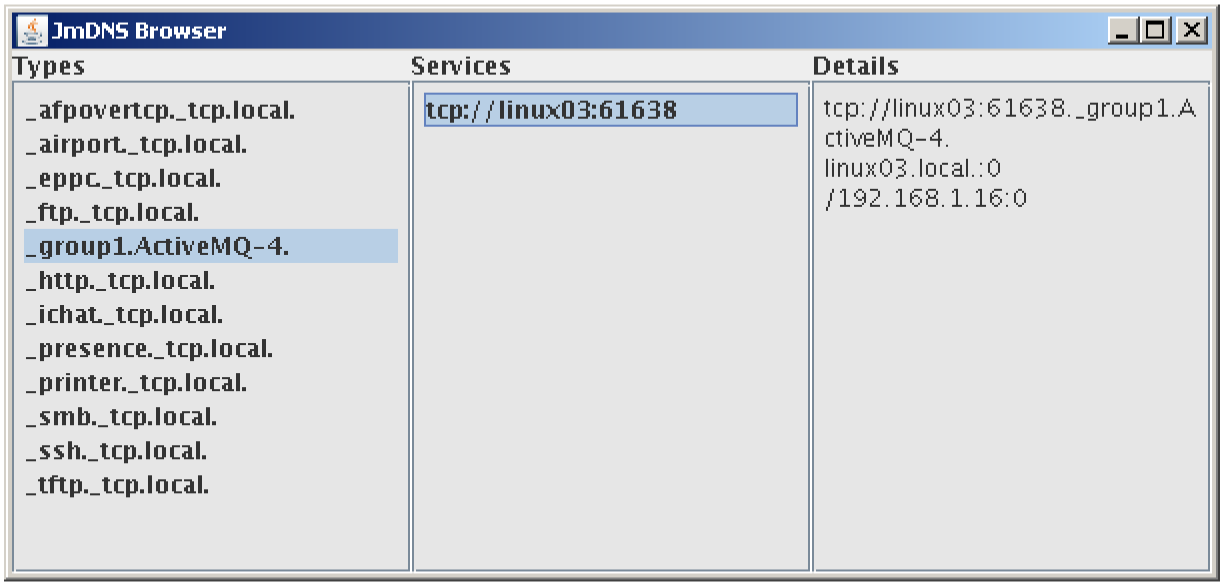
```
<transportConnectors>
    <transportConnector
        name="default"
        uri="tcp://localhost:61616"
        discoveryURI="rendezvous://group1"
    />
</transportConnectors>

<networkConnectors>
   <networkConnector
      name="default"
      uri=" rendezvous://group1"
   />
</networkConnectors>
```

The ActiveMQ distribution includes the jmDNS package in the $ACTIVEMQ_HOME/lib/optional/jmdns-<*version*>.jar file. When running a client, the CLASSPATH must include this file.

The advertisements of the Rendezvous connectors, as well as those of other jmDNS-compatible applications, can be inspected using the JmDNS Browser application contained in the jar file. To run the browser, use

```
> java -jar $ACTIVEMQ_HOME/lib/optional/ jmdns-<version>.jar
```

Shown below is a screen capture of the browser window.



```
_afpovertcp._tcp.local.
_airport._tcp.local.
_eppc._tcp.local.
_ftp._tcp.local.
_group1.ActiveMQ-4.
_http._tcp.local.
_ichat._tcp.local.
_presence._tcp.local.
_printer._tcp.local.
_smb._tcp.local.
_ssh._tcp.local.
_tftp._tcp.local.

tcp://linux03:61638

tcp://linux03:61638._group1.A
ctiveMQ-4.
linux03.local.:0
/192.168.1.16:0
```

The Rendezvous connectors show up as "_*groupname*.ActiveMQ-4." in the Types column of the browser window. Selecting this entry causes the transportConnector's uri value to be displayed under Services.

3.3 Composite Connectors

3.3.1 Static

The composite 'static' connector is used for specifying a list of low-level connector URIs. A broker uses the static connector to establish a network connector for each low-level connector URI that is specified in the list.

Static Connector URI Syntax (required portion in **bold**)

```
static:(uri1,uri2,uri3,...)?transportOptions
```

Note that you cannot have any spaces in between the ',' characters.

In the sample broker's XML configuration file below, the broker's <networkConnector> element includes a static URI with a list of two tcp low-level connector URIs; therefore, the broker will establish two network connectors to the corresponding remote brokers. .

```
<beans>

   <!-- Allows us to use system properties as variables in this
      configuration file -->
  <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
"/>

  <broker brokerName="mybroker" useJmx="true"
     xmlns="http://activemq.org/config/1.0">

    <transportConnectors>
       <transportConnector
              name="openwire"
              uri="tcp://localhost:61616"
       />
    </transportConnectors>

    <networkConnectors>
      <networkConnector
         name="linux01 and linux02"
          uri="static:(tcp://linux01:61616,tcp://linux02:61616)"
       />
     </networkConnectors>

  </broker>
</beans>
```

The static connector URI is used for configuring brokers and cannot be used for configuring clients.

3.3.2 Failover

The composite 'failover' connector, which is used by clients, layers re-connect logic on top of any of the low-level transport connectors, other than the VM and peer connectors. Like the static connector, the failover connector URI allows you to list one or more low level connector URIs. The failover algorithm, associated with this connector, randomly chooses one of the connector URIs in the list and attempts to use it to establish a connection to the corresponding broker. If it does not succeed in establishing the connection or the established connection subsequently fails, a new connection is established to one of the other URIs in the list.

The failover connector is used by clients to connect to a broker that resides in a network or cluster of brokers. The target cluster can be configured as a Master/Slave cluster. In this situation, the client will always connect to a master and will failover to a slave broker if and when the master fails. Master/Slave clusters are described in section 8.2.1.

Failover Connector URI Syntax (required portion in **bold**)

```
failover:(uri1,uri2,uri3,...)?transportOptions
```

In the example URI below, the client will initially connect to the broker on either linux01 or linux02. If the endpoint fails, then the client will be automatically reconnected to the other broker.

```
failover:(tcp://linux01:61616,tcp://linux02:61616)
```

Transport Options

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt (in ms)
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts (in ms)
useExponentialBackOff	true	Should an exponential back off be used between reconnect attempts
backOffMultiplier	2	The exponent used in the exponential back off attempts
maxReconnectAttempts	0	If not 0, then this is the maximum number of reconnect attempts before an error is sent back to the client
randomize	true	use a random algorithm to choose the URI to use for reconnect from the list provided

With the above default settings, the reconnect attempts will occur indefinitely, and the intervening delays will follow this sequence: 10ms, 20ms, 40ms, 80ms, 160ms, …, 30000ms.

The URI below is like the one above, except that it has the initialReconnectDelay transport option set to 100 and randomize turned off. Since randomize is turned off, it will always attempt to connect to the first URI, followed by the second, third, etc...

```
failover:(tcp://localhost:61616,tcp://remotehost:61616)?initialReconnectDelay=100&randomize=false
```

If the failover connector specifies only one URI (see example below) and the corresponding broker fails, then connection retry attempts will be made to that one broker.

```
failover:(tcp://linux01:61616)
```

3.3.3 Discovery

The composite 'discovery' connector URI is used by a client to establish a transport connector to a broker. The discovery connector allows you to do this without having to specify multiple IP addresses or host names. Either the multicast connector (described in section 3.2.7) or the rendezvous connector (described in section 3.2.8) is used as a discovery agent to accomplish this.

The discovery connector is just like the failover connector except that it uses a discovery agent to construct the list of potential broker candidates to connect to. It uses the same transport options as those for the failover connector except that 'randomize' is always set to true.

Discovery Connector URI Syntax (required portion in **bold**)

```
discovery:(discoveryAgentURI)?transportOptions
or
discovery:discoveryAgentURI
```

To configure a broker and client to discover each other using the multicast connector, specify the multicast connector URI for the discoveryURI attribute of the broker's transport connector, as shown in this broker configuration file example.

```
<transportConnectors>
    <transportConnector
        name="default"
        uri="tcp://localhost:61616"
        discoveryURI="multicast://default"
    />
</transportConnectors>
```

Then, a client would be configured to use a discovery connector URI with multicast in the client's jndi.properties file, as shown in this example.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to specify the JNDI names for the connection factories
connectionFactoryNames = local, mdisc

# These are the URLs or URIs to use for the abovementioned connection
# factories
connection.local.brokerURL = tcp://linux02:61616
connection.mdisc.brokerURL = discovery:(multicast://default)?initialReconnectDelay=100
```

Note that the above example assumes the network of target brokers are subscribed to the 'default' multicast group.

To configure a broker and client to discover each other using the rendezvous connector, specify the rendezvous connector URI for the discoveryURI attribute of the broker's transport connector, as shown in this broker configuration file example.

```
<transportConnectors>
    <transportConnector
        name="default"
        uri="tcp://localhost:61616"
        discoveryURI="rendezvous://group1"
    />
</transportConnectors>
```

For the corresponding client configuration, enclose the rendezvous connector URI in a discovery URI, and specify the groupname that was used for the broker's discoveryURI attribute. This example is from a client's jndi.properties file.

```
java.naming.factory.initial =
org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to specify the JNDI names for the connection
factories
connectionFactoryNames = local, rdisc

# These are the URLs or URIs to use for the abovementioned connection
# factories
connection.local.brokerURL = tcp://linux02:61616
connection.rdisc.brokerURL = discovery:(rendezvous://group1)
```

Use caution if your consumer is registering a durable subscription and is also being given a randomized list of brokers to connect with (e.g., via the discovery URI). After starting and stopping multiple times, the consumer may register the same durable subscription with all the brokers on the list. A broker that receives message from a publisher for that particular durable subscription will forward a copy of the message on to other brokers that also have that durable subscription. In effect, this causes the messages to be needlessly copied to all the brokers in the network with that durable subscription.

It is recommended that you always use the maxReconnectAttempts transport option with the discovery connector. If you do not specify this option and there are no brokers available on the network, then the Connection.start() method will hang your client until a broker does become available.

```
connection.mdisc.brokerURL = discovery:(multicast://default)?maxReconnectAttempts=10
```

3.3.4 Fanout

The 'fanout' connector URI is a composite URI that allows the client to simultaneously connect to multiple brokers and *replicate* operations across all of the brokers to which it is connected.

Fanout Connector URI Syntax (required portion in **bold**)

```
fanout:(fanoutURI)?transportOptions
                or
fanout:fanoutURI
```

Valid URI types for fanoutURI include static and multicast. Here's an example of a fanout connector used by a client.

```
fanout:(static:(tcp://linux01:61616,tcp://linux02:61616))?initialReconnectDelay=100
```

With the above example, the client will connect to the brokers running on both the linux01 and linux02 machines. If the client is a producer and it sends a message to a topic called "TOPIC.TEST", that send operation is replicated across both brokers.

The transport options for the fanout URI are the same as those used for the failover connector URI (see section 3.3.2), plus the two options shown in the following table.

Transport Options

Option Name	Default Value	Description
fanOutQueues	false	Should commands be replicated to queues as well as topics?
minAckCount	2	The minimum number of brokers to which connections must be established.

The "fanOutQueues" option is specific to the fanout connector. By default, the fanout does not replicate commands to queues; only topics. Therefore, if you'd like to fanout a message send command to multiple queues on multiple brokers, you'll have to set this option to 'true'.

```
fanout:(static:(tcp://linux01:61616,tcp://linux02:61616))?fanoutQueues=true
```

By default, a client's fanout transport waits for connections to be established to 2 brokers, or the number of static TCP URIs configured (if more than 2). Until this number of connections is established, the client's call to Connection.createSession() does not return. For example, a producer that uses the fanout connector listed below will wait until 2 brokers are running, and connections are established to those two brokers.

```
fanout:(multicast://default)
```

Another example would be a producer using the following fanout connector.

```
fanout:(static:(tcp://localhost:61629,tcp://localhost:61639,tcp://localhost:61649))
```

In this case, three broker connections are needed. However, this required number of connections can be overridden by using the minAckCount transport option. For example, this fanout connector allows the producer to run after connecting to just one broker.

```
fanout:(multicast://default)?minAckCount=1
```

***CAUTION**: It is not recommended that you use the fanout URI for consumers. Also, if a producer fans out across multiple brokers, who happen to be inter-connected, then there is a very high likelihood that a consumer on one of those brokers will get duplicate messages.*

3.4 Monitoring a Transport Connector

This section describes how to set up your client application so that it can monitor the status of its ActiveMQ transport connector (i.e., its connection to the message broker); however, doing so will require that your client make reference to ActiveMQ-specific classes, which will compromise your client's level of JMS portability.

To begin with, your client must, at a minimum, import the following ActiveMQ classes.

```
import org.apache.activemq.transport.TransportListener;
import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.command.Command;
```

Your client must then implement the TransportListener interface by implementing these methods.

```
public void  onCommand(Object command){}
public void  onException(IOException error){}
public void  transportInterupted(){}
public void  transportResumed(){}
```

Your client then registers itself as a transport listener with ActiveMQ by invoking the ActiveMQConnection class's addTransportListener method.

```
((ActiveMQConnection)connection).addTransportListener(this);
```

After registerating itself as a transport listener, the client's four methods listed above will be called asynchronoulsy to inform the client of events at the transport level. The command object passed to the onCommand method implements the Command interface. The Command interface has a series of boolean is*() methods (e.g., isShutdownInfo()) that can be used to determine the event type. More information on this transport event functionality is available via the javadoc for the 'org.apache.activemq.command' package.

4 Wildcards

Sections 5 and 6 describe how to configure a client and broker. Wildcards, which are used within client and broker configuration files, are a convenience mechanism used to refer to multiple destinations within a destination name hierarchy. For example, suppose your client is publishing price messages from a stock exchange feed. You might use some kind of destination naming format such as

PRICE.STOCK.NASDAQ.JAVA to publish Sun's price on NASDAQ and

PRICE.STOCK.NYSE.IBM to publish IBM's price on the New York Stock Exchange

A subscriber could then use exact destinations to subscribe to exactly the prices it requires. Or it could use wildcards to define hierarchical pattern matches to the destinations to subscribe from.

ActiveMQ supports the following wildcards, which are not part of the JMS specification.

- The '.' character is used to separate names in a path.
- The '*' character is used to match any name in a path
- The '>' character is used to recursively match any destination starting from this name

For example using the example above, these subscriptions are possible

Subscription	**Meaning**
PRICE.>	Any price for any product on any exchange
PRICE.STOCK.>	Any price for a stock on any exchange
PRICE.STOCK.NASDAQ.*	Any stock price on NASDAQ
PRICE.STOCK.*.IBM	Any IBM stock price on any exchange

5 Client Configuration

5.1 The jndi.properties File

When writing a JMS client application, you will typically want to rely on the Java Naming and Directory Interface (JNDI) to lookup a particular connection factory and/or destination object. A connection factory is used to render connections to brokers and a destination identifies a queue or topic. In the JMS nomenclature, the connection factory and destination objects are referred to as "*administered objects*". Using the JNDI to lookup or acquire these administered objects will help isolate your application from provider-specific code and thus your application will be more portable across different JMS providers.

The jndi.properties file is used by ActiveMQ client applications to configure and look up these two administered objects.

The following is an example jndi.properties file; notice how connector URIs, which were described in previous sections, are assigned to the different connection factories.

```
########################################################################
#
# This is the jndi.properties file, which is used to
# define/configure one or more administered objects (connection
# factory and destination) for the ActiveMQ JMS client.
#
# This file must be placed in the application's CLASSPATH in order to have
# the JVM load it into the default InitialContext. A JNDI context is a name
# space used to hold name/value pairs.
#
# The 'java.naming.factory.initial' property tells the JVM what JNDI context
# factory (or local JNDI provider) it should use for the application. In this
# case, we're telling it to use AMQ's 'ActiveMQInitialContextFactory' object
# as the context factory.

java.naming.factory.initial =
org.apache.activemq.jndi.ActiveMQInitialContextFactory


#
#                  Connection Factory Names
#
# Use the following property to specify the connection factories that will
# appear in the JNDI context defined by this properties file. Here's a code
# sample of how your application would lookup one of the factories defined
# for the context.
#
#  Context ctx = new InitialContext();
#  ConnectionFactory factory = ctx.lookup("linux01ConnectionFactory");
#
#
```

```
connectionFactoryNames = localConnectionFactory, linux01ConnectionFactory,
linux02ConnectionFactory

#
#                   Connection Properties
#
# This section contains the properties assigned to the connection
# objects that are rendered by their respective factories. These
# properties will not take affect until after you invoke the
# start() method on an instance of a connection.
#
# Note that if you want a property assigned to a specific connection
# factory, you need to qualify it with this prefix
#
# "connection.<factory-name>."
#
# If you only have one factory defined, you don't want to bother with the
# prefixing.
#
#
#
# Normally, in a JNDI environment, java.naming.provider.url is used to
# point the JVM to a remote JNDI provider. However, in the case of
# ActiveMQ, the property is being used to configure the application's
# transport URI connector. That is, the connector it will use to
# connect to the broker. If you do not specify this property, the
# 'tcp://localhost:61616' uri will be used by default. Use this property
# if you only have one connection factory defined. If you have more than
# one factory, use the "connection.<factory-name>.brokerURL" property.

#
# This will connect the application to an 'embedded' broker.
# java.naming.provider.url = vm://localhost

#
# Connect all connections rendered by the linux01ConnectionFactory to the
# remote broker running on linux01
#

connection.linux01ConnectionFactory.brokerURL = tcp://linux01:61616

#
# Connect all connections rendered by the linux02ConnectionFactory to the
# remote broker running on linux02
#

connection.linux02ConnectionFactory.brokerURL = tcp://linux02:61616


#
# Connect all connections rendered by the localConnectionFactory to the
# broker running on the local machine
#

connection.localConnectionFactory.brokerURL = tcp://localhost:61616

#
# Automatically assign this clientID to each connection rendered by the
# linux01ConnectionFactory. Be careful when using this and
# durable subscriptions. Setting this property will disable the
# auto-generation of the unique client ids.
```

```
#
# connection.linux01ConnectionFactory.clientID = joef.123

# Use this property to assign a prefix of your own to the
# auto-generated client ids.
#
#clientIDPrefix =
#
# Assign clientID prefixes to the connections rendered by their respective
# factories.

connection.localConnectionFactory.clientIDPrefix   = Pedro
connection.linux01ConnectionFactory.clientIDPrefix = linux01
connection.linux02ConnectionFactory.clientIDPrefix = linux02

# Automatically assign this user name and password to each connection
# rendered by the localConnectionFactory. Only used if security is
# turned on at the corresponding broker.
connection.localConnectionFactory.userName = user
connection.localConnectionFactory.password = password

#
#
#                Adding Destinations To The Context
#
# Register some queues (destinations) in this JNDI context using the form
# queue.[jndiName] = [physicalName]. The physicalName is the name as
# referenced by the broker. The application would use code similar
# to the following to look up these destinations.
#
# Queue myQ1 = (Queue) ctx.lookup("MyQueue");
# Queue myQ2 = (Queue) ctx.lookup("JunkQ");
#
queue.MyQueue = example.MyQueue
queue.JunkQ   = JunkQ
```

Example jndi.properties File

The above jndi.properties file provides you with a small sample of the properties that can be assigned to a connection factory. The table below lists and describes all the properties that can be assigned to a connection factory.

Property	**Default Value**	**Description**
alwaysSyncSend	false	Set this property to true if you always require messages to be sent synchronously.
brokerURL	null	The connection URL used for connecting to the ActiveMQ broker. For example, tcp://12.345.67.89:6167
clientID	null	Sets the JMS clientID to use for the created connection. If using durable subscriptions, note that setting this

		property will turn off auto-generation of the clientID.
clientIDPrefix	null	Sets the prefix used by auto-generated JMS clientID values, which are used if the JMS client does not explicitly specify one.
closeTimeout	15,000 (ms)	Sets the timeout before a close is considered complete. Normally a close() on a connection waits for confirmation from the broker; this allows that operation to timeout in order to avoid the client hanging if there is no broker
copyMessageOnSend	true	Should a JMS message be copied to a new JMS Message object as part of the send() method? By default, this is enabled to be compliant with the JMS. However, if your producer does not reuse the message object, then it can disable the copying of the message, which improves throughput.
disableTimeStampsByDefault	false	Sets whether or not timestamps on messages should be disabled. If you disable them it adds a small performance boost.
dispatchAsync	true	Enables or disables the default setting for whether consumers have their messages dispatched synchronously or asynchronously by the broker. For more information regarding asynchronous dispatching see section 12.3. Starting with version 5.1, the default value has been switched from ‘false’ to ‘true’.
exclusiveConsumer	false	Enables or disables whether or not queue consumers should be exclusive. For more information see section 12.5.
objectMessageSerializationDefered	false	When an object is set on a message of type ObjectMessage, the JMS spec requires the object to be serialized by that set method. Enabling this flag defers the object serialization. The object may subsequently get serialized if the message has to be sent over a network connection or persisted to secondary storage.
password		Sets the JMS password used for connections created from this factory

producerWindowSize	0	This is used to control the flow of messages from producer to broker.
statsEnabled		
useAsyncSend	false	Forces the use of asynchronous sends, which should improve performance; however, this means that the Session.send() method will return immediately whether the message has been sent or not which could lead to message loss.
useBeanNameAsClientIdPrefix		
useCompression	false	Enables the default "DEFLATE" (gzip) message compression algorithm to compress the message bodies. This property only needs to be set for the producer client. ActiveMQ uses the java.util.zip package's DeflaterOutputStream and InflaterOutputStream classes to compress and de-compress a message, repectively. More information on the DEFLATE algorithm can be found here: http://en.wikipedia.org/wiki/DEFLATE
useRetroactiveConsumer	false	Sets whether or not retroactive consumers are enabled. Retroactive consumers allow non-durable topic subscribers to receive old messages that were published before the non-durable subscriber started.
userName		Sets the JMS userName used by connections created by this factory
warnAboutUnstartedConnectionTimeout	500 (ms)	Enables the timeout from a connection creation to when a warning is generated if the connection is not properly started via Connection.start() and a message is received by a consumer. It is very common to forget to start the connection so this option makes the default case to create a warning if the user forgets. To disable the warning just set the value to anything < 0 (e.g., -1).

Many of the properties listed in the table above can also be assigned via the brokerURL property itself; however, when assigning properties via the brokerURL, the property must be preceded with a **"jms."** prefix. For example, the following assigns a clientIDPrefix of "Pedro" to all connections that are rendered by the connection factory called, "linux01ConnectionFactory".

```
connection.linux01ConnectionFactory.brokerURL =
     tcp://linux01:61616?jms.clientIDPrefix=Pedro
```

The following is a code snippet that illustrates how the client leverages JNDI to get ConnectionFactory and Destination objects.

```
// We rely on the default JNDI InitialContext to lookup a connection
// factory and destination that are specified in the jndi.properties file.
javax.naming.Context ctx = new InitialContext();

// Create a connection factory for the target broker. From the factory
// you can create many connection objects, but typically an application
// will only have one connection
ConnectionFactory factory = (javax.jms.ConnectionFactory)
ctx.lookup("ConnectionFactory");

//Now have the connection factory render a connection
Connection conn = factory.createConnection();

// Lookup and create a queue
Queue myQ = (Queue) ctx.lookup("MyQueue");
```

The snippet of code above assumes that the jndi.properties file resides somewhere within the client's CLASSPATH; therefore, the JVM will automatically load its properties into the default InitialContext. More information on the jndi.properties file can be found here:
http://java.sun.com/products/jndi/tutorial/beyond/env/source.html#APPRES

If your client is deployed in an environment where there may exist multiple jndi.properties files, you may want to create an ActiveMQ-specific InitialContext to avoid any conflicts. Take for example, the situation where your ActiveMQ client is a Java servlet that will be deployed as a web application to any number of different web containers (e.g., Tomcat, JBoss, Jetty, WebLogic, etc.). You can create an 'amq.properties' file, which adheres to the jndi.properties or java.util.Properties format, and deploy it to the web application's "/WEB-INF/classes" directory. During your web application's initialize lifecycle phase, it can create an ActiveMQ initial context as follows:

```
Properties props = new Properties();

props.load( servlet.getServletContext().getResourceAsStream(
                      "/WEB-INF/classes/amq.properties" ) );

Context jmsCtx = new InitialContext (props);
```

Or if it is a standalone client, and assuming the amq.properties file is in the CLASSPATH, you can do the following:

```
Properties props = new Properties();
ClassLoader myLoader = this.getClass().getClassLoader();
props.load(myLoader.getResourceAsStream("amq.properties"));
Context jmsCtx = new InitialContext(props);
```

The one drawback with the two examples above is that your code does make specific references to ActiveMQ; therefore, it compromises the code's portability. This next section illustrates a more portable approach within the context of a web application container like Tomcat.

5.2 The LDAP-based JNDI

This section describes how to use the JNDI to store and retrieve ActiveMQ's JMS administered objects (i.e., connection factory and destination) to and from a LDAP directory server (DS).

The primary benefit of using a LDAP DS is that all information, especially information pertaining to ActiveMQ administered objects, can be centrally and securely stored and managed. This is especially attractive for large enterprise class environments that employ many JMS clients.

Java objects are stored in the LDAP DS according to rfc2713.

The following ActiveMQ administered objects are stored in the DS as javax.naming.Reference objects.

- org.apache.activemq.ActiveMQConnectionFactory
- org.apache.activemq.command.ActiveMQTopic
- org.apache.activemq.command.ActiveMQQueue

The following is an example LDAP Interchange Format (LDIF) file for an ActiveMQConnectionFactory object.

```
dn: cn=factory2,ou=adminobjects,o=amq,dc=example,dc=com
objectClass: javaNamingReference
objectClass: javaObject
objectClass: javaContainer
objectClass: top
```

```
cn: factory2
javaclassname: org.apache.activemq.ActiveMQConnectionFactory
javafactory: org.apache.activemq.jndi.JNDIReferenceFactory
javareferenceaddress: #0#blobTransferPolicy.uploadUrl#http://localhost:8080/
 uploads/
javareferenceaddress: #1#brokerURL#tcp://localhost:61670
javareferenceaddress: #2#objectMessageSerializationDefered#false
javareferenceaddress: #4#redeliveryPolicy.initialRedeliveryDelay#1000
javareferenceaddress: #3#prefetchPolicy.queuePrefetch#1000
```

In order to interact with a LDAP DS, via the JNDI, your Java client must use an implementation of the JNDI service provider interface (SPI) that stores and retrieves Java objects to and from the LDAP DS. Luckily, there is such an implementation and it is included in JDK 1.5 and higher. The package that provides this implementation is called, “com.sun.jndi.ldap” and the two classes in this package that your Java code will interact with are called, “LdapCtxFactory” and “LdapCtx”. However, as you’ll see, the only classes that you will need to directly reference are LdapCtxFactory and javax.naming.Context; the latter being an interface. The LdapCtxFactory renders LdapCtx objects, which implement the Context interface

The first thing your Java code must do is acquire and configure an instance of this package’s Context interface implementation (i.e., LdapCtx), which it will do so via LdapCtxFactory. The following sections describe three different methods for doing this.

5.2.1 Environment Properties Hashtable

With this method, your Java code specifies the fully qualified class name of the LdapCtxFactory through an entry in an environment property Hashtable, which represents the resulting Context’s environment. The following code snippet illustrates this.

```
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");
```

Using the same Hashtable, your code must then specify the URL for the target LDAP directory service along with the proper security credentials (username and password). These will all be used to establish a connection to that target directory service.

```
env.put(Context.PROVIDER_URL, "ldap://ldap.wiz.com:
389/ou=adminobjects,o=amq,dc=example,dc=com");

env.put(Context.SECURITY_PRINCIPAL, "uid=admin,ou=system");

env.put(Context.SECURITY_CREDENTIALS, "secret");
```

Notice how the PROVIDER_URL property must contain the distinguished name (DN) that pertains to the root of the directory information tree (DIT) that is used for storing the ActiveMQ administered objects.

Now that your environment properties have all been properly set, your code can create an instance of the Context as this code snippet illustrates:

```
Context ctx = new InitialContext(env);
```

Remember that the object doing the actual Context implementation is LdapCtx.

5.2.2 The jndi.properties File

Another method for acquiring an instance of the Context is to use the jndi.properties file. With this approach:

- You specify the service provider and its environment properties through the jndi.properties file.
- The configuration burden is shifted from the developer to whoever launches the application.
- Modifications to the configuration will not require corresponding modifications to the application's source code.

Here is an example jndi.properties file.

```
#
# This file must be placed in the JMS client's CLASSPATH
#
# The 'java.naming.factory.initial' property tells the JVM what JNDI context
# factory (or local JNDI provider) it should use for the application. In this
# case, we're telling it to use the  LdapCtxFactory object
# as the context factory. .

java.naming.factory.initial = com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://192.168.1.148:10389/ou=adminobjects,o=amq,dc=example,dc=
com
java.naming.security.credentials=secret
java.naming.security.principal=uid=admin,ou=system
```

The JMS client will then create the Context with one simple statement as this snippet illustrates.

```
Context ctx = new InitialContext();
```

Unlike the previous method, this method does not have to specify an environment Hashtable, because the default property values are automatically picked up from the jndi.properties file.

NOTE: The jndi.properties file must be located in the JMS client's class path.

More information on the jndi.properties file can be found at this URL.

http://java.sun.com/products/jndi/tutorial/beyond/env/source.html

5.2.3 System Properties

You can also use 'system properties' as yet a third method; for example.

```
>java -Djava.naming.factory.initial=\
com.sun.jndi.ldap.LdapCtxFactory  -Djava.naming.provider.url=\
ldap:// 192.168.1.148:10389/ou=adminobjects,o=ActiveMQ2,dc=example,dc=com\
-Djava.naming.security.credentials=secret \
-Djava.naming.security.principal=uid=admin,ou=system YourJMSApplication
```

As when using the jndi.properties file, the JMS client would then create the Context with one simple statement as this snippet illustrates.

```
Context ctx = new InitialContext();
```

5.2.4 JNDI Operations

Now that your JMS client has acquired a Context, it can perform JNDI operations (lookup, bind, rebind, unbind, etc.) against the corresponding LDAP DS. Your JMS client doesn't have to concern itself with transforming the ActiveMQ administered objects to and from Reference objects. Recall that the Reference object is one of only a handful of object types that can be stored in the DS.

The following code snippets, which assume you're using the jndi.properties or system properties methods, illustrate how relatively simple it is to perform some of the JNDI operations

Bind an ActiveMQConnectionFactory object to the DS.

```
import javax.naming.InitialContext;
import javax.naming.Context;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JndiBind {

      public static void main(String[] args) throws Exception {
         new JndiBind().init();
      }
```

```
        public void init() throws Exception {
           try {
             Context ctx = new InitialContext();
             // Create an ActiveMQ connection factory, give it some URL and save it in the directory.
             ActiveMQConnectionFactory factory1 = new ActiveMQConnectionFactory();
             factory1.setBrokerURL("tcp://localhost:61683");
             ctx.bind("cn=factory3", factory1);
            }
            catch(Exception e){
               e.printStackTrace();
            }
        }
}
```

Lookup an ActiveMQConnectionFactory object from the DS. Note how in this example we're casting the ActiveMQConnectionFactory that is returned from the lookup to a JMS ConnectionFactory. This helps isolate the code from the JMS provider.

```
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.jms.ConnectionFactory;

public class JndiLookup {

        public static void main(String[] args) throws Exception {
            new JndiLookup().init();
        }

        public void init() throws Exception {
         try {
           Context ctx = new InitialContext();
           ConnectionFactory factory2 = (ConnectionFactory)ctx.lookup("cn=factory2");
         }
         catch(Exception e){
           e.printStackTrace();
         }
        }
}
```

5.3 Tomcat

This is an example of how ActiveMQ administered objects can be configured into Tomcat 6.0's JNDI InitialContext implementation. The example provides insight into the following JNDI ActiveMQ object: "*org.apache.activemq.jndi.JNDIReferenceFactory*". This object is used to integrate ActiveMQ with other JNDI service providers.

Through the use of an object factory, the JNDI framework allows for object implementations to be dynamically loaded into the JNDI name space. The JNDIReferenceFactory object is such a factory and can be used by JNDI providers to bind and create ActiveMQ-administered objects to and from its JNDI name space. Therefore, JNDIReferenceFactory can be used to create the two JMS administered objects: *ConnectionFactory* and *Destination*.

Here's an example of how you can configure Tomcat 6.0's JNDI implementation to bind and render ActiveMQ connection factories and destinations.

Step 1 is to define the administered objects through your web application's context.xml file, which is located in $TOMCAT_HOME/webapps/<*webapp name*>/META-INF/. The following is an example context.xml file, which defines the JNDI namespace for your particualr web application.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/TomcatServletExample">


   <!-- Define the connection factory used to connect to the local broker -->
  <Resource
          auth="Container"
          brokerName="LocalActiveMQBroker"
          brokerURL="tcp://localhost:61616"
         description="JMS Connection Factory"
          factory="org.apache.activemq.jndi.JNDIReferenceFactory"
          name="jms/localConnectionFactory"
          type="org.apache.activemq.ActiveMQConnectionFactory"/>


  <!-- Define a queue called Q.TEST -->
  <Resource
      auth="Container"
      description="my Queue"
      factory="org.apache.activemq.jndi.JNDIReferenceFactory"
      name="jms/Q.TEST" physicalName="Q.TEST"
      type="org.apache.activemq.command.ActiveMQQueue"/>


  <!-- Define a queue called Q.REQ -->
  <Resource
      auth="Container"
      description="my other Queue"
      factory="org.apache.activemq.jndi.JNDIReferenceFactory"
      name="jms/Q.REQ" physicalName="Q.REQ"
      type="org.apache.activemq.command.ActiveMQQueue"/>

</Context>
```

You do not have to make any modifications to the …/WEB-INF/web.xml file to access the administered objects defined in the …/META-INF/context.xml file.

This example JndiTest.java servlet illustrates how the administered objects are acquired via the JNDI.

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import javax.jms.*;
import javax.naming.InitialContext;

public class JndiTest extends HttpServlet {

    public void doGet(HttpServletRequest request,  HttpServletResponse response)
        throws IOException, ServletException  {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String body = "";

          try {
              InitialContext init = new InitialContext();

              javax.jms.Queue destination =
              (javax.jms.Queue) init.lookup("java:comp/env/jms/Q.TEST");

              ConnectionFactory connectionFactory =
              (ConnectionFactory)
init.lookup("java:comp/env/jms/localConnectionFactory");

              Connection connection = connectionFactory.createConnection();
              Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

              TextMessage message = session.createTextMessage();
              message.setText("My text message was send and received");
              connection.start();

              MessageProducer producer = session.createProducer(destination);
              producer.setDeliveryMode(DeliveryMode.PERSISTENT);
              producer.send(message);
              message = null;

              MessageConsumer consumer = session.createConsumer(destination);
              message = (TextMessage) consumer.receive(1000);
              if( message!=null ) {
                  body = message.getText();
              }

              producer.close();
              consumer.close();
              session.close();
              connection.close();

          } catch (Exception e) {
              System.out.println(e.toString());
          }

        out.println("<html>");
        out.println("<head>");
```

For more information on Tomcat JNDI see "Tomcat 6.0 JNDI How-To".

5.4 Spring

This ActiveMQ web page provides a good description of how to configure a Spring-based ActiveMQ client, http://activemq.apache.org/spring-support.html

6 Broker Configuration

There are several ways to configure a broker, but the most flexible and powerful method of configuring a message broker is through an XML configuration file. This section will describe how to specify an XML configuration file and the most important elements in the configuration file along with their attributes and properties.

6.1 Specifying the Broker's XML Configuration File

A standalone broker is started by invoking the following command:

```
> $ACTIVEMQ_HOME/bin/activemq
```

By default, the standalone broker will look for the XML broker configuration file called "activemq.xml", which is located in the $ACTIVEMQ_HOME/conf directory; this directory also happens to be the broker's default CLASSPATH directory. However, you can specify a different configuration file by using the "xbean:file:" command line option, as follows:

```
> $ACTIVEMQ_HOME/bin/activemq  xbean:file:/path/to/foo.xml
```

Here's an example of using the "xbean:file:" command line option on Windows.

```
>  %ACTIVEMQ_HOME%\bin\activemq  xbean:file:C:/path/to/foo.xml
```

Note the forward slashes in the path to the file and if a directory name has spaces, then replace the space character with the URL encoded version for the space character, which is '%20'. For example,

```
C:/Program%20Files/AMQ/apache-activemq-5.0/conf/foo.xml
```

You can also use the "xbean:" option to specify an external XML configuration file; however, when using this option, the specified file should be located in the $ACTIVEMQ_HOME/conf directory (default CLASSPATH).

```
> $ACTIVEMQ_HOME/bin/activemq  xbean:foo.xml
```

More information on the activemq command's options can be found at this ActiveMQ web page:

http://activemq.apache.org/activemq-command-line-tools-reference.html

When starting an embedded broker, via the vm connector you can also use the "xbean:file:" and "xbean:" options. You assign these options and their corresponding XML configuration file paths to the vm connector's 'brokerConfig' option. When using the "xbean:" option, the corresponding XML configuration file must be located in the client application's CLASSPATH. The example below illustrates how to start an embedded broker via the vm connector and how to direct it to an external XML configuration file called "foo.xml", which in this case must be in the application's CLASSPATH.

```
                  vm://localbroker?brokerConfig=xbean:foo.xml
```

In this example, the "xbean:file:" option is used to specify an absolute path to the "foo.xml" file.

```
            vm://localbroker?brokerConfig=xbean:file:C:/tmp/foo.xml
```

6.2 The Broker's XML Configuration File

At the highest level, the broker's XML configuration file is comprised of a <beans> element followed by one or more <broker> and <bean id> elements. There is also a <bean class> element that needs to immediately follow the opening <beans> element.

Within the configuration file, you will typically have only one <broker> element, which is used to configure an instance of a broker. Subsequent sections will go into more detail on the <broker> element's attributes and sub-elements or nested elements. If you specify more than one <broker> element, you will end up with multiple distinct instances of a broker all running within the same VM.

Below is an example of a very simple broker XML configuration file. Note that the outermost element must be a <beans> element and that the <bean> element is required. A starter element (e.g., <beans>) must have an ending element (e.g., </beans>); however, in some cases, if your element has no nested elements it can be both a starter and ending element (e.g., <bean class="some.class"/>) that is comprised of only attributes. Note the forward slash before the '>' character; this indicates both a starter and ending element. Also, comments in both XML and HTML files are surrounded by a starting '<!--' and ending '-->'. For example,

```
<!-- This is a comment -->
```

```
<beans>

   <!-- Allows us to use system properties as variables in this configuration file -->
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

  <!-- DO NOT USE UNDERSCORES FOR THE BROKER NAME -->
  <!-- If  you do not specify a brokerName, then a default name of 'localhost' will be
       assigned to the broker. However, all brokers are given unique broker ids. -->
  <!-- The xmlns attribute is required and identifies the name space for the broker element -->
  <broker brokerName="mybroker" useJmx="true" xmlns="http://activemq.org/config/1.0">

    <!-- All the broker's sub-elements go here -->

  <!-- This is the closing element for the <broker> element. -->
  </broker>

<!-- This is the closing element for the <beans> element and the file. -->
</beans>
```

6.2.1 broker

The <broker> element is the most important element in the configuration file. It is the element that identifies and configures an instance of a broker. If you specify more than one <broker>

element, you will end up with multiple distinct instances of a broker all running within the same VM. The following is a very simple example of a <broker> element with three attributes: brokerName, useJmx, and xmlns. Note that the element has no nested elements; therefore, it is both a starter and ending element. You will typically have one or more nested elements. The 'xmlns' attribute is not listed in the table below; it represents the XML name space for this XML file/document[1].

```
<broker brokerName="mybroker" useJmx="true" xmlns=http://activemq.org/config/1.0 />
```

Attributes For The <broker> element.

Attribute	**Default Value**	**Description**
advisorySupport	true	Allows the support of advisory messages to be disabled for performance reasons. **CAUTION**: setting this option to 'false' precludes the broker from forwarding messages to other brokers.
brokerName		Sets the name of the broker, which must be unique across the network of brokers. If a name is not specified, one will be given to the broker.
brokerObjectName		Sets the JMX ObjectName for this broker[2]
dataDirectory		Sets the default directory in which the data files will be stored by the default message store mechanism.
deleteAllMessagesOnStartup	false	Sets whether or not all messages are deleted from the message store on startup – used primarily for testing.
masterConnectorURI		Used for setting up a 'pure' Master/Slave configuration. See section 8.2.1.1.
persistent	true	Used for turning persistence on or off.
populateJMSXUserID	false	Sets whether or not the broker should populate the JMSXUserID message header field. This field contains the message producer's authenticated user name.
shutdownOnMasterFailure	false	Sets whether a slave, in a pure master/slave cluster, should shutdown if the master fails.

[1] See http://www.w3schools.com/xml/xml_namespaces.asp
[2] See http://java.sun.com/j2se/1.5.0/docs/api/javax/management/ObjectName.html for more information on how this attribute is used.

start	true	Sets whether or not the broker is started along with the Spring application context it is defined within. Normally you would want the broker to start up along with the application context, but sometimes when working with JUnit tests you may wish to start and stop the broker explicitly yourself.
useJmx	true	Sets whether or not the Broker's services should be exposed into JMX.
useLocalHostBrokerName	false	If the brokerName attribute is not set, then this sets whether or not the broker name should take on the name assigned to the local machine. If both the brokerName and this attribute are set to false, then the broker name will default to 'localhost'. However, all brokers are assigned unique broker ids. Only available in 5.0.
useVirtualTopics	true	Sets whether or not Virtual Topics should be supported by default if they have not been explicitly configured.

The following sections describe some of the <broker> element's more important nested elements.

6.2.2 transportConnector

The <*transportConnector*> element, which is wrapped by the <*transportConnectors*> element, is one of the <broker> element's nested elements. It is used to specify an IP address and port number on which the broker will listen for and accept network connection requests from both clients and other brokers. You can specify zero or more <transportConnector> elements. If you don't specify this element, then the broker will not listen for and accept connection requests.

Here's a very simple example of a broker configuration file that defines a broker called "localbroker" that uses the tcp connector URI to listen for connection requests on its local host's port number 61616.

```
<beans>

  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

  <broker brokerName="mybroker" useJmx="true" xmlns="http://activemq.org/config/1.0">
```

```
   <transportConnectors>
     <transportConnector name="openwire" uri="tcp://localhost:61616"  />
   </transportConnectors>

  </broker>

</beans>
```

This example illustrates how the broker can open multiple <transportConnector> elements using different low-level connector URIs. Note how the URI scheme (e.g., tcp://) identifies the underlying network protocol.

```
<beans>

  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

  <broker brokerName="mybroker" useJmx="true" xmlns="http://activemq.org/config/1.0">

    <transportConnectors>
     <transportConnector name="openwire" uri="tcp://localhost:61616" />
     <transportConnector name="ssl"      uri="ssl://localhost:61617"/>
     <transportConnector name="stomp"    uri="stomp://localhost:61613"/>
   </transportConnectors>

  </broker>

</beans>
```

This table lists all of the attributes associated with the <transportConnector> element.

Attribute	Default Value	Description
discoveryURI	null	Enables a discovery agent for this transport connector. The broker will listen for discovery advertisements from other brokers using this URI. See section 3.3.3.
name	null	This is the name assigned to the corresponding transport connector object. If a name is not given, the name will default to the string assigned to the 'uri' attribute; the ':' will be

		replaced with '_'.
uri		Specifies the URI for this transport connector. For example, tcp://localhost:61617. See section 3.

The two most important attributes are the 'uri' and 'discoveryURI' attributes. The uri attribute is required and the discoveryURI is required if you have a <networkConnector> element that makes reference to it.

6.2.3 networkConnector

The <*networkConnector*> element, which is wrapped by the <*networkConnectors*> element, is used to specify the IP address and port to which the broker will create a connection to another broker.

You can specify zero or more <networkConnector> elements; however, recall that through the 'static' composite URI connector, you can specify one or more IP address and port number combinations that the broker will connect to. If you do not specify any <networkConnector> elements, the broker will not initiate/establish connections to other brokers.

The example below illustrates a configuration that will force the broker to connect to two remote brokers (one running on linux01 and the other running on linux02). Note that the networkConnector's uri attribute has two IP address and port number combinations assigned to it.

```
<broker brokerName="mybroker" useJmx="true" xmlns="http://activemq.org/config/1.0">

   <networkConnectors>
     <!-- Establish connections to linux01 and linux02 -->
     <networkConnector
            name="linux01 and linux02"
            uri="static://(tcp://linux01:61616,tcp://linux02:61616)" />
   </networkConnectors>

 </broker>
```

The <networkConnector> element's uri attribute requires that you use either the 'static', 'rendezvous' or 'multicast' URI.

The multicast or rendezvous URIs are used as part of the automatic discovery protocol, which brokers use to automatically find one another (see sections 3.2.7 and 3.2.8). In the example below, the broker will advertise discovery packets out through the default multicast port and also receive discovery packets through the default multicast port. The brokers use the discovery packets and multicast to automatically join a cluster of brokers listening on and advertising through the given multicast IP address. When responding to a discovery packet, the broker will not only indicate that it is available to join the cluster, but also what IP address and port number it is listening on for connection requests. In the example below, the broker is listening on the “localhost:61616” IP address and port number combination.

```
<beans>

   <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

  <broker brokerName="mybroker" useJmx="true" xmlns="http://activemq.org/config/1.0">


   <transportConnectors>
      <transportConnector name="openwire"
                    uri="tcp://localhost:61616"
                    discoveryURI="multicast://default" />
   </transportConnectors>

   <networkConnectors>
     <!--  Send out discovery advertisement packets to auto discover the other brokers -->
     <networkConnector
             name="discovery"
             uri="multicast://default"/>
   </networkConnectors>

  </broker>
</beans>
```

This table lists and describes attributes for the <networkConnector> element.

Attribute	**Default Value**	**Description**
conduitSubscriptions	true	Multiple consumers subscribing to the same destination are treated as one consumer by the network. See example after this table.
decreaseNetworkConsumerPriority	false	If set to true, the priority associated with a queue consumer will be decreased the further away it is, in network hops, from

		the broker that received the message.
dispatchAsync	false	See section 12.3 for more information on asynchronous dispatching.
duplex	false	If true, a network connection will be used to both produce *AND* consume messages. In other words, messages can flow in either direction. This is useful for hub and spoke scenarios when the hub is behind a firewall or for clients implementing a request/reply protocol. This is available only for ActiveMQ version 5.0 and higher.
dynamicallyIncludedDestinations	empty	Any messages destined to destinations in this list will be forwarded across the network. An empty list means all destinations not in the excludedDestinations list will be forwarded.
dynamicOnly	false	If set to true, the broker will forward messages to the endpoint broker (broker at the other end of the connection) only if that endpoint broker has active consumers.
excludedDestinations	empty	Any messages destined to any destination in this list will not be forwarded across the network.
name	"bridge"	The name assigned to the corresponding network connection object.
networkTTL	1	This specifies the number of brokers in the network that messages and subscriptions can pass through. This basically represents the number of network hops that a subscription or message can go through on this particular network to reach its final destination.
staticallyIncludedDestinations	empty	Any messages destined for destinations in this list will always be passed across the network - even if no consumers have ever registered an interest
uri	Empty	Defines the URI for this network connector. R*equires that you use either the 'static', 'rendezvous' or 'multicast'*

		URI.
userName & password	Empty	The userName and password attributes are set in order to allow the broker to connect to a remote broker that has 'authentication' enabled. More information on authentication can be found in section 16.

You should pay close attention to the 'dynamicOnly' and 'networkTTL' attributes and their default settings, which are not properly set for large networks of brokers.

conduitSubscriptions
This example helps to further explain "conduitSubscriptions". Suppose you have two brokers, A and B. Connected to broker A, you have a consumer that subscribes to a queue called Q.TEST. Connected to broker B, you have two consumers that subscribe to the same queue. Then you start a producer on broker A that writes 30 messages to Q.TEST. By default, (conduitSubscriptions=true), 15 messages will be sent to the consumer on broker A and the resulting 15 messages will be sent to the two consumers on broker B. This is because broker A views the two subscriptions on broker B as one. If you set conduitSubscriptions to "false", then each of the three consumers is given 10 messages.

6.2.4 systemUsage

The <*systemUsage*> element is used to set maximum limits on the amount of system resource usage allowed for the ActiveMQ broker. The following is an example element taken from the default broker XML configuration file (activemq.xml).

```
<systemUsage>
   <systemUsage>
         <memoryUsage>
                   <memoryUsage limit="10 mb" percentUsageMinDelta="20"/>
         </memoryUsage>
         <tempUsage>
                   <tempUsage limit="100 mb"/>
         </tempUsage>
         <storeUsage>
                   <storeUsage limit="1 gb" name="foo"/>
         </storeUsage>
   </systemUsage>
</systemUsage>
```

The <memoryUsage> element is used to set the maximum amount of memory the broker will use. The <tempUsage> element is used to set the maximum size of the message store used for non-persistent messages that may optionally overflow from memory awaiting dispatch. The <storeUsage> element is used to control the maximum size of the ActiveMQ message store (i.e., the store used for persistent messages).

One relevant boolean property is called “sendFailIfNoSpace”. When turned on, this property forces the ActiveMQ message broker to simply throw a JMSException and not block the corresponding send if there is insufficient space.

```
<beans>
…
 <broker>
  …
   <systemUsage sendFailIfNoSpace="true">
        <memoryUsage>
          <memoryUsage limit="400kb" />
        </memoryUsage>
        <storeUsage>
          <storeUsage limit="10mb" />
        </storeUsage>
        <tempUsage>
          <tempUsage limit="64mb" />
        </tempUsage>
    </systemUsage>
   …
 </broker>
  …
</beans>
```

Here’s another example, and note how the settings are addressed for a particular message store.

```
<beans>

…

<broker xmlns="http://activemq.apache.org/schema/core"   persistenceAdapter="#store">

   …
  <systemUsage>
      <systemUsage sendFailIfNoSpace="true" >
        <memoryUsage>
          <memoryUsage limit="400kb" />
        </memoryUsage>
        <storeUsage>
          <storeUsage limit="10mb" store="#store" />
        </storeUsage>
        <tempUsage>
          <tempUsage limit="64mb" />
        </tempUsage>
      </systemUsage>
 </systemUsage>
 …
 </broker>

…

  <bean id="store" class="org.apache.activemq.store.amq.AMQPersistenceAdapter" >
    <property name="directory" value="target/amqdata" />
    <property name="maxFileLength" value="1000000" />
    <property name="checkpointInterval" value="5000" />
    <property name="cleanupInterval" value="5000" />
  </bean>

</beans>
```

Here is an interesting thread on ActiveMQ's user forum regarding the maxFileLength property. Note the situation described by Richard and Gary's response.

http://www.nabble.com/Usage-Manager-Store-is-Full---Root-Cause--td22147570.html

6.2.5 persistenceAdapter

The JMS API specifies two modes of message delivery: PERSISTENT and NON-PERSISTENT. The PERSISTENT mode of delivery, which is the default, instructs the JMS provider to take extra measures in order to ensure that a message is not lost while being routed to its final destination. To ensure this level of guaranteed message delivery, the message broker will persist or write a message out to stable storage or the "message store" as it is sometimes referred to. Persisting the message to the message store will ensure that the message is not lost if the broker were to fail. There are various different types of message store implementations that you can configure for ActiveMQ, and the <persistenceAdapter> element is used to specify which of these implementations the broker is to employ when persisting messages.

6.2.5.1 amqPersistenceAdapter

Starting with version 5.0 of ActiveMQ, the default message store for ActiveMQ is based on an embeddable transactional message storage solution that is supposed to be extremely fast and reliable. For more information on ActiveMQ's default message store and how it is configured, go to this ActiveMQ web page:

http://activemq.apache.org/amq-message-store.html

6.2.5.2 journaledJDBC

This is the default message store for ActiveMQ 4.x and earlier; it uses a high performance journal along with an embedded database called "Derby".

For more information on this store and how to configure it, go to this ActiveMQ web page:

http://activemq.apache.org/persistence.html

The above web page will also describe how to use other databases with and without the journal.

6.2.6 destinations

This is a sub-element of the <broker> element and it is used for specifying the destinations (queues and topics) that are to be created on broker startup as opposed to waiting for a client to connect to the broker and register those destinations.

```
<broker brokerName="mybroker" persistent="true" useShutdownHook="false">
    …

    <destinations>
      <queue name="MyQueue" physicalName="example.MyQueue" />
      <topic name="MyTopic" physicalName="example.MyTopic" />
    </destinations>

    …

</broker>
```

The name and physicalName attributes should match the corresponding names found in the client's JNDI context.

6.3 UseDedicatedTaskRunner

By default, ActiveMQ dedicates a thread to service a destination; therefore, if you create a large number of destinations, there will also be an equally large number of threads. However, you can configure ActiveMQ to instead use a thread pool to service destinations. A thread pool can help reduce the overall number of threads used by ActiveMQ, which may also have performance implications. To force ActiveMQ to use a thread pool, set the system property called, “org.apache.activemq.UseDedicatedTaskRunner” to false. Note that this system property is set to true in the ActiveMQ startup script, which is found in the ACTIVEMQ_HOME/bin directory.

```
if "%ACTIVEMQ_OPTS%" == "" set ACTIVEMQ_OPTS=-Xmx512M -Dorg.apache.activemq.UseDedicatedTaskRunner=true
```

7 Wire Protocols

ActiveMQ clients and message brokers communicate with one another by streaming command objects, which include messages, to one another through network connections. At the client, all communications between the client and broker is taken care of by the ActiveMQ client class files or libraries.

The broker and client can optionally both reside in the same Java Virtual Machine (JVM or VM). In this configuration, the broker is referred to as an embedded or intra-VM broker and it communicates with its clients via direct method invocation instead of through a network communication channel. Embedded brokers are further described in section 3.2.1.

ActiveMQ currently employs two encoding formats (wire protocols) used to stream the command objects through a network connection. The two are called “OpenWire” and “STOMP”.

STOMP, which stands for Streaming Text Orientated Messaging Protocol, is used to support ActiveMQ clients written in languages other than Java and it is also used to develop bridges or gateways between ActiveMQ and other JMS providers (e.g., MQSeries).

The OpenWire protocol is the default encoding format used by the native Java, C/C++, and .NET ActiveMQ client libraries, which are the only native client libraries currently available. Unlike STOMP, which is text-based, OpenWire employs a more efficient binary format.

The OpenWire protocol has a handful of parameters, which are assigned through a “Connector URI” (See section 3.1). The following table lists and describes the OpenWire parameters.

OpenWire Options

Parameter Name	Default Value	Description
stackTraceEnabled	true	Should the stack trace of exception that occurs on the broker be sent to the client?
tcpNoDelayEnabled	false	Does not affect the wire format, but provides a hint to the peer that the TCP NODELAY option should be enabled on the communications Socket. Setting the TCP_NODELAY option will turn off Nagle's algorithm, which is used to improve network efficiency. This is also referred to as "nagling". The algorithm aggregates small messages on the sending side into larger TCP packets thus improving overall network efficiency. When the Nagle algorithm is turned off, packets are immediately transmitted thus increasing performance, but potentially compromising network efficiency. On most operating systems and JVMs, the Nagle algorithm is turned on by default.
cacheEnabled	true	Should commonly repeated values be cached so that less marshalling occurs?
tightEncodingEnabled	true	For more information on tight encoding, see section 17.9
prefixPacketSize	true	Should the size of the packet be prefixed before each packet is marshaled?
maxInactivityDuration	30000	The maximum inactivity duration (before which the socket is considered dead) in milliseconds. On some platforms it can take a long time for a socket to appear to die, so we allow the broker to kill connections if they are inactive for a period of time. Used by some transports to enable a keep alive heart beat feature. Set to a value <= 0 to disable inactivity monitoring.

maximumConnections	Integer.MAX_VALUE	Specifies the maximum number of connections that will be accepted by the TcpTransportServer. You can apply this option to the URI attribute that is associated with the <transportConnector> element. The <transportConnector> element is found in the broker's activemq.xml file.
cacheSize	1024	If cacheEnabled is true, then this specifies the maximum number of values to cache.

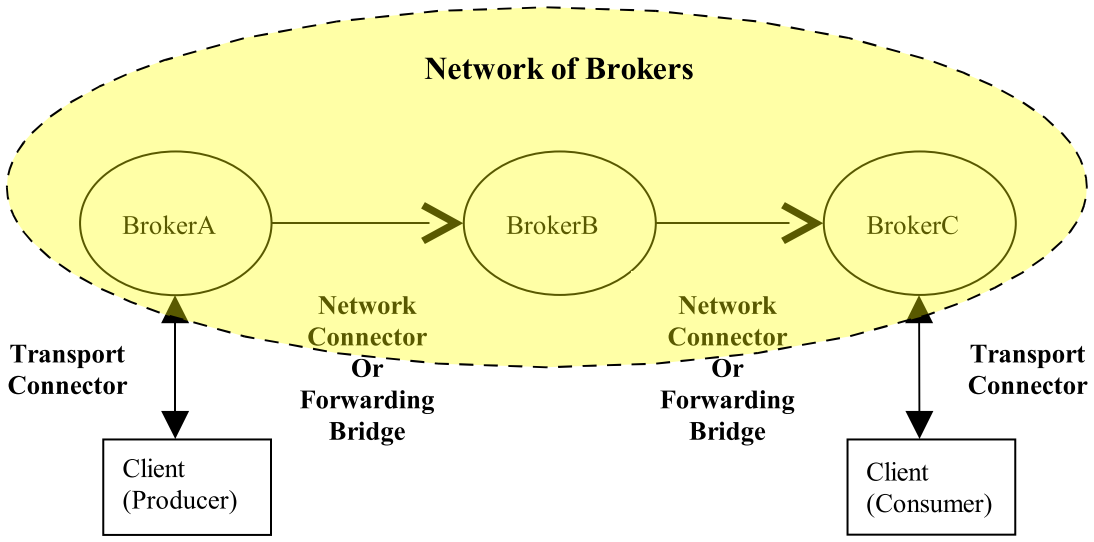
This is an example of how OpenWire parameters are specified through a Connection URI.

```
tcp://localhost:61616?wireFormat.cacheEnabled=false&wireFormat.tightEncodingEnabled=false
```

8 Network of Brokers

8.1 Store and Forward

A store and forward network of brokers is one that supports the forwarding of messages across multiple brokers in order to reach a destination with an active consumer. Take for example a network of three brokers: A, B, and C. A is connected to B, B is connected to C, a producer is connected to A, and a consumer is connected to C.



When the producer on A produces a message to a queue on A, we would like that message to be forwarded to the corresponding queue on C so that the consumer connected to C can read that message from the queue on C. In order to achieve this, all the brokers must have appropriate transport and network connector elements assigned to their configurations. That is, broker A must have a network connector to broker B and broker B must have a network connector to broker C. At a minimum, brokers B and C must also have transport connectors, which they use to listen for and accept incoming network connection requests.

One thing to note is that the default settings for the network connector element's "networkTTL" and "dynamicOnly" attributes are set to "1" and "false", respectively. A networkTTL setting of 1 precludes a message from being forwarded across more than one broker. A dynamicOnly setting of false will not preclude the broker from forwarding a message to another broker that no longer has an active consumer. With the default networkTTL setting, the message sent from the producer on A would reach B, but never make it to C. In order to reach C, networkTTL must be given a value of at least '2'. With the default dynamicOnly setting, it is very likely that the broker will forward a message to a broker that at one time had an active consumer for the destination; therefore, the messages forwarded to that broker may be permanently lost. More information on the network connector element's attributes can be found in section 6.2.3.

CAUTION: When reading through the documentation on the ActiveMQ web site, you may run across a broker boolean attribute called, "advisorySupport". The documentation states that setting this attribute to false (default is true) will improve performance a little because it disables the support for advisory messages (see section 9); however, setting this option to 'false' precludes the broker from forwarding messages to other brokers.

8.2 High Availability

8.2.1 Master/Slave Broker Configurations

The configurations associated with ActiveMQ's Master-Slave network of brokers provide for fault tolerance at the broker level. The configurations are comprised of a cluster of brokers, where each cluster has one master along with one or more dormant slave brokers that get called into service (become the master) if and when the master fails. The slave is dormant because while it waits to be called into service, it has all network connectivity disabled; therefore, it cannot accept connection requests from clients and other brokers and cannot initiate connections to other brokers. ActiveMQ does not have the capability for an active or hot standby Master-Slave configuration.

The Mater-Slave functionality is only available with version 4.0 and higher of ActiveMQ.

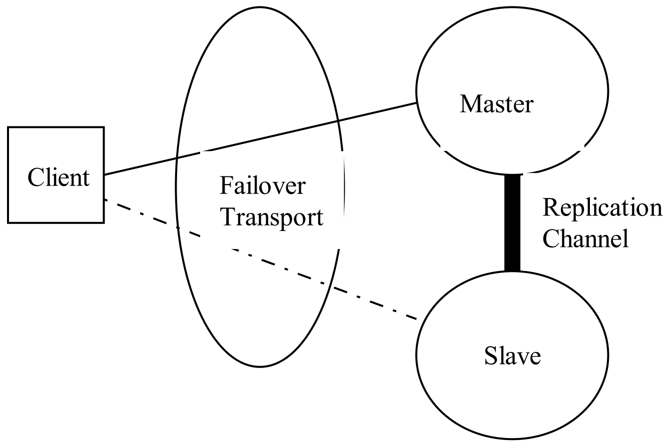
These are the three types of Master-Slave broker configurations.

Master Slave Type	Requirements	Pros	Cons
Pure Master Slave	ActiveMQ 4.0 or higher	No central point of failure.	Requires manual restart to bring back a failed master and can only support 1 slave
Shared File System Master Slave	A Shared Networked File System (NFS). ActiveMQ 4.1 or higher	Run as many slaves as needed. Automatic recovery of old masters	Requires a distributed shared file system (e.g., NFS or SAN).
JDBC Master Slave	A shared database ActiveMQ 4.1 or higher	Run as many slaves as needed. Automatic recovery of old masters	Requires a shared database. Also relatively slow as it cannot use ActiveMQ's high performance journal

If a shared network file system (NFS) is available, then the "Shared File System Master-Slave" configuration is recommended. This configuration also uses ActiveMQ's high-performance journal, which serves as a database cache.

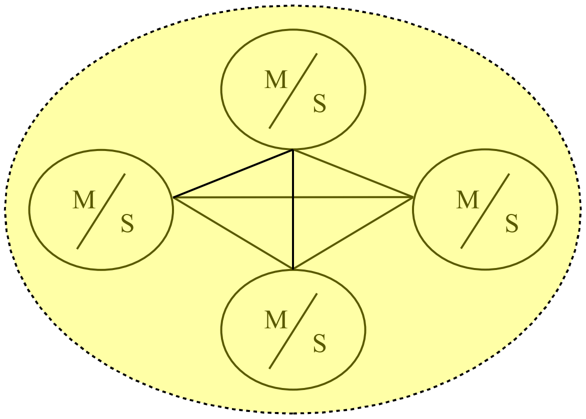
8.2.1.1 Pure Master Slave

A pure Master-Slave broker configuration is one that does not use a shared file system or database. In this configuration, a master has only one slave and it maintains all message and state information with the slave through a 'replication' (communications) channel. The advantage of this configuration is that a shared file system or database is not required; however, the disadvantage is that performance is impacted because of the overhead involved with communicating all message and state information through the replication channel.



Pure Master-Slave Configuration

This pure mast-slave configuration can be expanded upon by connecting multiple master brokers to form a large, highly available network or cluster of brokers.



A Highly Available Network Comprised of Pure Master-Slave Configurations

These are the characteristics of a pure Master-Slave deployment:

- A master must constantly keep the slave synchronized via the replication channel. This includes all messages, message states, acknowledgements and transactional states.
- While a slave is actively connected to the master, it does not permit or start any network or transport connectors; during this time, its sole purpose is to mirror the state of the master.

  NOTE: See the following JIRA in relation to the above bullet-item: https://issues.apache.org/activemq/browse/AMQ-1511

- The master will only respond to a client when the corresponding message exchange (synchronization) has been successfully passed to the slave. For example, a transaction commit will not complete until the master and the slave have processed the commit.

- If the master fails, the slave has two optional modes of operation:

    - Starts all its network and transport connectors thus allowing clients connected to the master to resume on the slave

    - Can be configured to close down. In this mode, the slave is simply used to duplicate state for the master. In the event of a catastrophic failure, the slave's data repository can be used as a backup for recovery.

Clients that interact with a pure master-slave deployment should employ a failover transport that has the master as the first transport in the failover list and has the *randomize* parameter disabled (i.e., set to false). This will ensure that it first connects to the master and if and when the master fails, it will move on to the next transport on the list, which is the slave. Take the following transport, which the client would have in its jndi.properties file, as an example.

```
   failover:(tcp://masterhost:61616,tcp://slavehost:61616)?randomize=false
```

Setting **randomize** to 'false' disables randomness so that the transport will always try the master first, then the slave if it can't connect to the master. And again recall that the slave does not accept connections until it becomes the master.

Limitations of Pure Master Slave

Only one slave can be connected to the Master.

A failed master cannot be re-started without first shutting down the slave; there is no automatic synchronization from slave back to master. The master must be manually synchronized with the slave as described below.

Recovering a Master Slave

After a master has failed it must first be re-synchronized with the slave prior to being restarted. Here are the steps for this manual re-synchronization process:

1. Shutdown the slave broker; the clients connected to the slave do not need to be shutdown because they will begin to retry connection attempts to both the master and the slave. Again, this only applies to clients that are using the 'failover' transport as described above.

2. Copy the slave's data directory to that of the master broker's

3. Start the master, then start the slave.

Configuring The Pure Master Slave

A master broker doesn't require any special configuration; it's a normal broker until a slave broker connects to it. The one exception to this rule is the “waitForSlave” attribute, which was introduced with the release of ActiveMQ 5.2. If you set this attribute to true, it does mark the broker as being a master broker in a pure master/slave configuration and it also tells the master broker not to fully initialize until the slave broker has connected to it. So if you set this attribute to true, the master broker will not accept connections from either clients or other brokers until a connection identifying itself as being from the slave broker is received. This ensures that the master and slave brokers remain in sync with one another. Example configuration:

```
<broker waitForSlave="true"… >

  . . .
</broker>
```

To mark a broker as a slave there is just one property to set as this example, taken from the slave broker’s XML configuration file, demonstrates:

```
<broker masterConnectorURI="tcp://masterhost:62001"… >

  . . .
</broker>
```

Broker Property	default	Description
masterConnectorURI	null	URI to the master broker e.g. ***tcp://masterhost:62001***
shutdownOnMasterFailure	false	If set to true, the slave will shut down if the master fails, otherwise the slave will take over as being the new master. The slave ensures that there is a separate copy of each message and acknowledgement on another machine which can protect against catastrophic hardware failure. If the master fails you might want the slave to also shut down because you may always want to duplicate messages to 2 physical locations to prevent message loss on catastrophic data centre or hardware failure. If you would rather the system keep on running after a master failure then leave this flag as false.

Configuring the Authentication of the Slave

In ActiveMQ 4.1 and higher, you can use a **<masterConnector/>** element to also configure a slave broker. This element also provides you with the capability of specifying a userid and password for those cases where the master requires authentication. The example below illustrates the use of the <masterConnector> element; note that it is a sub-element of the <services> element.

```
<broker brokerName="slave" useJmx="false" shutdownOnMasterFailure="true"
xmlns="http://activemq.org/config/1.0">

  <transportConnectors>
    <transportConnector uri="tcp://localhost:62002"/>
  </transportConnectors>

  <services>
    <masterConnector remoteURI= "tcp://localhost:62001" userName="Chuckee"
     password="Cheese"/>
  </services>

</broker>
```

8.2.1.2 Shared File System Master Slave

If your organization uses a Storage Area Network (SAN - http://en.wikipedia.org/wiki/Storage_area_network) or a highly reliable Networked File Service (NFS), then you are recommended to use the "Shared File System Master-Slave" configuration. With this configuration, you set up two or more brokers that reference the same, shared data directory (message store). The first broker to acquire the exclusive lock for the shared data directory becomes the master broker. Those brokers that cannot acquire the lock become dormant slaves that periodically attempt to acquire the lock. The slaves are "dormant" because they do not activate their transport and network connectors until they become a master. If and when the master fails, the lock gets released, which allows one of the slaves to acquire the lock and thus becomes the new master. When a slave becomes a master, it immediately activates its transport and network connectors. In the meantime, clients will loose their connections to the failed master and will automatically failover to the new master. So the idea here is to provide high availability by introducing redundancy at both the message broker and message store levels.

The following snippet, taken from a broker's XML configuration file, demonstrates how to set up a shared master/slave configuration.

```
<! In this particualr case, all brokers in the master/slave cluster
   must share the same name. -->
<broker xmlns="http://activemq.org/config/1.0"
        brokerName="master"
        dataDirectory="${activemq.base}/shared-data" >

 <!-- The transport connectors ActiveMQ will listen to -->
 <transportConnectors>
       <transportConnector uri="tcp://localhost:61617" />
 </transportConnectors>
 <managementContext>
      <managementContext connectorPort="1399"
       jmxDomainName="org.apache.activemq"/>
 </managementContext>

</broker>
```

Lets assume that the above configuration is given to two brokers, each running on separate machines, and that the dataDirectory attribute points to a highly reliable shared file system, which in this particular case is located at, “${activemq.base}/shared-data”. The final location to the resulting data directory will actually be "${activemq.base}/shared-data/master”, because the value assigned to the brokerName attribute is appended to the path assigned to the dataDirectory attribute. If you do not specify the brokerName attribute, the default name of “localhost” will be assigned to the broker.

The first broker to grab the lock to the shared data directory becomes the master. The second broker, which becomes the slave, displays the following warning message and will wait to acquire the lock.

```
WARN  AMQPersistenceAdapter - Waiting to Lock the Store C:\Program Files\AMQ\apache-activemq-5.1-SNAPSHOT\bin\..\masterslave-data\master
```

Again, while waiting for the lock to be released, the slave broker is dormant and will not activate its transport or network connectors. Client and other brokers will not be able to connect to it.

***CAUTION**: It is very important that you use the same name for all brokers in a shared master/slave configuration cluster; if not, they will end up using different data directories and will not comprise a master/slave cluster. For example, if one broker is called “master” and the other “slave”, then they will end up referencing the data directories "${activemq.base}/shared-data/master” and "${activemq.base}/shared-data/slave”, respectively.*

Another approach to setting up a shared master/slave configuration is to use the persistenceAdapter element instead of the dataDirectory attribute as this example illustrates.

```
<broker useJmx="false"  xmlns="http://activemq.org/config/1.0">

    <persistenceAdapter>
      <journaledJDBC dataDirectory="/sharedFileSystem/broker"/>
    </persistenceAdapter>

…
</broker>
```

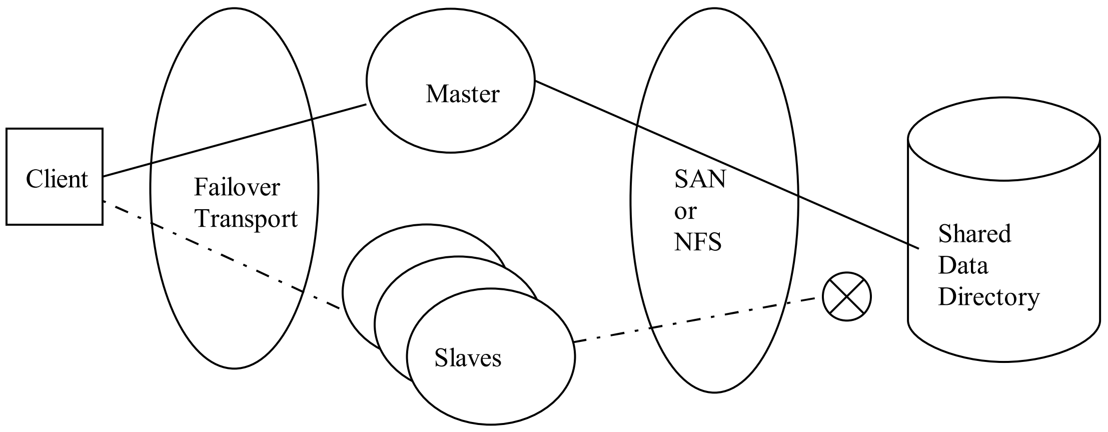
When using the above method, the slave will put out this different message letting you know that is has not acquired the lock.

```
INFO  journalPersistenceAdapterFactory - Journal is locked... waiting 10 seconds for the journal to be unlocked
```

Clients should be using the failover transport with a list of all the brokers in this particular Master-Slave cluster. For example, if there are three brokers in the cluster, the failover transport would look something like this.

```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

Randomization is turned on (default behavior) and only the master broker activates its transport connectors; therefore, the clients can only connect to the master.



Shared File System Master-Slave

8.2.1.3 JDBC Master Slave

Except for the following, the JDBC Master-Slave configuration behaves very much like the Shared Master Slave configuration.

1. Instead of a shared file system, the cluster of brokers is pointed at a shared database.
2. The ActiveMQ high performance journal cannot be used; therefore, performance will not be as good as when using the Shared File System Master Slave.
3. Use the <jdbcPersistenceAdapter> element, instead of <journaledJDBC>, to point the brokers to the shared database.

8.3 Isolating Your Broker From Other Brokers

If you are within a development or testing environment, you may have a requirement to isolate your broker from other brokers or network of brokers within that environment. To address this requirement, edit your broker's XML configuration file as follows.

1. Give the <broker> element's 'brokerName' attribute a unique value (name).
2. Through the <broker> element's <persistenceAdapter> sub-element, ensure that the broker is given its own unique data directory. If it shares a data directory with another broker, then it will become part of a shared master/slave cluster. If the broker is not given a <persistenceAdapter> element, then it will, by default, be given a unique data directory based on its broker name.
3. Ensure that the port numbers assigned via the <broker> element's <transportConnector> and <managementContext> sub-elements do not conflict with those assigned to other brokers on the same machine.
4. If you do not want your broker auto-discovering other brokers, then remove any 'multicast' connector URIs in the <transportConnector> and <networkConnector> elements so that your broker does not automatically discover and connect to other brokers. If you want your broker to auto-discover other brokers within your own private cluster, then ensure that you use the same unique multicast IP address for all brokers in your private cluster. For example, this snippet from an XML configuration file has changed the 'default' multicast IP address to a unique multicast IP address of 239.255.2.5

```
<transportConnectors>
    <transportConnector
        name="default"
        uri="tcp://localhost:61616"
        discoveryURI="multicast://239.255.2.5"
    />
</transportConnectors>

<networkConnectors>
   <networkConnector
      name="default"
      uri="multicast://239.255.2.5"
   />
</networkConnectors>
```

You can then give your XML configuration file a unique name and pass it to the activemq script via its "xbean:file:" option as follows.

```
 > $ACTIVEMQ_HOME/bin/activemq  xbean:file:/path/to/mybroker.xml
```

9 Advisory Messaging

ActiveMQ provides a facility called, “Advisory Messaging”, which allows your client application to monitor the system by subscribing to a predefined set of administrative or advisory topics. ActiveMQ automatically publishes messages to these topics to reflect the following types of system events:

- The stopping and starting of consumers, producers and connections
- Temporary destinations being created and destroyed
- Messages expiring on topics and queues
- Brokers sending messages to destinations with no consumers.
- The stopping and starting of connections

Please refer to the following ActiveMQ web site page for a complete list of advisory events.

http://activemq.apache.org/advisory-message.html

You’ll note that starting with ActiveMQ 5.2, additional advisory messages were introduced. By default, the publishing of the majority of these new advisory messages is turned off. To turn them on, use the destination policy element as this example illustrates (note the use of the wildcard characters to specify all queues and all topics).

```
<broker>
   …
  <destinationPolicy>
    <policyMap>
     <policyEntries>

      <policyEntry   queue=">"
       memoryLimit="5mb"
       advisoryForConsumed="true"
       advisoryForDelivery="true"
       advisoryWhenFull="true"
       advisoryForSlowConsumers="true"
       advisdoryForFastProducers="true"
       advisoryForDiscardingMessages="true" />

      <policyEntry topic=">"
       advisoryForConsumed="true"
```

```
         advisoryForDelivery="true"
         advisoryWhenFull="true"
         advisoryForSlowConsumers="true"
         advisdoryForFastProducers="true"
         advisoryForDiscardingMessages="true" />

       </policyEntries>
      </policyMap>
    </destinationPolicy>
•••
</broker>
```

All advisory topics have the "ActiveMQ.Advisory" prefix assigned to their names and there are two basic types of advisory topics: client and destination. The following lists the client advisory topics:

- ActiveMQ.Advisory.Connection – sent messages that reflect all connection start and stop events.
- ActiveMQ.Advisory.Consumer.Queue – sent messages that reflect all consumer start and stop events on all queues.
- ActiveMQ.Advisory.Producer.Queue – sent messages that reflect all producer start and stop events on all queues.
- ActiveMQ.Advisory.Consumer.Topic – sent messages that reflect all consumer start and stop events on all topics.
- ActiveMQ.Advisory.Producer.Topic – sent messages that reflect all producer start and stop events on all queues.

These are the destination advisory topics:

- ActiveMQ.Advisory.Queue – sent messages that reflect all queue create and destroy events.
- ActiveMQ.Advisory.Topic – sent messages that reflect all topic create and destroy events
- ActiveMQ.Advisory.TempQueue – sent messages that reflect all create and destroy events for all temporary queues.
- ActiveMQ.Advisory.TempTopic – sent messages that reflect all create and destroy events for all temporary topics.
- ActiveMQ.Advisory.Expired.Queue – sent messages that reflect all expired messages for all queues

- ActiveMQ.Advisory.Expired.Topic – sent messages that reflect all expired messages for all topics.
- ActiveMQ.Advisory.NoConsumer.Queue – sent messages that reflect all messages sent to all queues that do not have a consumer
- ActiveMQ.Advisory.NoConsumer.Topic – sent messages that reflect all messages sent to all topics that do not have a consumer

The above names can also serve as prefixes that can be assigned qualifiers in order to receive events for a particular destination. The following example illustrates how to subscribe to all create and destroy events for a queue called, "Q.REQ".

```
Topic advisoryTopic = session.createTopic("ActiveMQ.Advisory.Queue.Q.REQ");
```

Or you can use the ".>" wildcard string to receive events for any number of clients and or destinations. For example, to receive all advisory messages, you can wild-card the "ActiveMQ.Advisory" prefix as fallows.

```
Topic advisoryTopic = session.createTopic("ActiveMQ.Advisory.>");
```

10 Administration

10.1 Command Line Tools

The command line tools are activemq and activemq-admin which are located at $ACTIVEMQ_HOME/bin. Activemq starts an ActiveMQ broker and is illustrated in Section 6 Broker Configuration. This section describes activemq-admin. Releases prior to ActiveMQ 5 contained standalone tool scripts, which have been combined into activemq-admin, with the prior script names now passed as task name command line arguments to activemq-admin. These tasks, and their options are described in detail at the following link:

http://activemq.apache.org/activemq-command-line-tools-reference.html

Note that the information at the above URL represents the 4.1.1 standalone utilities, but the detailed information is correct for AMQ 5.

The following summarizes the capabilities of activemq-admin, and provides examples and sample output where appropriate:

Tasks:

start	run an ActiveMQ broker
stop	shutdown an ActiveMQ broker
list	list all running brokers in the specified JMX context
query	query the JMX context for broker statistics and information
bstat	predefined query that displays useful broker statistics
browse	browse the messages of a specific queue

Examples:

start a broker

```
> activemq-admin start xbean:file:$ACTIVEMQ_HOME/conf/dick1.xml&
```

list brokers

```
> activemq-admin list
```

BrokerName = dick1

query broker information

```
> activemq-admin query
```

This command displays detailed information for all MBean types within the JMX context – these include: brokers, destinations (queues, topics, and subscriptions), and connections

```
> activemq-admin query -QQueue=Q.REQ
```

Type = Queue

DispatchCount = 24

Destination = Q.REQ

MaxEnqueueTime = 11850504

QueueSize = 12

Name = Q.REQ

DequeueCount = 24

MemoryPercentageUsed = 0

ConsumerCount = 0

MemoryLimit = 30198988

EnqueueCount = 12

```
MinEnqueueTime = 247613
AverageEnqueueTime = 6049061.708333333
BrokerName = dick1
```

```
> activemq-admin query -QBroker=dick1
```

```
TemporaryQueues = [Ljavax.management.ObjectName;@64dc11
TemporaryTopics = [Ljavax.management.ObjectName;@3eca90
DurableTopicSubscribers = [Ljavax.management.ObjectName;@161d36b
QueueSubscribers = [Ljavax.management.ObjectName;@17f1ba3
InactiveDurableTopicSubscribers = [Ljavax.management.ObjectName;@1d520c4
MemoryLimit = 67108864
Type = Broker
TemporaryTopicSubscribers = [Ljavax.management.ObjectName;@1ef8cf3
BrokerId = ID:linux01-3613-1194543805448-0:0
TotalMessageCount = 12
MemoryPercentageUsed = 0
TotalEnqueueCount = 111
Queues = [Ljavax.management.ObjectName;@1174b07
Topics = [Ljavax.management.ObjectName;@147c5fc
TopicSubscribers = [Ljavax.management.ObjectName;@1ac1fe4
BrokerName = dick1
TotalDequeueCount = 24
TemporaryQueueSubscribers = [Ljavax.management.ObjectName;@ecd7e
StatisticsEnabled = true
TotalConsumerCount = 8
```

browse a queue

```
> activemq-admin browse -amqurl tcp://linux01:61410 Q.REQ
```

```
JMS_HEADER_FIELD:JMSDestination = Q.REQ
JMS_BODY_FIELD:JMSText = Test Message[0] 0.028848685449153866{}
JMS_CUSTOM_FIELD:MessageNumberInt = 0
JMS_HEADER_FIELD:JMSDeliveryMode = persistent
```

JMS_HEADER_FIELD:JMSMessageID = ID:linux01-3619-1194543865208-0:0:1:1:1

JMS_CUSTOM_FIELD:MessageNumberStr = 0

JMS_HEADER_FIELD:JMSExpiration = 0

JMS_HEADER_FIELD:JMSPriority = 4

JMS_HEADER_FIELD:JMSRedelivered = false

JMS_HEADER_FIELD:JMSTimestamp = 1194543866209

bstat

```
> activemq-admin bstat dick1
```

This command prints out statistics for broker “dick1”. The output is similar to the query output, but is only for the requested broker.

shutdown a broker

```
> activemq-admin stop dick1
```

Stopping broker: dick1

10.2 JConsole

Administration of ActiveMQ brokers can be accomplished by means of JMX, the Java Management Extensions. ActiveMQ support of JMX means that a JMX Console, such as the JConsole GUI, can be used to monitor and change an ActiveMQ broker and the JVM in which it runs.

JConsole is included in the JDK, and the executable can be found in the $JAVA_HOME/bin directory. JConsole is capable of connecting to local and/or remote JMX agents running in the JVM of ActiveMQ brokers. Manageable broker attributes and objects are represented by Managed Beans (MBeans). These are displayed in a tree hierarchy on the MBeans tab of the main JConsole window.

10.2.1 Configuration

A combination of system properties, broker properties, and “managementContext” properties are used to configure ActiveMQ administration through JMX. Various properties are used to enable/ disable administration entirely, allow remote administration, require password authentication, specify SSL characteristics for the administrative connection, etc. Fortunately, the default values for most of these properties make it relatively easy to get started with ActiveMQ administration using JConsole.

JMX system properties are specified when the broker startup script is run (for a standalone broker), or when a client and embedded broker are started. The standard activemq startup script sets the SUNJMX environment variable to "-Dcom.sun.management.jmxremote", then uses this value as an argument to the java command when the broker is started. In a similar manner, this system property should be set whenever a client with embedded broker is started, if JMX administration of the broker is desired. Several other JMX system properties exist – these are documented at http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#properties.

Broker properties are specified in the standalone broker's configuration file, or at the end of the URL that starts an embedded broker. The broker property "useJmx" can be used to enable or disable access to the ActiveMQ JMX management. The default value is "true", but this can be changed either in the broker XML configuration file, using

```
<broker brokerName="broker4" useJmx="false"> … </broker>
```

or as a broker option on a URL, such as

```
vm:broker:(vm://localhost)?useJmx="false”
```

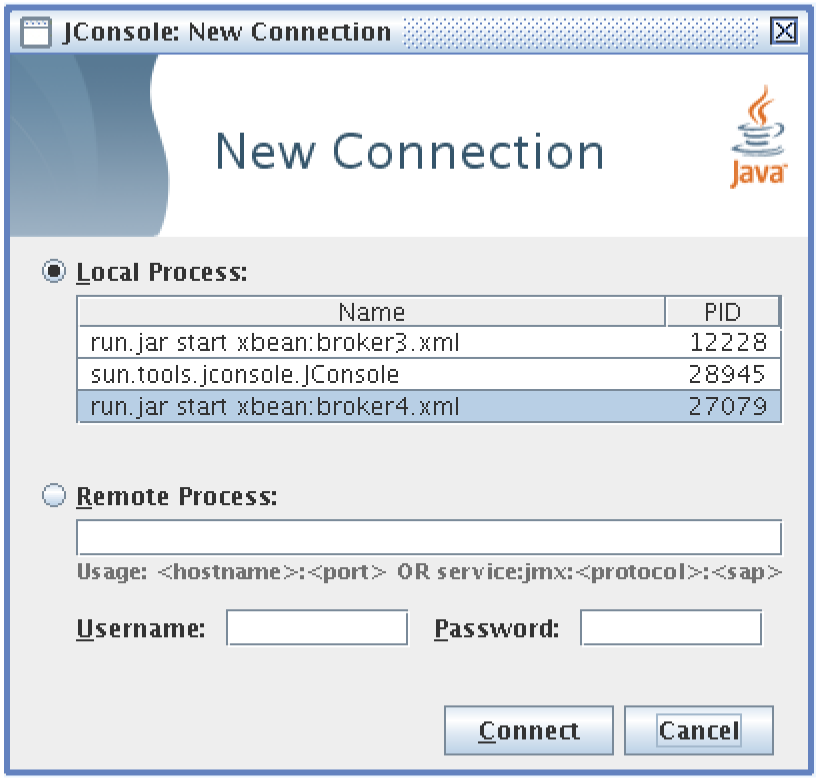
The "managementContext" properties are specified in the <managementContext> elements of the standalone broker's XML configuration file. Here is an example snippet.

```
<!-- Use the following to configure how ActiveMQ is exposed in JMX -->
    <managementContext>
        <managementContext
        connectorPort="1099" jmxDomainName="org.apache.activemq"/>
    </managementContext>
```

Specifying the "connectorPort" attribute in the above way allows JConsole to remotely connect to the JMX agent for JVM and ActiveMQ administration. For multiple brokers running on the same machine, unique port numbers should be assigned. An alternative to specifying the connectorPort attribute is to set the com.sun.management.jmxremote.port property, although this has some security/access implications (see the JMX system properties documentation).

10.2.2 Connection

JConsole can establish either a local or remote connection to a JMX agent. When JConsole first comes up, or when the New Connection menu item is selected, the New Connection dialog box is displayed. An example is shown below.



Name	PID
run.jar start xbean:broker3.xml	12228
sun.tools.jconsole.JConsole	28945
run.jar start xbean:broker4.xml	27079

Under "Local Process:" are listed all the JVM processes that have a JMX agent and are running with the same user id as the JConsole process – including the JConsole process itself. Select a listed process, and the Connect button, to establish the JConsole connection.

If the connectorPort attribute or the com.sun.management.jmxremote.port system property has been configured for a broker, the "Remote Process:" option can be used to establish the administrative connection. Two syntax alternatives are available to specify the host and port, as shown in the dialog box. Here are examples of each:

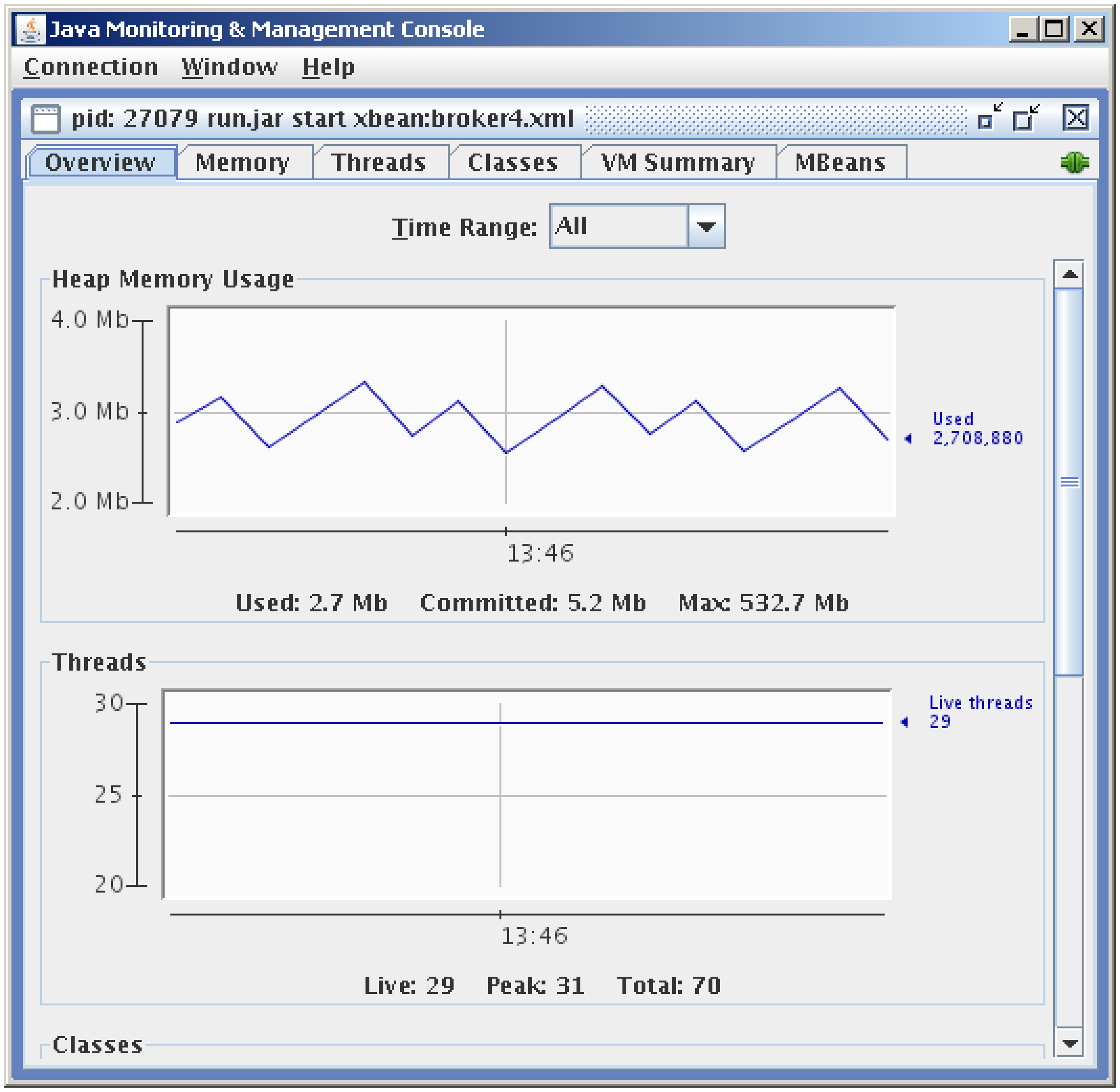
```
linux03:1099
```

```
service:jmx:rmi:///jndi/rmi://linux03:1099/jmxrmi
```

If the connectorPort attribute was used to specify a port, or the system property com.sun.management.jmxremote.authenticate has been set to false, then Username and Password entries need not be provided.

10.2.3 Operation

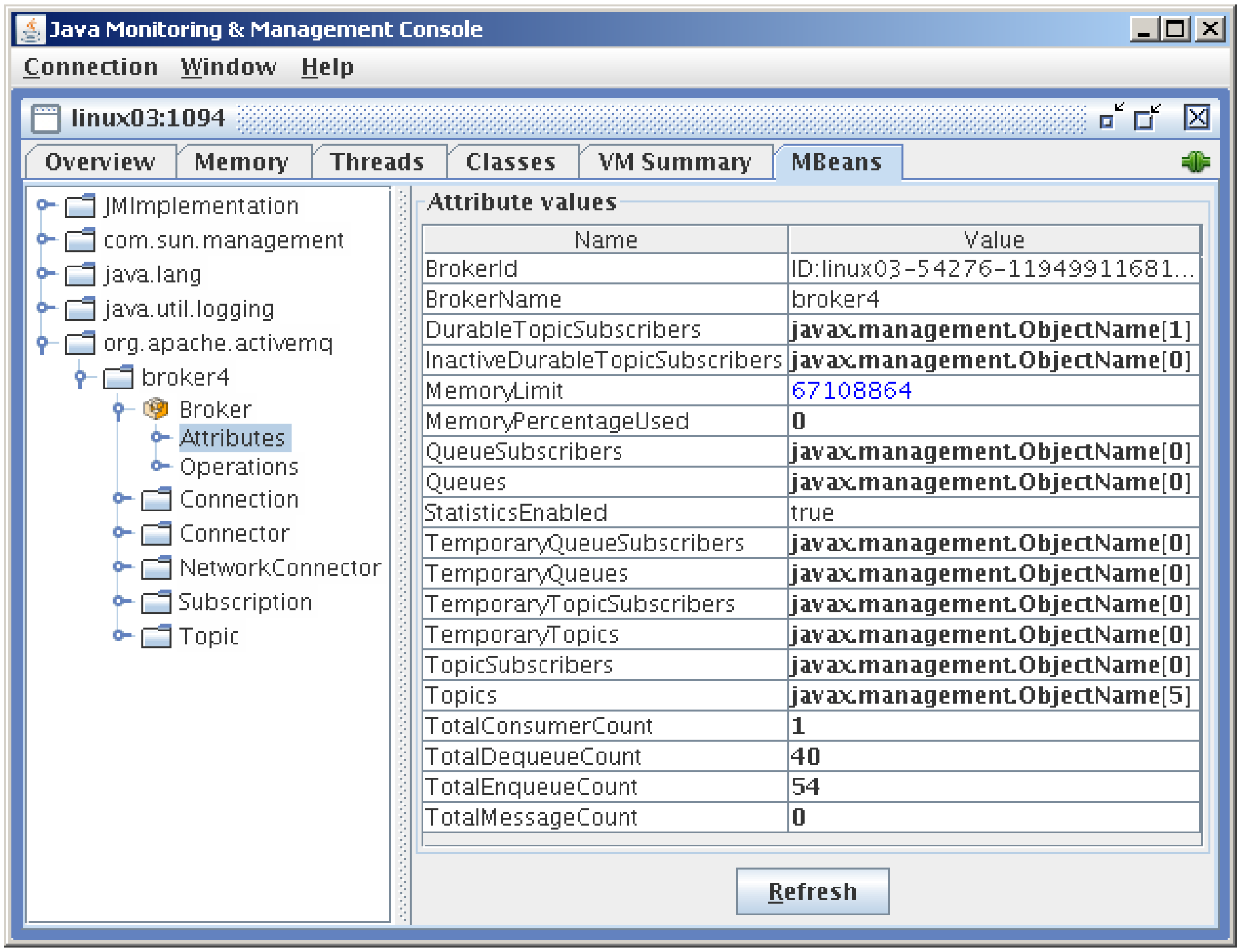
After the administrative connection has been established, the main JConsole window is displayed.



Used: 2.7 Mb Committed: 5.2 Mb Max: 532.7 Mb

Live: 29 Peak: 31 Total: 70

Selecting the MBeans tab and expanding the org.apache.activemq JMX domain name allows the administrator to inspect object properties and statistics, and to perform certain administrative operations. Attributes, whose values exist in arrays, can be accessed by double-clicking on the array name in the Value column. Attributes with values are shown in blue, such as the broker's MemoryLimit in the window shown below, can be changed by simply entering a new value.



Name	Value
BrokerId	ID:linux03-54276-11949911681...
BrokerName	broker4
DurableTopicSubscribers	javax.management.ObjectName[1]
InactiveDurableTopicSubscribers	javax.management.ObjectName[0]
MemoryLimit	67108864
MemoryPercentageUsed	0
QueueSubscribers	javax.management.ObjectName[0]
Queues	javax.management.ObjectName[0]
StatisticsEnabled	true
TemporaryQueueSubscribers	javax.management.ObjectName[0]
TemporaryQueues	javax.management.ObjectName[0]
TemporaryTopicSubscribers	javax.management.ObjectName[0]
TemporaryTopics	javax.management.ObjectName[0]
TopicSubscribers	javax.management.ObjectName[0]
Topics	javax.management.ObjectName[5]
TotalConsumerCount	1
TotalDequeueCount	40
TotalEnqueueCount	54
TotalMessageCount	0

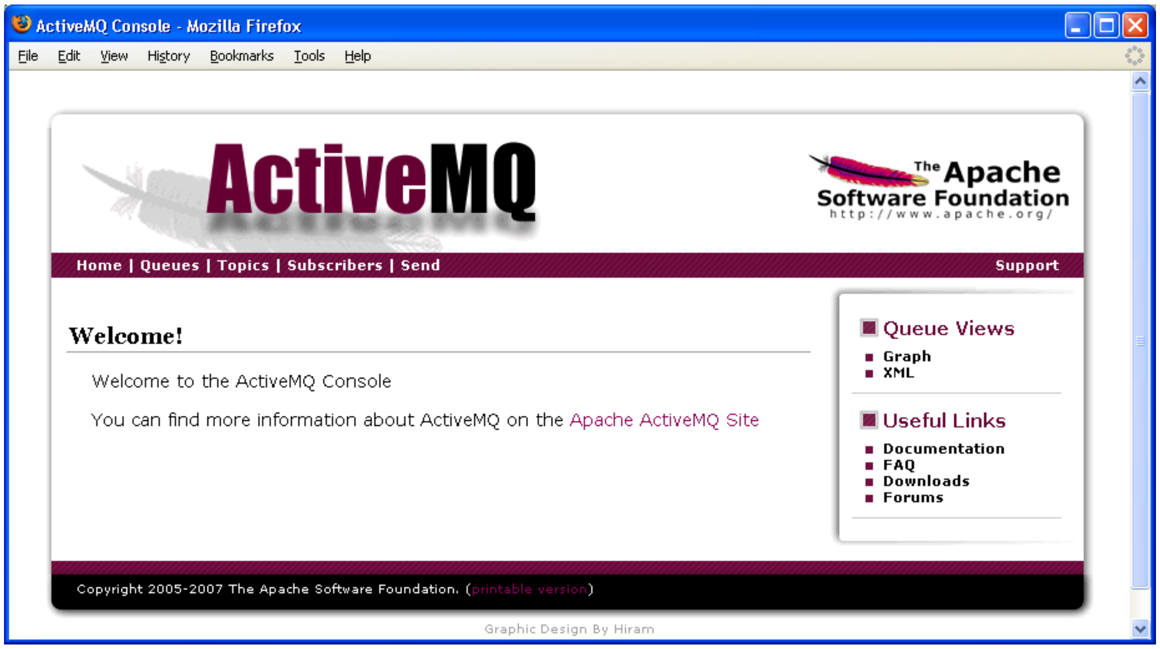
10.3 Web Console

A brief description and configuration information for the ActiveMQ web console is located on the following ActiveMQ web page: http://activemq.apache.org/web-console.html.

By default, the web console runs in the ActiveMQ broker and listens on port 8161. The above link describes how to change this port as well as how to configure the web browser in the jetty web container if you prefer to have the web console run independent of the ActiveMQ broker.

The following are the major features of the web console:

Home page – the menu is on the shaded line below ActiveMQ.

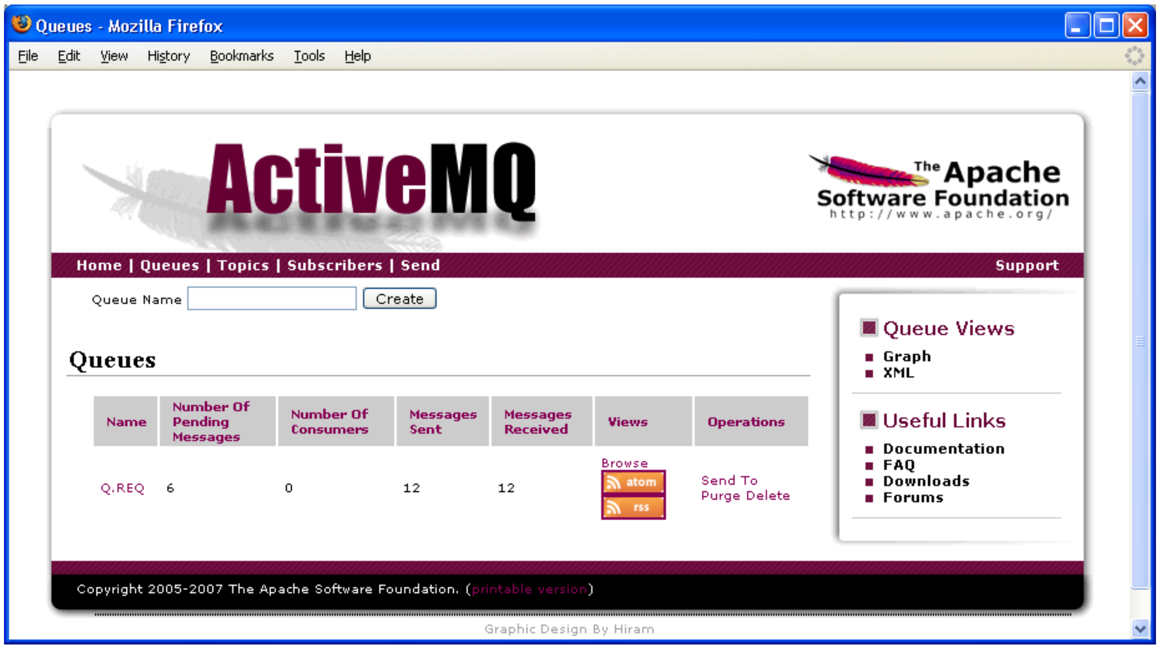

ActiveMQ Console - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support
Queue Views
Graph
XML
Useful Links
Documentation
FAQ
Downloads
Forums
Copyright 2005-2007 The Apache Software Foundation. (printable version)
Graphic Design By Hiram

Welcome!

Welcome to the ActiveMQ Console

You can find more information about ActiveMQ on the Apache ActiveMQ Site

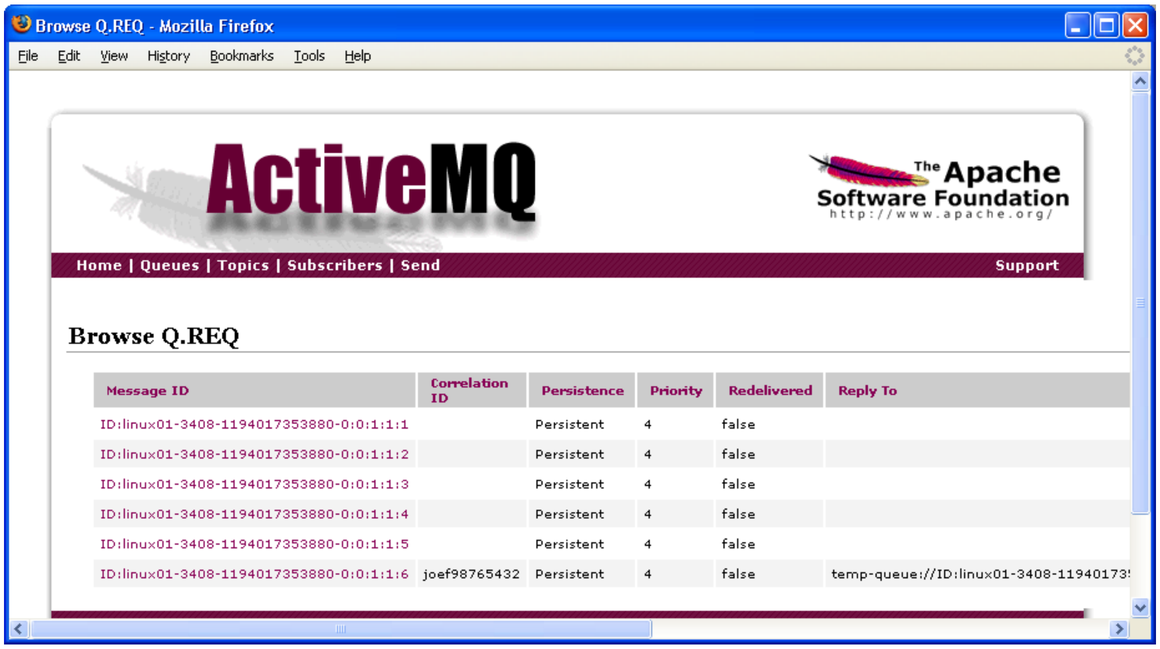
The queue screen displays statistics, as well as send, purge, and delete operations.


Queues - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support
Queue Name
Create
Queue Views
Graph
XML
Useful Links
Documentation
FAQ
Downloads
Forums
Copyright 2005-2007 The Apache Software Foundation. (printable version)
Graphic Design By Hiram

Queues

Name	Number Of Pending Messages	Number Of Consumers	Messages Sent	Messages Received	Views	Operations
Q.REQ	6	0	12	12	Browse atom rss	Send To Purge Delete

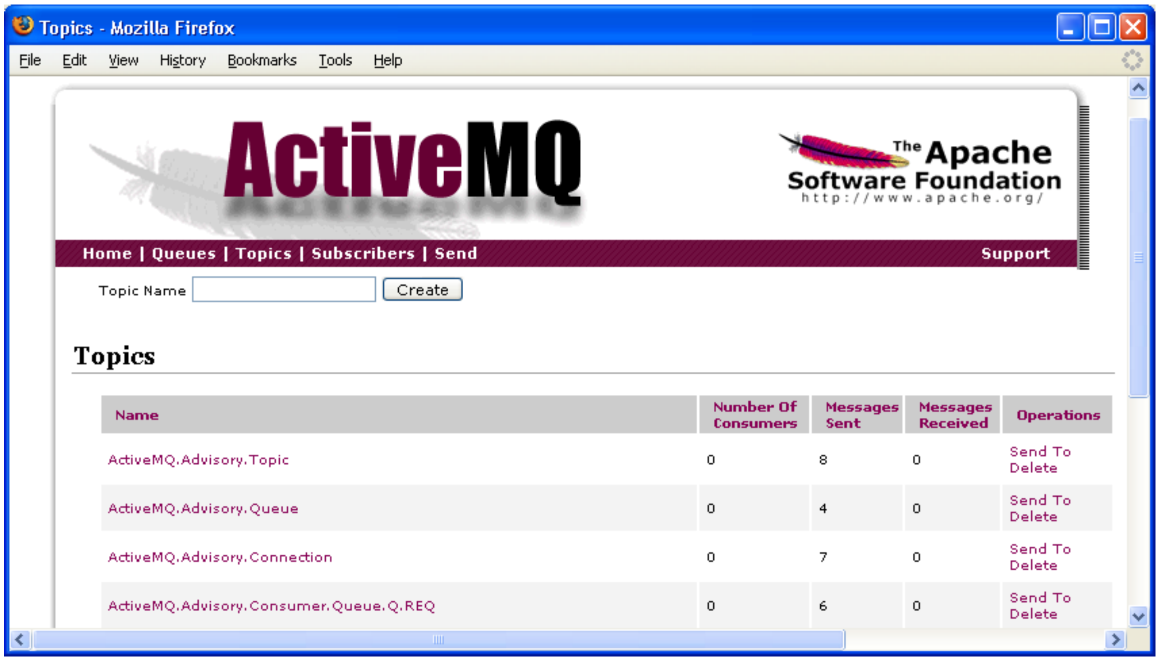
The queue browse screen is displayed by selecting a queue on the Queues screen. This screen displays the messages on the queue.


Browse Q.REQ - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support

Browse Q.REQ

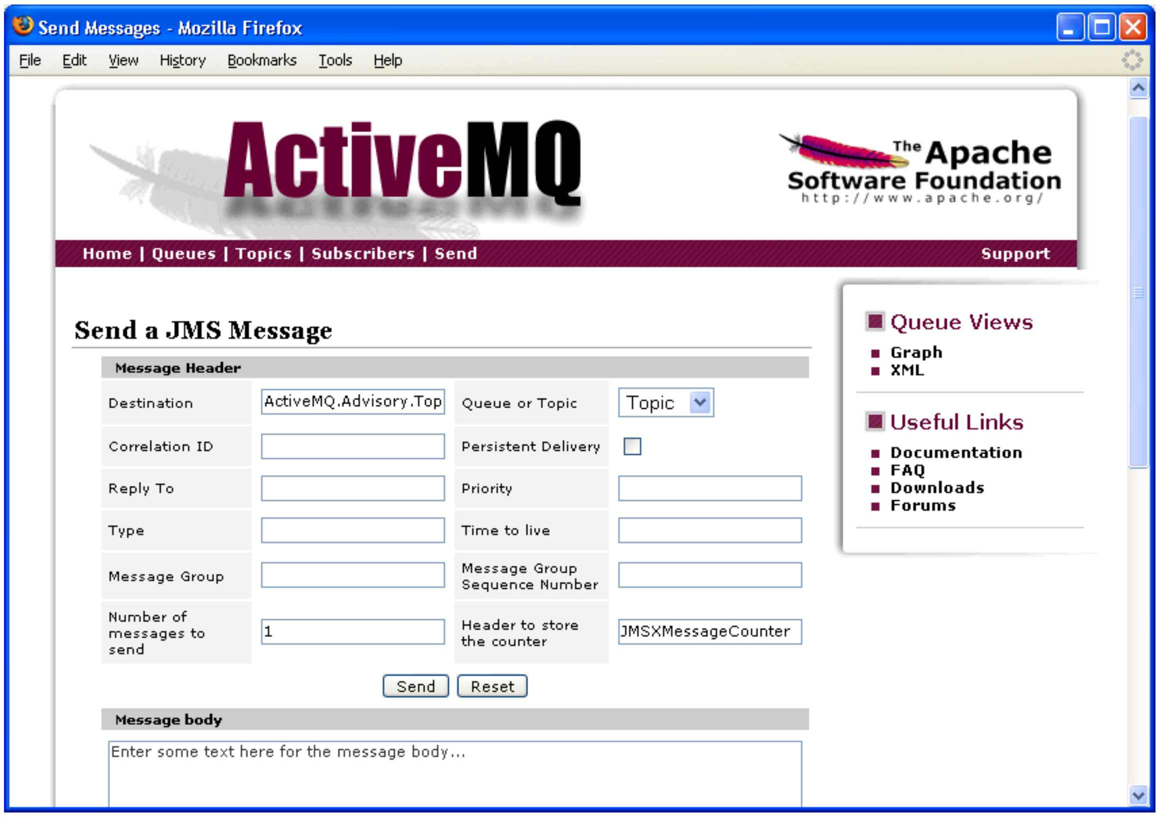
Message ID	Correlation ID	Persistence	Priority	Redelivered	Reply To
ID:linux01-3408-1194017353880-0:0:1:1:1		Persistent	4	false	
ID:linux01-3408-1194017353880-0:0:1:1:2		Persistent	4	false	
ID:linux01-3408-1194017353880-0:0:1:1:3		Persistent	4	false	
ID:linux01-3408-1194017353880-0:0:1:1:4		Persistent	4	false	
ID:linux01-3408-1194017353880-0:0:1:1:5		Persistent	4	false	
ID:linux01-3408-1194017353880-0:0:1:1:6	joef98765432	Persistent	4	false	temp-queue://ID:linux01-3408-11940173

The following screens allow the monitoring and creation of topics, subscriptions, and sending messages to queue/topic destinations.


Topics - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support
Topic Name
Create

Topics

Name	Number Of Consumers	Messages Sent	Messages Received	Operations
ActiveMQ.Advisory.Topic	0	8	0	Send To Delete
ActiveMQ.Advisory.Queue	0	4	0	Send To Delete
ActiveMQ.Advisory.Connection	0	7	0	Send To Delete
ActiveMQ.Advisory.Consumer.Queue.Q.REQ	0	6	0	Send To Delete


Send Messages - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support
Queue Views
Graph
XML
Useful Links
Documentation
FAQ
Downloads
Forums

Send a JMS Message

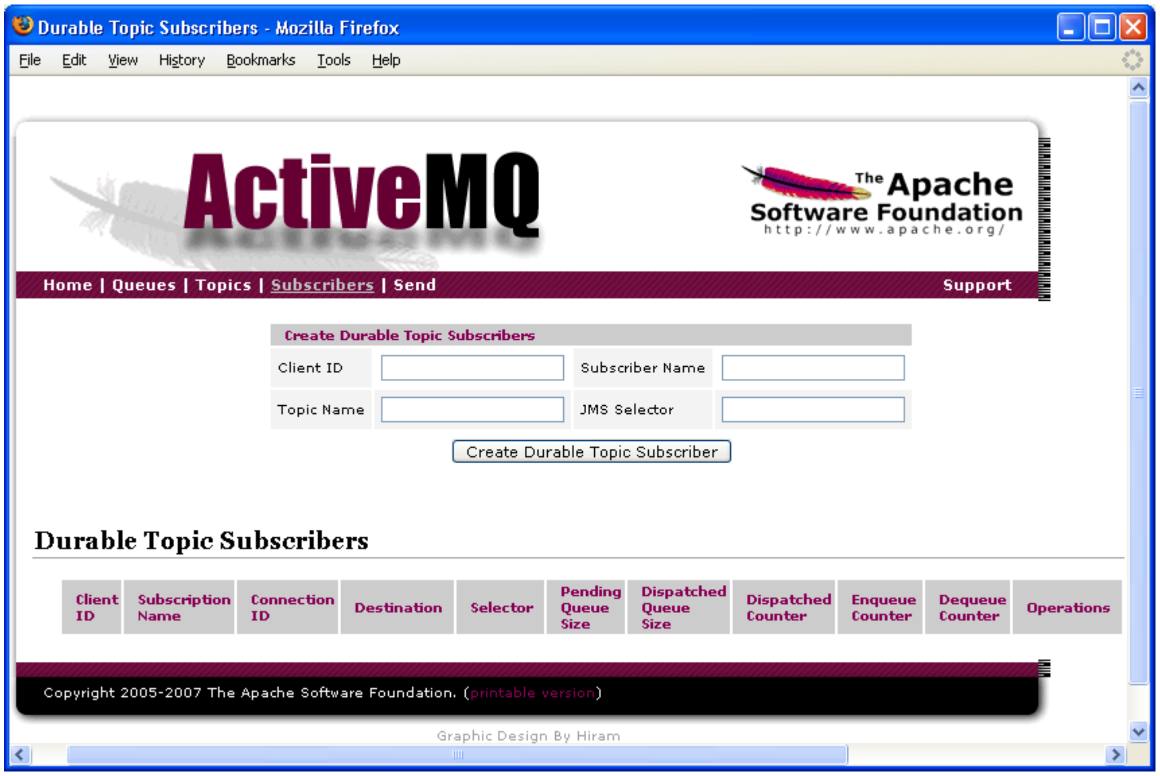
Message Header

Destination	ActiveMQ.Advisory.Top	Queue or Topic	Topic
Correlation ID		Persistent Delivery	☐
Reply To		Priority	
Type		Time to live	
Message Group		Message Group Sequence Number	
Number of messages to send	1	Header to store the counter	JMSXMessageCounter

Send Reset

Message body

Enter some text here for the message body...


Durable Topic Subscribers - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support

Create Durable Topic Subscribers

Client ID		Subscriber Name	
Topic Name		JMS Selector	

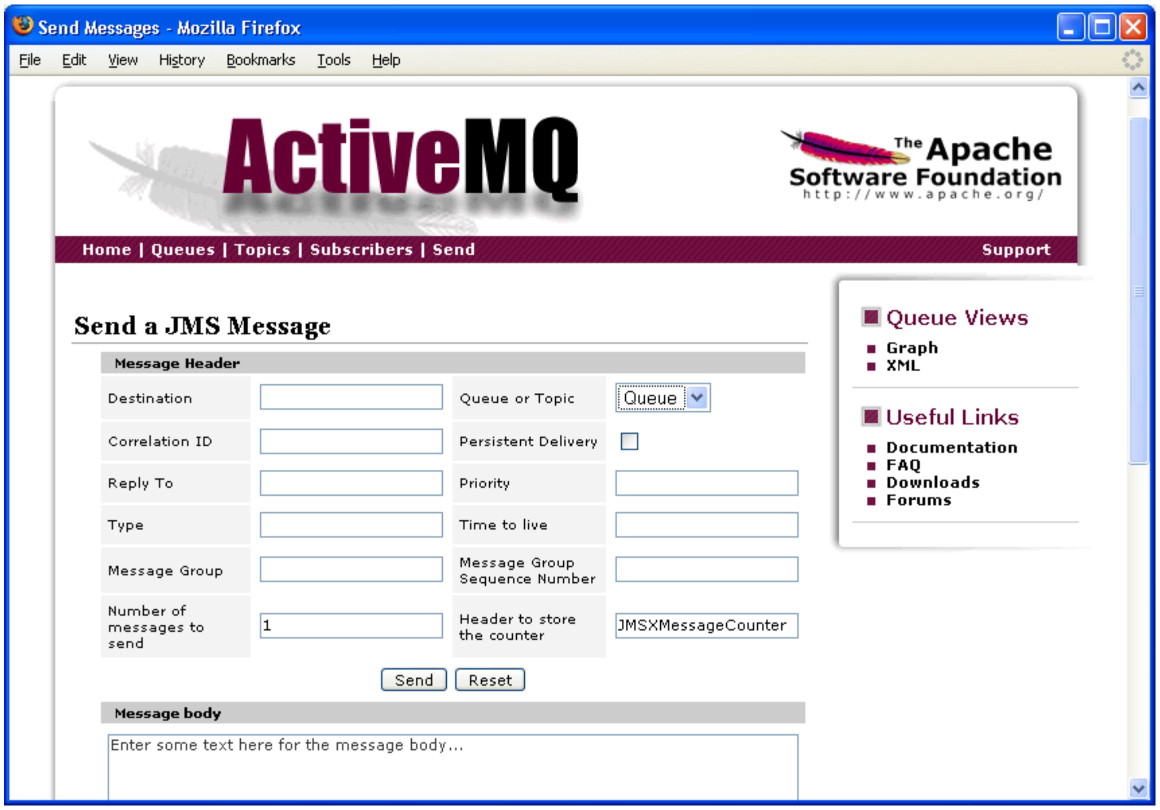
Create Durable Topic Subscriber

Durable Topic Subscribers

Client ID	Subscription Name	Connection ID	Destination	Selector	Pending Queue Size	Dispatched Queue Size	Dispatched Counter	Enqueue Counter	Dequeue Counter	Operations

Copyright 2005-2007 The Apache Software Foundation. (printable version)

Graphic Design By Hiram


Send Messages - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ActiveMQ
The Apache Software Foundation http://www.apache.org/
Home | Queues | Topics | Subscribers | Send
Support
Queue Views
Graph
XML
Useful Links
Documentation
FAQ
Downloads
Forums
Send
Reset

Send a JMS Message

Message Header

Destination		Queue or Topic	Queue
Correlation ID		Persistent Delivery	☐
Reply To		Priority	
Type		Time to live	
Message Group		Message Group Sequence Number	
Number of messages to send	1	Header to store the counter	JMSXMessageCounter

Message body

Enter some text here for the message body...

10.3.1 Securing Access to the Web Console

The following link takes you to a web page that describes how to secure access to the ActiveMQ web console.

http://activemq.apache.org/user-submitted-configurations.html

10.4 DestinationSource

The DestinationSource class, which is found in the org.apache.activemq.advisory package, is an administrative convenience class that can be used by client applications to keep track of the destinations that are available in a broker and allows the client to listen to destinations as they are created and deleted. You can also use DestinationSource to get a listing of queues and topics that are currently available on the broker.

The following code snippet shows you how to acquire a DestinationSource object and get a listing of currently available queues.

```
…
```

```
ActiveMQConnection connection = connectionFactory.createConnection();
Set<ActiveMQQueue> queueSet = amqcon.getDestinationSource().getQueues();
Iterator<ActiveMQQueue> queues = queueSet.iterator();
for(ActiveMQQueue queue: queues )  {
      System.out.println( "Queue: " + queue.getPhysicalName() );
}
…
```

If you download the ActiveMQ source code base, you'll find a test case in …/activemq-core/src/test/java/org/apache/activemq/advisory called, "DestinationListenerTest". The test case shows you how to set up a DestinationListener.

11 Logging

11.1 Commons-logging

Logging refers to the generation of messages that report information and errors as software executes. Logging is useful not only to reveal exception conditions, but also to assist in debugging code, and to help understand how a product functions.

ActiveMQ uses the Apache Software Foundation's (ASF) "commons-logging" package for logging. This org.apache.commons.logging package is a thin API veneer that can be configured to use any logging implementation at runtime.

By default, commons-logging (and therefore ActiveMQ) uses the feature-rich and popular log4j logging implementation, also from the ASF. Another logging implementation that is available through the commons-logging package is the JDK 1.4 (and newer) java.util.logging package.

The User Guide that describes commons-logging development and administration can be found at http://commons.apache.org/logging/guide.html.

11.2 Log4j

Because ActiveMQ uses the commons-logging API described above, the default logging implementation is log4j. This logging package was designed for minimal overhead and optimal logging performance.

Log4j has three main components: Loggers, Appenders, and Layouts. A Logger is a Java object that performs logging. An Appender represents a logging output destination. A Layout specifies the format of the Logger's output.

Loggers have names, and exist in a hierarchy of descendants from a nameless "root logger". A Logger may have a severity level or priority assigned, from the set (descending order) of FATAL, ERROR, WARN, DEBUG, and TRACE. Additional levels OFF (nothing is logged) and ALL (everything is logged) are supported. If a Logger has no level assigned, it inherits the level of its parent Logger.

Appenders exist for the console and regular files, as well as more sophisticated destinations such as a Windows Event Log or a remote socket server. More than one Appender can be attached to a logger.

Layouts can be used to specify any number of pieces of information relevant to the message being logged, such as the format of the current date and time, the source code filename, line number, class name, method name, thread name, and severity level.

The most straightforward way of administering log4j logging in ActiveMQ is through the use of Java properties, for example by using a $ACTIVEMQ_HOME/conf/log4j.properties file. The file that is installed with ActiveMQ begins with these four lines:

```
log4j.rootLogger=INFO, stdout, out

log4j.logger.org.apache.activemq.spring=WARN

log4j.logger.org.springframework=WARN

log4j.logger.org.apache.xbean.spring=WARN
```

The first line establishes the severity level INFO for the root Logger, and assigns two Appenders named "stdout" and "out" to the root Logger. The next three lines assign severity level WARN to the named Loggers.

Various properties can be assigned to any of the named Appenders. For example:

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%-5p %-30.30c{1} - %m%n
```

More information about log4j configuration can be found at http://logging.apache.org/log4j/1.2/manual.html.

11.3 java.util.logging

As an alternative to log4j, ActiveMQ can be configured to use the JDK 1.4 (or newer) java.util.logging package. One advantage provided by java.util.logging is the ability to modify a logging severity level at runtime using an administrative tool such as JConsole.

To configure ActiveMQ for java.util.logging, create a file $ACTIVEMQ_HOME/conf/commons-logging.properties with the following line:

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Jdk14Logger
```

The java.util.logging package uses Logger, Handler, and Formatter classes in a manner analogous to the log4j's Logger, Appender, and Layout. A single LogManager object maintains Logger and Handler information for the Virtual Machine.

In general, java.util.logging Loggers have names, and a hierarchical arrangement. A Logger may have a severity level or priority assigned, from the set (descending order) of SEVERE, WARNING, INFO, CONFIG, FINE, FINER, and FINEST. As with log4j, additional levels OFF (nothing is logged) and ALL (everything is logged) are supported, and if a Logger has no level assigned, it inherits the level of its parent Logger.

By default, the java.util.logging LogManager gets configuration information from the file $JAVA_ROOT/lib/logging.properties. To establish a separate properties file for use with ActiveMQ, you would define a system property to give java.util.logging.config.file a value. For example, on the "java" command line that starts a broker, you could insert:

```
-Djava.util.logging.config.file="${ACTIVEMQ_BASE}"/conf/logging.properties
```

You can use this properties file to register and configure Handlers and Formatters, and to assign severity levels. For examples, these two lines register a FileHandler and a ConsoleHandler with the LogManager, and establish a severity level of INFO for all Loggers:

```
handlers = java.util.logging.FileHandler,java.util.logging.ConsoleHandler

.level=INFO
```

With a more specific reference to the full name of a Logger, you can fine-tune severity levels:

```
org.apache.activemq.spring.level=WARNING

org.springframework.level=WARNING

org.apache.xbean.spring.level=WARNING

org.apache.activemq.level=FINEST
```

Several Handler properties can be set, for example the Formatter to be used:

```
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

Other Handler properties can be used to specify values such as the pathname of a log file, a size limit for the file, and a severity level to apply specifically to the Handler. Further information is available in the JDK documentation, and in http://java.sun.com/javase/6/docs/technotes/guides/logging/overview.html.

11.4 Controlling Logging with JConsole

If you have configured ActiveMQ to use java.util.logging, then the Java Monitoring and Management Console (JConsole) can be used to change Logger severity levels at runtime.

Start by using JConsole's New Connection dialog box to connect to the Java Virtual Machine.

Select the MBeans tab, and expand the tree widget to expose java.util.logging/Logging/Attributes.

Double-clicking on the LoggerNames Value will cause a list of Loggers to be displayed. If you take any Logger's full name, and go to the java.util.logging/Logging/Operations page, you will be able to invoke the getLoggerLevel and setLoggerLevel methods of that Logger object.

11.5 Client Logging

To control logging for an ActiveMQ Java client, ensure that the following files are in the client's class path.

- log4j.properties – change the logging level via the 'log4j.rootLogger' statement in this file
- common-logging-<*version number*>.jar
- log4j-<*version number*>.jar

12 Destination/Consumer Options

The configuration options that ActiveMQ makes available for assigning to destinations and/or consumers allow you to extend their functionality without having to extend the JMS API. These options are assigned through either a connector URI or destination. For example, the following jndi.properties file will assign a priority level of 10 to any consumer that associates itself with the destination called "TEST.Q".

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616
queue.TEST.Q = TEST.Q?consumer.priority=10
```

And this jndi.properties file will assign a prefetch limit to all destinations that are derived from the connector URI that has been assigned to the 'local' connection factory. Recall that with in the JMS, destinations are assigned to sessions and sessions are derived from connections.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616?jms.prefetchPolicy.all=50
queue.Q.REQ = Q.REQ
```

Note from the two examples above that the 'consumer' prefix is used when assigning the option through a destination, while the 'jms' prefix is used when assigning the option through a connector URI. The following table lists and describes briefly all the options that can be assigned to a destination/consumer.

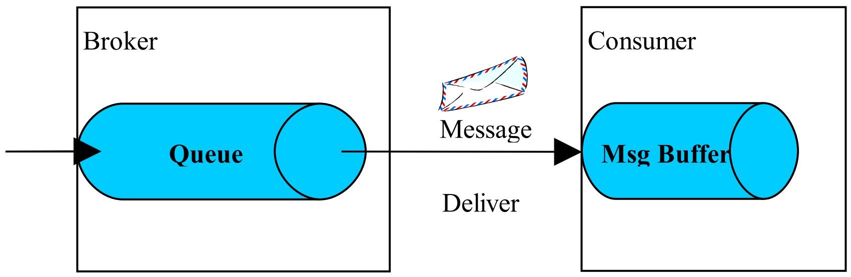
Option Name	Default Value	Description
prefetchSize	variable	The number of messages the consumer is capable of pre-fetching.
maximumPendingMessageLimit	0	Used for setting the max number of non-persistent, pending messages that can be accumulated by the broker.
noLocal	false	In some cases, a client's connection may both produce to and consumer from the same destination. When set to 'true', the noLocal attribute will prohibit delivery of messages produced by the client's own connection.
dispatchAsync	false	Should the broker asynchronously dispatch messages to the consumer?
retroactive	false	Is this a retroactive consumer?
selector	null	JMS Selector used with the consumer.

exclusive	false	Specifies whether or not the consumer will be treated as an exclusive consumer.
priority	0	Sets the priority level associated with the consumer. The lower the value the higher the priority.

The following sections will describe in more detail some of the options listed in the table above.

12.1 Prefetch Limits

To achieve a high level of message throughput, the ActiveMQ message broker dispatches or pushes messages to a consumer as fast as possible so that the consumer always has messages in its local memory (message buffer) to process. This is opposed to the consumer having to pull each individual message from the broker, which adds a significant amount of overhead or latency per message.



Consumer’s Message Buffer

However, this method of pushing as many messages as possible on to the consumer must incorporate a throttling mechanism. This is because it typically takes much less time for the broker to deliver a message than it does for the consumer to process it; therefore, the consumer could very easily be overwhelmed with messages. This would be the case if, for example, the consumer must access a DB as part of the message processing. The throttling mechanism that ActiveMQ incorporates is referred to as the “*prefetch limit*”. This limit specifies how many messages the broker can push to the consumer without first getting an acknowledgement back from the consumer for a message it has processed. When the prefetch limit has been reached, the broker will stop delivering messages to the consumer until the consumer starts to acknowledge processed messages.

With the prefetch set to 0, consumers pull messages from the queue when they are ready. In other words, the broker will not dispatch the message to the consumer until the consumer asks for via the receive method.

These are the default “prefetch limit” values assigned to the various destination types.

- queues: 1000

- persistent/durable topics: 100
- non-persistent topics: Short.MAX_VALUE -1 (32766)
- queue browsers: 500

The default values can be overridden by assigning a ‘jms.prefetchPolicy’ parameter to the connector URI that is used by the client for establishing a connection to the broker. The following lists and describes the different jms.prefetchPolicy parameters.

Parameter Name	Description
jms.prefetchPolicy.all	Sets the prefetch size for all consumer types
jms.prefetchPolicy.queuePrefetch	Sets the prefetch size for queue consumers
jms.prefetchPolicy.topicPrefetch	Sets the prefetch size for topic consumers
jms.prefetchPolicy.durableTopicPrefetch	Sets the prefetch size for durable topic consumers
jms.prefetchPolicy.queueBrowserPrefetch	Sets the prefetch size for queue browsers

For example, the connector URI depicted in this jndi.properties file will set the prefetch size for all consumer types.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616?jms.prefetchPolicy.all=50
queue.Q.REQ = Q.REQ
```

Keep in mind that the client can derive one or more JMS sessions from a connection to the broker and that these sessions can be assigned different destinations.

The connector URI depicted in this jndi.properties file will set the prefetch size for just queue consumers.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616?jms.prefetchPolicy. queuePrefetch=1
queue.Q.REQ = Q.REQ
```

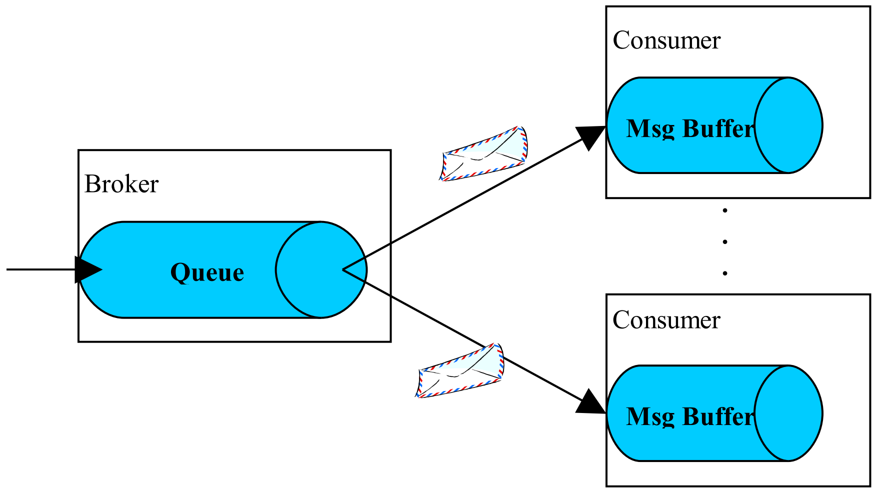
The prefetch size can also be configured on a per destination basis, instead of a connection basis. This is done by assigning the “consumer.prefetchSize” option to the destination as depicted in this jndi.properties file.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
```

```
connection.local.brokerURL = tcp://linux02:61616
queue.Q.REQ = Q.REQ?consumer.prefetchSize=10
```

12.2 Slow Consumers

Consumers that do not process messages from their local message buffer in a timely manner are referred to as "slow consumers". With respect to non-persistent messages, slow consumers can cause message back-flow problems on the broker. For example, if a slow consumer's prefetch limit has been reached, the broker cannot dispatch any more messages to that consumer; therefore, the messages for that consumer may accumulate in the broker's local memory. With messages accumulating in the broker's local memory, you run the risk of compromising producers and other faster consumers that are using the services of the broker. Since these pending messages are non-persistent, the broker cannot write them out to secondary storage to free up local memory. One way of mitigating this possible issue is to configure a cluster of consumers that read off the same destination.



Consumer Cluster

However, clustering consumers may not always be possible as would be the case if; for example, there exists a slow *exclusive* consumer (See section 12.5). Or there may exist the situation where all consumers in the cluster slow down due to a slow DB that they are all accessing.

ActiveMQ allows you to configure the maximum number of pending, non-persistent messages that the broker will accumulate for a slow consumer (i.e., in addition to that consumer's prefetch limit). When this maximum number is reached, older messages are discarded thus making room for newer messages. Configuring this functionality is done through the "maximumPendingMessageLimit" destination option. The following table lists and describes the possible values for this option.

Possible Values For maximumPendingMessageLimit	**Description**
0	A value of zero for the limit specifies that the broker should not keep any pending messages in memory other than the consumer's prefetch limit amount.
> 0	A value greater than zero will accumulate pending messages up to the amount specified. When this value is breached, older messages will be discarded to make room for new messages.
-1	A value of -1 disables the discarding of messages

This jndi.properties file illustrates how the pending message limit size is assigned to a consumer that associates itself with the Q.REQ destination.

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616
queue.Q.REQ = Q.REQ?consumer.maximumPendingMessageLimit =10
```

Another configuration approach taken is to configure the pending message limits at the broker. This allows you to centralize configuration at the broker level instead of having to configure each and every client. You can configure the broker to follow one of two pending message limit strategies on a per destination basis. These two strategies are called the "constantPendingMessageLimitStrategy" and the "prefetchRatePendingMessageLimitStrategy".

ConstantPendingMessageLimitStrategy

This strategy is similar to setting the maximumPendingMessageLimit option to a value greater than zero. The example below is a snippet taken from a broker's XML configuration file that illustrates how to set this strategy for all queues starting with the prefix "FOO"

```
<broker brokerName="slave" xmlns="http://activemq.org/config/1.0">
…
<destinationPolicy>
```

```
      <policyMap>
           <policyEntries>


            <policyEntry queue="FOO.>">
               <!-- lets force old messages to be discarded
                           for slow consumers -->
               <pendingMessageLimitStrategy>
                  <constantPendingMessageLimitStrategy limit="30"/>
               </pendingMessageLimitStrategy>
          </policyEntry>

         </policyEntries>
     </policyMap>
  </destinationPolicy>

</broker>
```

With the above example in-place, all consumers associated with queues starting with the prefix "FOO" will be assigned a constantPendingMessageLimitStrategy with a limit of 30.

PrefetchRatePendingMessageLimitStrategy

This strategy will calculate the maximum number of pending messages using a specified multiplier of the consumer's prefetch size. So with this strategy, you can for example specify to keep around 5.5 times the prefetch count for each consumer. Here's an example

```
<broker brokerName="slave" xmlns="http://activemq.org/config/1.0">
…

<destinationPolicy>
      <policyMap>
           <policyEntries>


            <policyEntry queue="FOO.>">
               <!-- lets force old messages to be discarded
                              for slow consumers -->
               <pendingMessageLimitStrategy>
                   <prefetchRatePendingMessageLimitStrategy multiplier="5.5"/>
               </pendingMessageLimitStrategy>
          </policyEntry>

         </policyEntries>
     </policyMap>
  </destinationPolicy>

</broker>
```

You can also influence the strategy that the broker will use to discard or evict pending messages that result from slow consumers. The default eviction strategy is referred to as, "oldestMessageEvictionStrategy", which specifies that the oldest messages are to be evicted first.

A second eviction strategy is called, “oldestMessageWithLowestPriorityEvictionStrategy”. This strategy states that the pending message with the lowest priority will be evicted first, which allows you to evict the lower priority pending messages first even if they are newer.

The example broker XML configuration file below illustrates how to setup a pending message limit strategy of prefetchRatePendingMessageLimitStrategy in combination with an eviction strategy of oldestMessageEvictionWithLowestPriorityStrategy for all queue destinations having a prefix of “FOO”.

```
<broker brokerName="slave" xmlns="http://activemq.org/config/1.0">
…

<destinationPolicy>
      <policyMap>
           <policyEntries>

             <policyEntry queue="FOO.>">

               <pendingMessageLimitStrategy>
                   <prefetchRatePendingMessageLimitStrategy multiplier="5.5"/>
               </pendingMessageLimitStrategy>
                <messageEvictionStrategy>
                    <oldestMessageEvictionWithLowestPriorityStrategy/>
               </messageEvictionStrategy >

            </policyEntry>

         </policyEntries>
     </policyMap>
  </destinationPolicy>

</broker>
```

From here on things need work, because of the high water mark thing. Does the high water mark override the pending message limit strategy??? Do we just ignore it?

This example broker XML configuration file illustrates how to set the oldestMessageEvictionWithLowestPriorityStrategy for all queue destinations having a prefix of “FOO”. Again, note the high water mark is set to 50.

```
<broker brokerName="slave" xmlns="http://activemq.org/config/1.0">
…

<destinationPolicy>
      <policyMap>
           <policyEntries>

             <policyEntry queue="FOO.>">
```

```
               <messageEvictionStrategy>
                 <oldestMessageEvictionWithLowestPriorityStrategy
                              evictExpiredMessagesHighWatermark=50 />
               </messageEvictionStrategy >
            </policyEntry>

         </policyEntries>
     </policyMap>
  </destinationPolicy>

</broker>
```

Other approaches for dealing with slow consumers are "Asynchronous Dispatch" (see section 12.3) and "Message Cursors" (see section 13.11).

12.3 Asynchronous Dispatch

See the following web page for an explanation of the dispatchAsync option.

http://open.iona.com/wiki/display/ProdInfo/Understanding+the+Threads+Allocated+in+ActiveMQ

By default, the dispatchAsync option is set to true; therefore, the dispatch thread within the broker is used.

You can configure asynchronous dispatching at either the connection or destination. Here are some examples as they would appear in a jndi.properties file.

```
# You can assign dispatchAsync to a connection factory either indirectly via the brokerURL
# or directly to the factory

connection.<factory name>.brokerURL= tcp://localhost:61616?jms.dispatchAsync=true

connection.<factory name>.dispatchAsync=true

# Or you can assign it to a destination

queue.TEST.Q = TEST.Q?consumer.dispatchAsync=true
```

12.4 Retroactive Consumers

A retroactive consumer is one that receives messages from a Topic that has been configured to make every attempt to go back in time and re-dispatch any messages that the consumer may have missed. Missed messages may occur if the consumer is connecting to the broker for the first time, or may have gotten disconnected from the broker and is re-connecting.

The retroactive consumer feature is not guaranteed. For various reasons, messages the consumer has missed may have been evicted and thus will no longer be available.

It is very likely that retroactive consumers will receive duplicate messages. For example, if a retroactive consumer re-connects to a broker, it may receive the same missed messages that it was given during its previous session with the broker.

You configure a Topic to be retroactive by assigning it an attribute called, “consumer.retroactive” and assigning that attribute a value of ‘true’ as this example jndi.properties file illustrates.

```
# Set up the connection factory
connectionFactoryNames = localConnectionFactory
connection.localConnectionFactory.brokerURL = tcp://localhost:61616

# Create a test topic and make it retroactive
topic.TEST.TOPIC = TEST.TOPIC?consumer.retroactive=true
```

Through its XML configuration file, the message broker can be given one of a number of different “*subscription recovery policies*“. These policies mandate how the broker is to go back in time and retrieve messages for retroactive consumers. The broker XML configuration snippet below, illustrates one such policy called, “fixedCountSubscriptionRecoveryPolicy“. In the snippet, all topics that start with the “TEST.” prefix will be assigned this policy.

```
<broker>
…
    <destinationPolicy>
      <policyMap>
        <policyEntries>
           <policyEntry topic="TEST.>">
            <subscriptionRecoveryPolicy>
                 <fixedCountSubscriptionRecoveryPolicy maximumSize="10" />
            </subscriptionRecoveryPolicy>
          </policyEntry>
        </policyEntries>
      </policyMap>
    </destinationPolicy>
…
</broker>
```

The following lists and describes all of the recovery subscription policies that can be assigned to Topics.

<u>fixedCountSubscriptionRecoveryPolicy</u>

This policy mandates that the broker is to re-dispatch a fixed count of missed messages. The fixed count is configured through the fixedCountSubscriptionRecoveryPolicy’s maximumSize attribute. In the example above, the broker will re-dispatch up to the last 10 messages that the consumer has missed.

fixedSizedSubscriptionRecoveryPolicy

This policy mandates that the broker limit the amount of memory used for storing missed messages. The amount of memory is configured through the fixedSizedSubscriptionRecoveryPolicy's maximumSize attribute.

```
< fixedSizedSubscriptionRecoveryPolicy maximumSize=”1024” />
```

This is the default recovery subscripotion policy for all Topics and the default setting for maximumSize is “100 * 64 * 1024” or 6,553,600 bytes.

noSubscriptionRecoveryPolicy

This policy disables subscription recovery for the given destinations(s).

lastImageSubscriptionRecoveryPolicy

This policy will only return the first or oldest message to be missed.

queryBasedSubscriptionRecoveryPolicy

This policy re-dispatches messages based on a specific message selector. For example, the setting below will re-dispatch only those messages whose type is 'car' and color is blue and weight greater than 2500 pounds.

```
<queryBasedSubscriptionRecoveryPolicy query="JMSType = 'car' AND color =
'blue' AND weight >2500" />
```

timedSubscriptionRecoveryPolicy

This policy retains missed messages based on an expiry time. For example, the setting below will maintain missed messages for 60 seconds. The time unit used for recoverDuration is in milliseconds.

```
<timedSubscriptionRecoveryPolicy recoverDuration="60000" />
```

12.5 Exclusive Consumers

ActiveMQ dispatches messages from a queue in the same sequential order that they arrived in the queue and at times, it is important for a consumer to process messages in the same order that they arrived in the queue. However, if you have a cluster of consumers concurrently reading from a queue, you will lose the ability to process the messages in the same order that they arrived at the queue. This is because ActiveMQ's default behavior is to balance the queue's message load across the cluster of consumers; therefore, the messages will be processed concurrently by different consumers.

Starting with ActiveMQ 4.x and higher, a new feature has been introduced called, "Exclusive Consumer" or "Exclusive Queues". With this feature enabled, if there is a cluster of consumers reading off a particular queue, the broker picks a consumer in the cluster to consume all messages in a queue. This ensures that all messages in the queue are processed in the same order that they arrived. In other words, ActiveMQ will single-thread all of the queue's messages through one consumer. If the chosen consumer fails, then the broker will automatically choose another consumer in the cluster. So in effect, what you have is a master/slave cluster configuration for consumers. One consumer in the cluster is chosen as the master to receive all messages and if the master fails, a new master is chosen from the cluster.

To create an exclusive consumer, you assign the "consumer.exclusive=true" parameter to the destination's entry as it is defined in the JNDI name space. The following example illustrates how this is done in the jndi.properties file.

```
########################################################################
#
# This is the jndi.properties file, which that is used to
# define/configure one or more administered objects (connection
# factory and destination) for the ActiveMQ JMS client.
#
# This file must be placed in the application's CLASSPATH in order to have
# the JVM load it into the default InitialContext.

java.naming.factory.initial =
org.apache.activemq.jndi.ActiveMQInitialContextFactory

connectionFactoryNames = local

connection.local.brokerURL = tcp://localhost:61616

queue.TestQ = TestQ?consumer.exclusive=true
```

12.6 Dead Letter Queue (DLQ)

A dead letter queue, which can also be a topic, is used to hold messages that could not be delivered. For example, the message expired or its redelivery count was exceeded.

12.6.1 individualDeadLetterStrategy

By default, the DLQ is a queue (it can be configured to be a topic) and is named, "ActiveMQ.DLQ". All undelivered messages will, by default, be placed in the ActiveMQ.DLQ queue. However, you can alter the default behavior via a <*policyEntry*> element in the broker's XML configuration file. The following examples of a <policyEntry>, assigns a DLQ by the name of DLQ.Q.Test and DLQ.Topic.Test for individual destinations called Q.Test and Topic.Test, respectively. Undelivered messages from Q.Test and Topic.Test will therefore be placed in DLQ.Q.Test and DLQ.Topic.Test, respectively.

```
<beans>
 <broker …>

   • • •

   <destinationPolicy>
     <policyMap>
       <policyEntries>

         <policyEntry queue="Q.TEST">
           <deadLetterStrategy>
             <individualDeadLetterStrategy  queuePrefix="DLQ." />
           </deadLetterStrategy>
         </policyEntry>

        <policyEntry topic="Topic.TEST">
           <deadLetterStrategy>
             <individualDeadLetterStrategy  queuePrefix="DLQ." />
           </deadLetterStrategy>
         </policyEntry>

       </policyEntries>
     </policyMap>
   </destinationPolicy>
• • •

</broker>
</beans>
```

This table describes the attributes for the <individualDeadLetterStrategy> element.

Attributes For <individualDeadLetterStrategy>	Default Value	Description
queuePrefix	ActiveMQ.DLQ.Queue.	Sets the prefix to use for all dead letter queues for queue messages
topicPrefix	ActiveMQ.DLQ.Topic.	Sets the prefix to use the dead letter queues for topic messages
useQueueForQueueMessages	true	Sets whether a queue or topic should be used for **queue** messages. If you set this to 'false', you would end up with a topic having the following prefix, "ActiveMQ.DLQ.Queue."
useQueueForTopicMessages	true	Sets whether a queue or topic should be used for **topic** messages. If you set this to false, you' will end up having a topic with the following prefix, "ActiveMQ.DLQ.Topic.".
processExpired	true	If set to false, expired messages will not be placed in the DLQ.
processNonPersistent	true	If set to false and the message is non-persistent, then the message will not be placed in the DLQ.

12.6.2 sharedDeadLetterStrategy

The <sharedDeadLetterStrategy> element can be used for setting the system's default ActiveMQ.DLQ processExpired and processNonPersistent attributes. For example, this policyEntry will result in expired messages **not** being placed in the default system's ActiveMQ.DLQ.

```
<policyEntry queue=">" >
   <deadLetterStrategy>
```

```
            <sharedDeadLetterStrategy processExpired="false"/>
      </deadLetterStrategy>
</policyEntry>
```

12.6.3 DiscardingDLQBroker

This is a plugin that allows one to configure queues and topics, all or matched based on regular expressions, to drop messages that are being sent to the DLQ.

This plugin,, which is only available in ActiveMQ 5.2+, is useful when one uses constant pending message limit strategy or the other eviction rules, but doesn't want to incur the overhead of yet another consumer to clear the DLQ.

This is an example configuration.

```
<plugins>
<bean xmlns=http://www.springframework.org/schema/beans id="discardingDlqBroker"
class="org.apache.activemq.plugin.DiscardingDLQBroker">

 <property name="dropAll" value="true"/>
 <property name="dropTemporaryTopics" value="true"/>
 <property name="dropTemporaryQueues" value="true"/>

<!--drops by destination name, using java regular expressions
 http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html
delimited by spaces, so destination names cannot contain spaces
 <property name="dropOnly" value="topic_1 queue_1"/>
-->

<!-- Used to specify how often a message gets logged indicating the number of messages that
have been dropped. -->
<property name="reportInterval" value="1000"/>
</bean>
</plugins>
```

13 Advanced Features

13.1 Asynchronous Sends

When your producer sends a *persistent* message to the ActiveMQ message broker, the corresponding Producer.send() method will block until the broker sends the producer back an acknowledgement (ack), which indicates that the broker has successfully placed the message in the target destination and has also written the message out to secondary storage. This is sometimes referred to as a *synchronous* send. The one exception is if your producer is sending persistent messages as part of a transaction. In this case it is the commit() method that will be synchronous (i.e., blocks), and not the sends that are part of the transaction. When the commit() method successfully completes, it is an indication that all the persistent messages have been successfully written out to secondary storage.

This synchronous mode of sending persistent messages guarantees a high-level of reliability; however, it also introduces a latency penalty, because the application or thread of execution must block until it receives the 'ack' from the message broker. If your application has been designed to tolerate a few lost messages in failure scenarios, then you can disable synchronous sends and enable *asynchronous* sends. Enabling asynchronous sends will preclude the Producer. send() method from blocking or waiting on the 'ack' from the broker thus eliminating the latency penalty associated with synchronous sends. To enable asynchronous sends, set the 'useAsyncSend' URI connection property to 'true'. This is an example of setting it for a tcp connector URI.

```
tcp://localhost:61616?jms.useAsyncSend=true
```

Or you can use several methods for assigning it to a connection factory in your jndi.properties file as follows

```
connection.<factory name>.brokerURL= tcp://localhost:61616?jms.useAsyncSend=true

connection.<factory name>.useAsyncSend = true
```

13.1.1 Flow Control

Starting with version 5.0 of ActiveMQ, you can control the flow of asynchronous sends. In other words, you can control the maximum amount of message data that an asynchronous producer can transmit to the message broker prior to receiving an acknowledgment from the broker that it has accepted the previously sent messages. By default, this flow control is disabled for producers that have enabled asynchronous sends; however, you can enable flow control for asynchronous sends by assigning the 'producerWindowSize' property to the asynchronous producer's connector URI as depicted in this example.

```
tcp://localhost:61616?jms.useAsyncSend=true&jms.producerWindowSize=1024000
```

Or you can use several methods for assigning it to a connection factory in your jndi.properties file as follows.

```
connection.<factory name>.brokerURL=\ tcp://localhost:61616?
jms.useAsyncSend=true&jms.producerWindowSize=1024000

connection.<factory name>.producerWindowSize=1024000
```

You can also disable flow control on a per destination basis via the broker's XML configuration file. In the sample XML snippet below, all topics starting with the "TEST." prefix will have producer flow control disabled.

```
<broker>
….

<destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic="TEST.>" producerFlowControl="false"/>
        </policyEntries>
      </policyMap>
</destinationPolicy>
…
</broker>
```

13.2 Message Groups

"Message Groups" is an ActiveMQ feature that leverages the JMS-defined JMSXGroupID message property to build upon another ActiveMQ feature called, "Exclusive Consumer"[3].

To quickly recap, an exclusive consumer is a consumer that has been picked, from a cluster of consumers, by the message broker to receive all messages from the corresponding queue (i.e., the queue that the cluster is reading from). Dispatching all messages to the exclusive consumer ensures that all messages in the queue are processed in the same order that they arrived in the queue. If the exclusive consumer were to fail, the broker picks another consumer from the cluster to become the exclusive consumer. This configuration is also referred to as a master/slave cluster of consumers; the master being the consumer picked to do the receiving and slaves are the consumers on stand-by waiting to take over if and when the master fails.

With message groups, the broker will pick multiple exclusive consumers, from a cluster of consumers, and dispatches to each of these exclusive consumers all the messages that have been

[3] Unlike the "exclusive consumer" feature, which has to be enabled by assigning the "consumer.exclusive=true" parameter to a destination, the message grouping feature is automatically performed by the message broker.

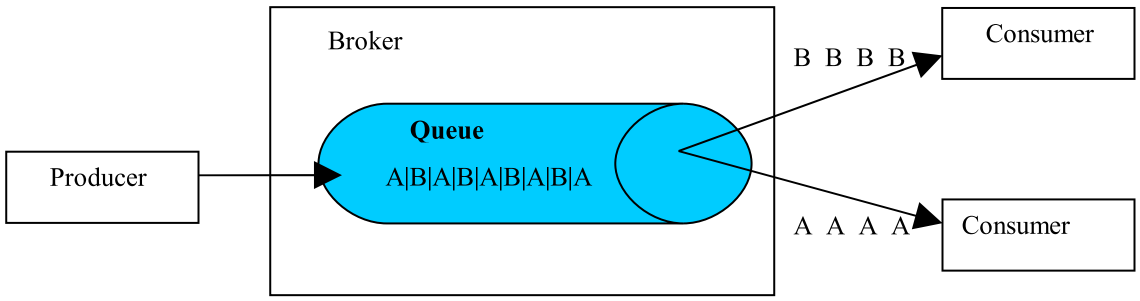
grouped by a particular JMSXGroupID property value. For example, suppose you have four consumers in a cluster that are all reading off a queue called Q.TEST. The producer that is sending messages to the queue assigns a JMSXGroupID value of either "A" or "B" to each message it produces and sends to the queue. The message broker will pick one consumer from the cluster to process all messages with the JMSXGroupID of "A" and another consumer from the cluster to process all messages with the JMSXGroupID of "B". The other two consumers that are not picked would serve as backups in case any of the selected consumers fail. So in effect, the broker is treating a message group as an atomic unit of work and load balancing those units of work across the consumer cluster. By assigning a message group to an exclusive consumer, the broker is also ensuring that the messages in the group are processed in the same order that they arrived at the queue.

So in order to take advantage of the message groups feature your producers must assign a value to the JMSXGroupID message header property. The producer can also "*close"* a message group by assigning an integer value of zero to the JMS-specific "JMSXGroupSeq" message property. After closing a message group, any subsequent message group with the same JMSXGroupID property will be assigned to another consumer. The following code snippet illustrates the sending of a message group, followed by the closing of that group.

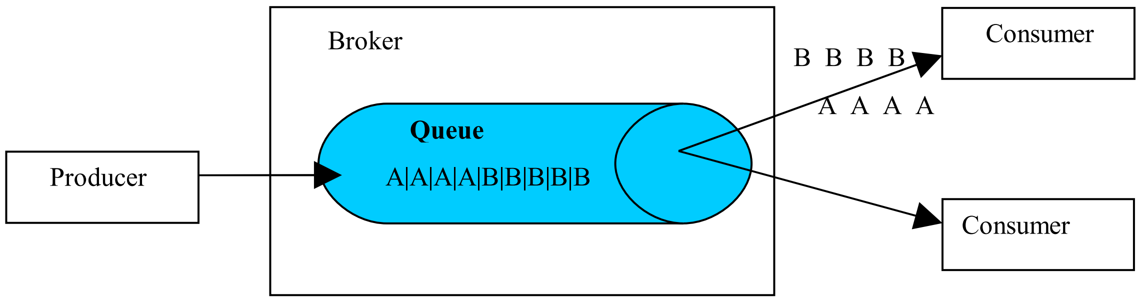
```
// Send 10 messages and group them into the "CHECKING" message group
for(int i=0; i<10; i++) {
  msg = session.createTextMessage();
  msg.setText("Message[" + i + "]");
  msg.setStringProperty("JMSXGroupID","CHECKING");
  //each message in the group needs to have a sequence number
  msg.setIntProperty("JMSXGroupSeq", i+1);
  qSender.send(msg);
}

//Now close the message group so that the next "CHECKING"
//message group can be reassigned to another consumer
msg = session.createTextMessage();
msg.setText("Message[close]");
msg.setStringProperty("JMSXGroupID","CHECKING");
msg.setIntProperty("JMSXGroupSeq",0);
qSender.send(msg);
```

The ActiveMQ message broker will also set a boolean message header property called JMSXGroupFirstForConsumer on the first message of a message group.

The figure below illustrates how the broker partitions message groups and assigns them to different consumers. In this case, the producer has produced two message groups (A and B) and each of the groups is assigned to a different consumer.



In the figure above, note how the producer has interleaved the messages for both groups. It has been noted that if the messages for the groups are not interleaved, the broker may pick the same consumer to process both message group as depicted below.



13.2.1 Message Groups vs Selectors

It is possible to emulate the message group functionality using message selectors. For example, you can create two consumers, where one employs a message selector for JMSXGroupID set to “A” and the other employs a message selector for JMSXGroupID set to “B”. However, the disadvantages of using selectors are:

- The consumer has to be aware of which selector it has to use.
- The producer also has to be aware of consumers and their selectors. For example, if a new consumer is invoked with a new message selector, then the producer has to become aware of this and specify the new JMSXGroupID value that the new consumer is looking for.
- If a consumer fails, the messages it was selecting will not be read.
- You run the risk of creating more than one consumer with the same selector, which would break the ordering of the message group.

13.3 Topic Message Ordering

If you have many publishers publishing messages to a topic that also has many subscribers, the subscribers to that topic may not all receive messages from the topic in the same order (i.e., relative to one another). For example, suppose you have publishers "A" and "B" that simultaneously publish 3 messages (A1, A2, A3, and B1, B2, B3) to the same topic. If you have two subscribers subscribed to the same topic, they may receive the messages in the following order.

```
subscriber1:  A1, A2, B1, A3, B2, B3

subscriber2:   B1, A1, B2, B3, A2, A3
```

Note that each subscriber receives the messages in the same sequential order that they were published by their respective publisher (i.e., A1, followed by A2, etc). However, the two subscribers do not receive the messages in the same order relative to one another.

When enabled for a particular topic, the "Topic Message Ordering" (TMO) feature ensures that each subscriber receives the messages from the topic in the same order, relative to one another. So with TMO enabled, the two subscribers would receive messages from the topic in the following order.

```
subscriber1:  A1, B1, B2, A2, A3, B3

subscriber2:   A1, B1, B2, A2, A3, B3
```

Enabling TMO is done on a per destination basis through the <strictOrderDispatchPolicy> element, which is located in the broker's XML configuration file.

The following is a sample of a broker's XML configuration file that illustrates the use of the <strictOrderDispatchPolicy> element.

```
<broker brokerName="mybroker " xmlns="http://activemq.org/config/1.0">
…

<destinationPolicy>
      <policyMap>
           <policyEntries>

        <policyEntry topic=">">
            <dispatchPolicy>
              <strictOrderDispatchPolicy />
            </dispatchPolicy>
          </policyEntry>

         </policyEntries>
     </policyMap>
  </destinationPolicy>

</broker>
```

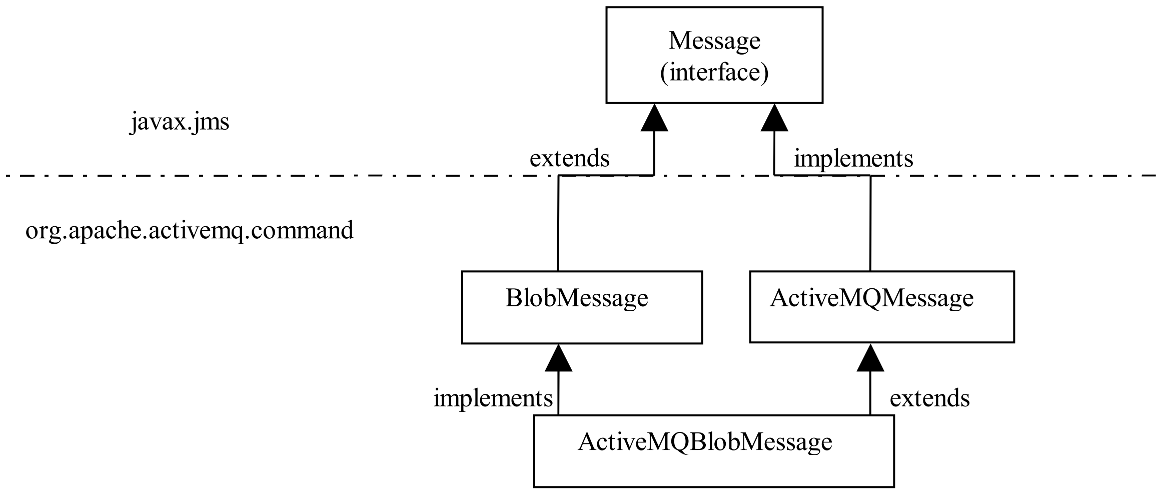
Note the wildcard value (">") that is assigned to the 'topic' attribute of the <policyEntry> element that encapsulates the <strictOrderDispatchPolicy>. This wildcard, in conjunction with the <strictOrderDispatchPolicy> element, enables TMO for all topics. In the example below, TMO is only enabled for those topics having a prefix name of "FOO.".

```
<broker brokerName="mybroker" xmlns="http://activemq.org/config/1.0">
…

<destinationPolicy>
      <policyMap>
           <policyEntries>

        <policyEntry topic="FOO.>">
            <dispatchPolicy>
              <strictOrderDispatchPolicy />
            </dispatchPolicy>
          </policyEntry>

         </policyEntries>
     </policyMap>
  </destinationPolicy>

</broker>
```

The one disadvantage of TMO is the performance cost associated with the extra synchronization required for its implementation.

13.4 Binary Large Objects (BLOBs)

A BLOB is a very large file that comprises either binary and/or text data. There may be instances where application developers have to address a requirement to send BLOBs across a distributed application while at the same time requiring some of the features that ActiveMQ provides such as reliability, high availability, transactions, load balancing, message ordering, etc. Starting with version 5.0, ActiveMQ supports the sending of BLOBs; however, your application will have to reference ActiveMQ-specific classes, which will compromise the portability of the application. The primary class to reference is ActiveMQBlobMessage, which in turn extends ActiveMQMessage and implements BlobMessage. The latter two implement and extend the javax.jms.Message interface, respectively. The figure below illustrates the hierarchy between the classes



ActiveMQ sends BLOBs "*out-of-band*", which means that the BLOB itself is transmitted through a network connection other than the main transport and network connections used by ActiveMQ for normal client-to-broker and broker-to-broker communications. This out-of-band methodology is used so as not to clog the main connections with these potentially huge data files. File transfer protocols such as FTP, SCP, or even HTTP can be used for the out-of-band transmission. Even though the transmission of a BLOB does not occur through a main ActiveMQ communication channel, the corresponding ActiveMQ command objects used to control the file's transmission do; therefore, ActiveMQ QoS features such as reliability, high availability, transactions, persistence, load balancing, message grouping, etc. can be leveraged for the transmission of the BLOB.

13.4.1 Sending a BLOB

To send a BLOB, you must first acquire a JMS Session from a Connection; however, in this case you must cast the Session to an ActiveMQSession, which implements the JMS Session interface.

```
ActiveMQSession session = (ActiveMQSession)conn.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

Through the ActiveMQSession, you then create an ActiveMQBlobMessage, which again, implements BlobMessage. You derive a BlobMessage from one of the following two methods, which are provided by ActiveMQSession.

```
public BlobMessage createBlobMessage(URL url) throws JMSException

public BlobMessage createBlobMessage(URL url, boolean deletedByBroker) throws
JMSException

public BlobMessage createBlobMessage(File file) throws JMSException
```

The first two methods create an initialized BlobMessage object that is used to send a message containing a URL, which points to some network addressable BLOB. The java.net.URL class represents a Uniform Resource Locator, which is a pointer to a "resource" (e.g., http://www.some-shared-site.com/uploads/blob.txt) on the network; in this particular case, it points to the BLOB. For the second method, the ‘deletedByBroker’ parameter specifies whether the broker is to delete the resource after the receiving client has acknowledged receipt of the message. The following example illustrates sending a URL pointing to a BLOB resource located at www.some-shared-site.com

```
BlobMessage message = session.createBlobMessage(new URL("http://localhost:
8161/fileserver/blob.txt/"));

producer.send(message);
```

The next section will describe how the consumer downloads the BLOB file.

The third createBlobMessage method creates an initialized BlobMessage object that is used to send a message containing a pointer to a local File (BLOB). When the message is sent, via the Producer.send() method, the corresponding ActiveMQSession will first upload the BLOB file according to the “*BLOB transfer policy”* that is associated with the corresponding Connection.

Remember that a JMS Producer is derived from a JMS Session, and a Session is derived from a JMS Connection.

For example, this simple jndi.properties file demonstrates how a BLOB transfer policy is assigned to all Connections rendered by the “local” connection factory.

```
java.naming.factory.initial =

     org.apache.activemq.jndi.ActiveMQInitialContextFactory

connectionFactoryNames = local


connection.local.brokerURL=tcp://localhost:61616?
jms.blobTransferPolicy.uploadUrl=http://localhost:
8161/fileserver/&jms.copyMessageOnSend=false


queue.Q.REQ = Q.REQ?consumer.prority=10
```

The uploadUrl associated with the transfer policy specifies the URL (directory) that will receive all BLOBs for this particular connection factory. After uploading the BLOB to the uploadUrl, the ActiveMQSession will send the BlobMessage on to the broker and target destination. The default BLOB transfer policy has an uploadUrl set to “http://localhost:8080/uploads/. Note that if the transfer policy identifies a web site, the web server for that site must be configured to allow uploading.

CAUTION: note how the **jms.copyMessageOnSend** option is set to ‘false’. With this option set to true, the upload will not work. This is because the ActiveMQBlobMessage will lose its BlobUploader object during the copy process. The following JIRA exists for this problem; please check the JIRA to see if a fix has been applied to your version of ActiveMQ.

https://issues.apache.org/activemq/browse/AMQ-1770

Also, be aware that the BLOB resources that are uploaded to the web server's repository are not deleted. This is also true for the `createBlobMessage(URL url, boolean deletedByBroker) method.` See the following JIRA.

https://issues.apache.org/activemq/browse/AMQ-1529

You can use ActiveMQ's embedded web server (jetty) to service the uploading/downloading of the BLOB's. The following XML snippet, taken from the broker's XML configuration file, illustrates the XML elements used for launching and configuring the embedded jetty web server.

```
<broker>
   …
</broker>

<jetty xmlns="http://mortbay.com/schemas/jetty/1.0">
     <connectors>
       <nioConnector port="8161"/>
     </connectors>
    <handlers>
       <webAppContext contextPath="/admin"
               resourceBase="${activemq.base}/webapps/admin" logUrlOnStart="true"/>
       <webAppContext contextPath="/demo"
               resourceBase="${activemq.base}/webapps/demo" logUrlOnStart="true"/>
       <webAppContext contextPath="/fileserver"
              resourceBase="${activemq.base}/webapps/fileserver" logUrlOnStart="true"/>
    </handlers>
 </jetty>
```

The following is some sample code illustrating how to upload.

```
// Grab our JNDI context
Context ctx = new InitialContext();
/**
Lets assume the following brokerURL is associated with the connection factory
tcp://localhost:61616?jms.blobTransferPolicy.uploadUrl=http://localhost:
8161/fileserver/&jms.copyMessageOnSend=false
**/

ConnectionFactory factory = (javax.jms.ConnectionFactory) ctx.lookup("connectionFactory");

Connection conn = factory.createConnection();

Queue myQueue = (javax.jms.Queue) ctx.lookup("Q.BLOB");
```

```
ActiveMQSession session = (ActiveMQSession) conn.createSession(false,
Session.CLIENT_ACKNOWLEDGE);

MessageProducer qSender = session.createProducer(myQueue);

conn.start();

File blobFile = new File("/tmp/blobtest.htm");

ActiveMQBlobMessage message = (ActiveMQBlobMessage)session.createBlobMessage(blobFile);

qSender.send(message);
```

13.4.2 Receiving a BLOB

A consumer receives a BlobMessage in the same manner, regardless of whether it is a URL or File based BlobMessage. The best way to describe how to receive a BlobMessage is via the sample code below; please refer to the comments in the code.

```
// Assuming we have already gotten out  InitialContext,  let’s first get our
// connection factory
ConnectionFactory factory = (ConnectionFactory)ctx.lookup(“connectionFactory”);

// Now get a Connection from the factory
Connection conn = factory.createConnection();

// Lookup the queue that the BlobMessage was sent to
Queue myQueue = (Queue) ctx.lookup("Q.BLOB");

// Create an ActiveMQ-specific session for receiving the BlobMessage
ActiveMQSession session = (ActiveMQSession)
        conn.createSession(false, Session.CLIENT_ACKNOWLEDGE);

// Create a reader for the queue
MessageConsumer qReader = session.createConsumer(myQueue);

// Start the connection
conn.start();

// Wait for the BlobMessage
Message message = qReader.receive();


if (message != null ){

   // See if the message received is of type BlobMessage
```

```
    if (message instanceof BlobMessage ) {
        // Get an InputStream to read the BLOB
        InputStream in = ((BlobMessage)message).getInputStream();
        // If there was a problem with the out of band download, then an InputStream
        // will not be received.
        if( in != null ) {
            // Write BLOB data out to a file
            FileOutputStream fout = new FileOutputStream("/tmp/rcvdBlob.htm");
            for( int i; (i=in.read()) != -1; )
                fout.write(i);
            fout.flush();
            fout.close();
            // Acknowledge the message
            message.acknowledge();
        } else System.out.println("Did not get input stream");
    } else System.out.println("Message received was not instanceof  BlobMessage");
}else System.out.println("BLOB not received");
```

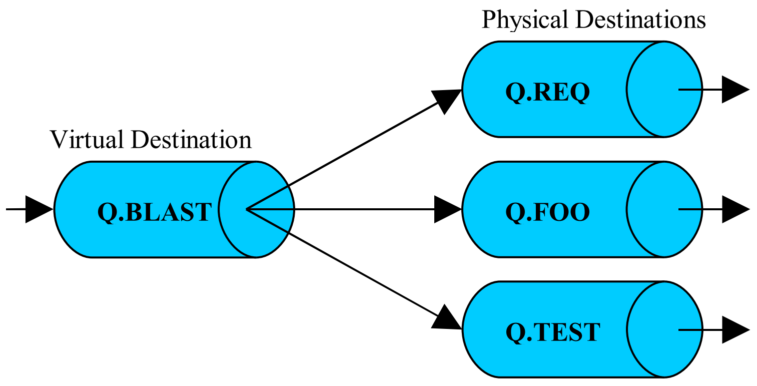
13.5 Composite Destinations

13.5.1 Client-Side Composite Destinations

The composite destination is an ActiveMQ-specific feature that allows you to map a single, "*virtual*" JMS destination to a collection of physical JMS destinations. In other words, the composite destination feature allows you to send a message to multiple physical destinations (queues and/ or topics) in one single send operation.

You can configure this feature through the client-side JNDI by assigning a list of physical destinations to one virtual destination. Take for example the jndi.properties file below and note how the virtual queue called Q.BLAST has been mapped to a list of physical queues. When a producer sends a message to the Q.BLAST destination, that same message will be forwarded to each of the three physical queues: Q.REQ, Q.FOO, and Q.TEST.

```
java.naming.factory.initial =
     org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616
queue.Q.BLAST = Q.REQ, Q.FOO, Q.TEST
```



You can also mix and match the composite list with both topics and queues. If you have a physical destination list comprising both queues and topics, you must qualify entries with either the “topic://” or “queue://” scheme prefix. Given the JNDI example above, suppose you want to add a topic (TOPIC.TEST) to the existing composite list of physical queues (Q.REQ, Q,FOO, and Q.TEST) so that when your producer sends a message to Q.BLAST that message will be sent to all three queues and a topic. You would modify the composite list as depicted in this jndi.properties file.

```
java.naming.factory.initial =
     org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
connection.local.brokerURL = tcp://linux02:61616
queue.Q.BLAST = Q.REQ, Q.FOO, Q.TEST, topic://TOPIC.TEST
topic.TOPIC.TEST = TOPIC.TEST
```

13.5.2 Broker-Side Composite Destinations

The previous section showed you how to configure composite destinations on the client. This section describes how to configure composite destinations through the broker’s XML configuration file, which allows you to centralize the configuration of the composite destinations. The sample broker XML configuration file below illustrates how the composite destination in the previous section is configured via the <destinationInterceptors> element.

```
<broker brokerName="mybroker " xmlns="http://activemq.org/config/1.0">
...
   <destinationInterceptors>
      <virtualDestinationInterceptor>
        <virtualDestinations>
          <compositeQueue name="Q.BLAST">
            <forwardTo>
```

```
              <queue physicalName="Q.REQ" />
              <queue physicalName="Q.FOO" />
              <queue physicalName="Q.TEST" />
              <topic physicalName="TOPIC.TEST" />
            </forwardTo>
          </compositeQueue>
        </virtualDestinations>
      </virtualDestinationInterceptor>
    </destinationInterceptors>

  </broker>

...

</beans>
```

With the broker configuration above, all producers connecting to the broker, and not just the ones using the jndi.properties file described in the previous section, can reference the "Q.BLAST" composite destination.

13.5.3 Selecting Composite Destinations

Starting with version 5.0 of ActiveMQ, you can assign selectors to composite destinations. This feature allows you to route a message from a composite virtual destination to its corresponding physical destinations based on selectors that may be assigned to the physical destinations. Let's take the following example; you have a virtual composite destination called, Q.BLAST and it is mapped to three physical destinations (Q.REQ, Q.FOO, and Q.TEST). Without a selector assigned to any of the three physical destinations, a message sent to Q.BLAST will be forwarded to all three physical destinations. Let's now suppose that you want a message sent to Q.REQ only if the message has a header property called "color" with an assigned value of "blue". This would be the corresponding configuration in the broker's XML configuration file.

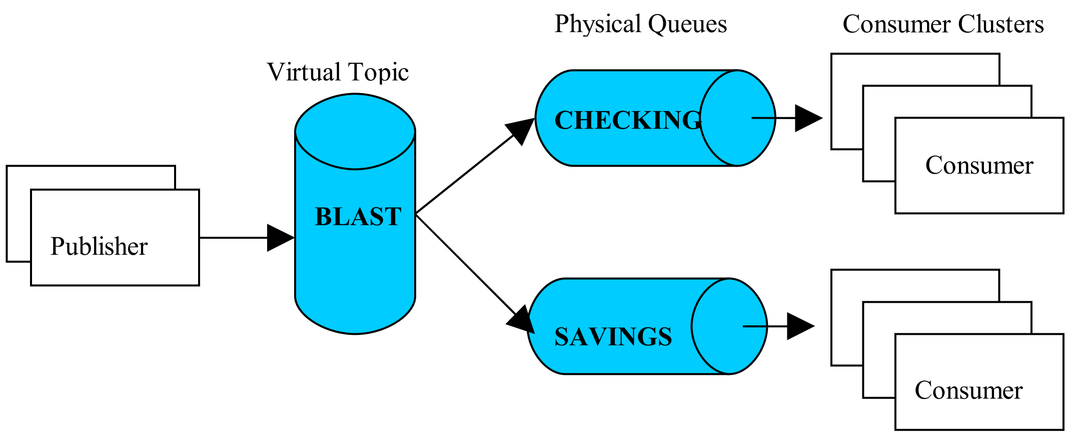
```
<beans>

  <broker brokerName="mybroker " xmlns="http://activemq.org/config/1.0">

...

   <destinationInterceptors>
      <virtualDestinationInterceptor>
        <virtualDestinations>
          <compositeQueue name="Q.BLAST">
            <forwardTo>
              <filteredDestination selector=”color=’blue’” queue="Q.REQ" />
              <queue physicalName="Q.FOO" />
              <queue physicalName="Q.TEST" />
            </forwardTo>
          </compositeQueue>
        </virtualDestinations>
      </virtualDestinationInterceptor>
    </destinationInterceptors>

  </broker>

...
```

```
</beans>
```

13.6 Virtual Topics

To set the stage for describing Virtual Topics, let’s first briefly discuss JMS durable subscriptions. A durable subscription is derived, by a consumer (subscriber), from a JMS session. The durable subscription is used by the subscriber to receive all messages published to a topic; including all messages published while the subscriber was inactive. The JMS specification mandates that a session with durable subscriptions must pertain to a JMS connection that has been assigned a unique client identifier. The consumer must also provide a unique name for each durable subscription that it creates.

One disadvantage with durable subscriptions is that a subscriber must ensure a unique client identifier and subscription name. Another disadvantage is that having to use unique identifiers precludes you from load balancing messages across multiple consumers and the ability to failover across those consumers. By using a virtual topic you can address these disadvantages and still retain the publish/subscribe semantics of a topic. With a virtual topic, a publisher publishes messages to the virtual topic just as it would with a regular topic. However, the topic is presented to the consumer as a queue and not a topic. For example, suppose we have a publisher that publishes messages to a virtual topic called “VirtualTopic.BLAST” (the “VirtualTopic.” prefix is required for virtual topics) and we have consumers on different systems (SAVINGS and CHECKING) that need to receive messages that have been published to VirtualTopic.BLAST. A consumer in the SAVINGS system that wishes to receive messages published to VirtualTopic.BLAST associates itself with a queue called “Consumer.SAVINGS.VirtualTopic.BLAST” (the “Consumer.” Prefix is required) and a consumer in the CHECKING system associates itself with a queue called “Consumer.CHECKING.VirtualTopic.BLAST”. In effect, what you have are consumers subscribing to a queue in order to receive messages published to a topic. This gives you all the advantages a queue provides such as message grouping, load balancing, and failover. The queue also serves as a “durable” subscription because if the consumer becomes inactive, then when it is re-activated it will receive any messages that were published to the topic while it was inactive.



The consumer also does not have to concern itself with creating a unique client identifier, which is required by durable subscriptions. Another advantage is that the queues can be monitored, via JMX, which allows you to look at the current depth of the queue and browse through the messages.

As described above, virtual topic names must have the "VirtualTopic." prefix and corresponding queues must follow this syntax

```
Consumer.<name assigned by consumer>.VirtualTopic.<name assigned by publisher>.
```

Via the broker's XML configuration file, you can alter this naming convention to whatever you wish.

The following broker XML configuration file illustrates how to make the topic called "BLAST" a virtual topic.

```
<beans>

  <broker brokerName="mybroker " xmlns="http://activemq.org/config/1.0">

...

    <destinationInterceptors>
      <virtualDestinationInterceptor>
        <virtualDestinations>
          <virtualTopic name="VirtualTopic.BLAST"/>
        </virtualDestinations>
      </virtualDestinationInterceptor>
    </destinationInterceptors>
  </broker>

...

</beans>
```

This example illustrates how the consumer prefix has been changed from the default “Consumer.” to “VirtualTopicConsumers.”.

```
<beans>

  <broker brokerName="mybroker " xmlns="http://activemq.org/config/1.0">

...

    <destinationInterceptors>
      <virtualDestinationInterceptor>
        <virtualDestinations>
          <virtualTopic
                  name="VirtualTopic.BLAST"
                  prefix="VirtualTopicConsumers.*." />
        </virtualDestinations>
      </virtualDestinationInterceptor>
    </destinationInterceptors>
  </broker>

...

</beans>
```

In the example below, all topics are being treated as virtual topics; note the use of the wildcard character “>” to indicate 'match all topics'.

```
<beans>

  <broker brokerName="mybroker" xmlns="http://activemq.org/config/1.0">

...

    <destinationInterceptors>
      <virtualDestinationInterceptor>
        <virtualDestinations>
          <virtualTopic name=">" />
        </virtualDestinations>
      </virtualDestinationInterceptor>
    </destinationInterceptors>
  </broker>

...

</beans>
```

This is a sample jndi.properties file identifying the topic called “BLAST” as a virtual topic.

```
java.naming.factory.initial =
     org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactoryNames = local
```

```
connection.local.brokerURL = tcp://linux02:61616
topic.BLAST = VirtualTopic.BLAST
```

This code sample illustrates the publisher looking up the virtual topic and then publishing 5 messages to the topic.

```
...

javax.naming.Context ctx = new InitialContext();

// Create a connection factory for the target broker.
ConnectionFactory factory = (javax.jms.ConnectionFactory)
            ctx.lookup("localConnectionFactory");

//Now have the connection factory render a connection
Connection conn = factory.createConnection();

// Lookup the virtual topic called BLAST
Topic myTopic = (Topic) ctx.lookup("BLAST");

// Have the connection render a session for sending to this Topic
Session session = conn.createSession(false, Session.CLIENT_ACKNOWLEDGE);

// Now have the session render a producer for the Topic
MessageProducer topicSender = session.createProducer(myTopic);

//Start the connection
conn.start();

// publish the five test messages
for(int i=0; i<5; i++){
    msg = session.createTextMessage();
    String temp = Math.random() + "{}";
    msg.setText("Test Message["  + i + "] " + temp);
    topicSender.send(msg);
}
...
```

CAUTION: When using virtual topics within a network of brokers, you need to take some precautions to avoid duplicate messages. This is because a broker, in a network of brokers, will forward a message sent to the virtual topic and may also forward messages sent to the corresponding physical queues. Therefore, you must ensure that the messages sent to the Consumer.*.VirtualTopic.> destination (physical queues) are not forwarded. This can be done by disabling the forwarding of messages on the corresponding physical queues.

Here is an example taken from a broker's configuration file:

```
...
```

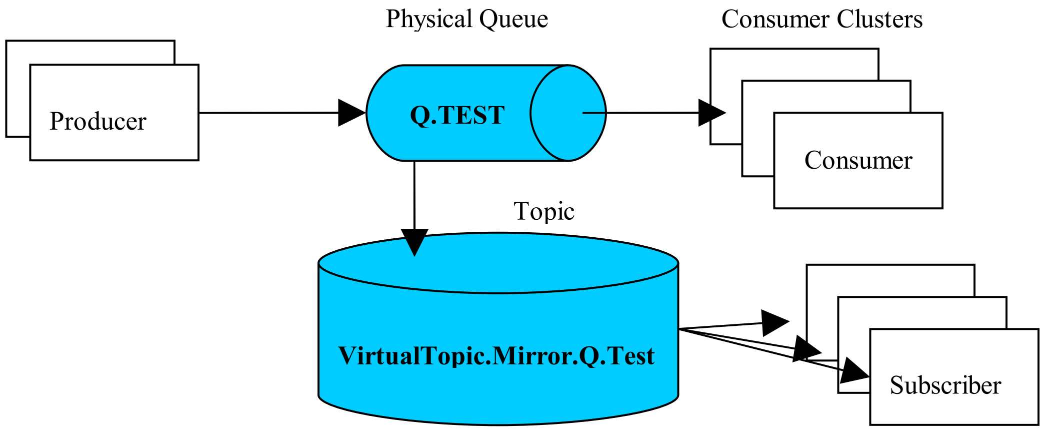
```
<networkConnectors>
  <networkConnector uri="static://(tcp://localhost:61617)">
      <excludedDestinations>
         <queue physicalName="Consumer.*.VirtualTopic.>"/>
       </excludedDestinations>
   </networkConnector>
</networkConnectors>
...
```

13.6.1 Virtual Topics vs. Composite Destinations

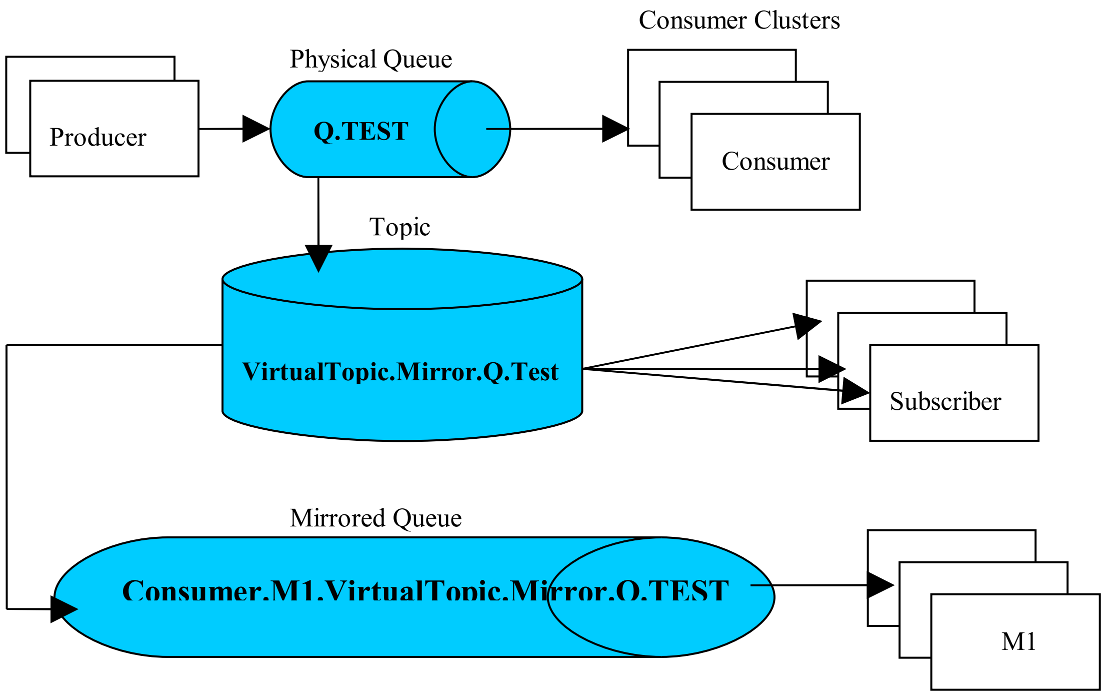
By now you may be telling yourself that virtual topics can be implemented using composite destinations, which is a correct statement. For example, you can create a composite topic that maps to one or more physical queues. However, this requires that you statically configure the mapping from topic to queues via the JNDI or broker's XML configuration file. On the other hand, as long as you adhere to the proper naming convention for virtual topics, the virtual topic and its queues can be created dynamically.

13.7 Mirrored Queues

When enabled, the mirrored queues feature allows an application to monitor the messages that flow through a queue. The way this feature works is that whenever a message is sent to a particular queue, that message is also published to a topic; therefore, any application that is interested in monitoring the message traffic through the queue can subscribe to that topic. For example, suppose you have one or more producers sending messages to a queue called "Q.TEST" and you also have one or more consumers receiving messages from that same queue. Now suppose you also have one or more client applications that wish to monitor the messages that are flowing through "Q.TEST". The monitoring applications would simply subscribe to a topic called "VirtualTopic.Mirror.Q.TEST", which would allow them to begin receiving all messages flowing through "Q.TEST" (see figure below).



You can also piggy-back this feature with “Virtual Topics” (see section 13.6). For example, suppose you have many instances of a client called “M1” that wish to monitor the queue called “Q.TEST” and you would like to “load balance” the messages across all instances of M1. You can achieve this by having all instances of M1 subscribe to the queue called “Consumer.M1.VirtualTopic.Mirror.Q.TEST”. So with this configuration, you have essentially created a “mirrored queue” for “Q.TEST”.



By default, the mirrored queue functionality is disabled, because enabling it will cause ActiveMQ to create a topic for each queue that is created. To enable this functionally set the <broker> element's useMirroredQueues attribute to "true" as depicted in this snippet of a broker's XML configuration file.

```
<beans>

  <broker brokerName="mybroker" useMirroredQueues="true"
         xmlns="http://activemq.org/config/1.0">
...

  </broker>

...

</beans>
```

13.8 Message Transformers

Starting with version 5.0, ActiveMQ allows application developers to add their own message transformation objects to the ActiveMQ message bus. An ActiveMQ transformation object, or transformer, must implement the org.apache.activemq.MessageTransformer interface. A transformer can be positioned such that it is given a message prior to the message being delivered to the message broker and/or prior to the message being delivered to the consumer.

To install a message transformer, you invoke the setTransformer() method of any one of the following ActiveMQ classes: ActiveMQConnectionFactory, ActiveMQConnection, ActiveMQSession, ActiveMQMessageConsumer, and ActiveMQMessageProducer. The transformer is inherited across child objects; therefore, if you install it via the ActiveMQConnectionFactory, it will be inherited by all subsequent connections, sessions, consumers, and producers.

13.9 Connection Pooling

The org.apache.activemq.pool package provides a JMS service provider object (PooledConnectionFactory) through which you can pool instances of Connection, Session, and MessageProducer objects. However, note this comment taken from ActiveMQ 5.0's PooledConnectionFactory.java file.

```
/**
 * <b>NOTE</b> this implementation is only intended for use when sending
 * messages. It does not deal with pooling of consumers; for that look at a
 * library like <a href="http://jencks.org/">Jencks</a> such as in <a
 * href="http://jencks.org/Message+Driven+POJOs">this example</a>
 *
 */
```

ActiveMQ's PooledConnectionFactory is primarily intended for use with frameworks/tools like Spring's JmsTemplate and Jenck's AMQPool. As the comment above mentions, if you intend to pool consumers, then use the Jenck's package, which can also be deployed via Spring.

13.10Spring Support

Refer to ActiveMQ's Spring Support page for examples on how to configure an ActiveMQ JMS client in Spring. In that page, note the following:

- The 'zeroconf' transport is no longer supported in ActiveMQ 5.0
- The reference to Spring's SingleConnectionFactory class; this class returns the same Connection from all createConnection() calls, and ignores calls to Connection.close(). So in effect, it allows you to re-use the same connection.
- The JmsTemplate gotchas - http://activemq.apache.org/jmstemplate-gotchas.html

Refer to the Spring JMS web page for more information on using JMS within the Spring framework.

13.11Message Cursors

In previous versions of ActiveMQ, the message broker held all in-transit messages in memory. With this memory model, a consumer that cannot keep up with the rate of messages being produced for a particular destination would cause the broker's maximum message limit to be reached. When this limit is reached, the broker cannot accept any more messages from producers; therefore, those producers that are in the process of sending messages to the broker are blocked until the slow consumers can alleviate the message congestion at the broker.

Starting with version 5.0, a new memory model has been implemented that prevents producers from being blocked by slow consumers. This memory model employs "message cursors" and is fully described on the following ActiveMQ web page.

http://activemq.apache.org/message-cursors.html

13.12Enterprise Integration Patterns

Via Apache Camel, ActiveMQ supports the "Enterprise Integration Patterns" (EIP) that are documented in the corresponding book by Gregor Hohpe and Bobby Woolf.

This ActiveMQ web page describes how to use the EIP/Camel within ActiveMQ.

http://activemq.apache.org/enterprise-integration-patterns.html

13.13Individual Acknowledge

Starting with version 5.2 of ActiveMQ, a new ActiveMQ-specific message acknowledgment has been added to the existing set that is mandated by the JMS specification. This acknowledgement mode is referred to as INDIVIDUAL_ACKNOWLEDGE and is used to acknowledge an individual message. When using this acknowledgement mode, the message.acknowledge() method invocation will only acknowledge that message. This is opposed to the CLIENT_ACKNOWLEDGE, which acknowledges all messages received up to that point by the session.

Be advised that this is an ActiveMQ proprietary acknowledgement mode; it does not adhere to the JMS specification.

13.14Prioritizing Messages

ActiveMQ does not currently support the prioritized message consumption. That is, messages with a higher priority are consumed prior to those messages with a lower priority. However, there are a couple of techniques that can be implemented to address prioritized message consumption. One technique relies on the use of selectors, while the other relies on Camel to implement the resequencer messaging pattern. Click here for more information on these two techniques.

14 Extending ActiveMQ's Functionality

ActiveMQ incorporates a plug-in architecture that allows you to extend its functionality via an architecture that is similar, in concept, to the plug-in architecture found in the Apache web server. That is, you develop a plug-in module that adheres to a defined interface, which allows the plug-in to be included in the core engine's processing chain. In ActiveMQ's case, the interface is defined by a combination of the org.apache.activemq.broker.BrokerPlugin and org.apache.activemq.broker.Broker classes. The BrokerPlugin is a very simple interface comprising one method call.

```
package org.apache.activemq.broker;

/**
 * Represents a plugin into a Broker
 *
 * @version $Revision: 564271 $
 */

public interface BrokerPlugin {

   /**
    * Installs the plugin into the interceptor chain of the broker, returning the new
    * intercepted broker to use.
    */
   Broker installPlugin(Broker broker) throws Exception;

}
```

You create a class that implements this interface and is called out as a plugin (Spring bean) in the ActiveMQ XML configuration file (see sample below). Your plugin bean can then optionally reference other Spring beans that are listed in the XML file.

```
<beans xmlns="http://www.springframework.org/schema/beans"
         xmlns:amq="http://activemq.org/config/1.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
        http://activemq.apache.org/camel/schema/spring
        http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

 <!-- Allows us to use system properties as variables in this configuration file -->
 <bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

  <broker xmlns="http://activemq.org/config/1.0" brokerName="localhost"
           dataDirectory="${activemq.base}/data" plugins="#myPlugin">

  <!-- The transport connectors ActiveMQ will listen to -->
  <transportConnectors>
    <transportConnector name="openwire" uri="tcp://localhost:61616" />
  </transportConnectors>
 </broker>

  <bean id="myPlugin" class="org.myorg.MyPlugin">
         <!-- You can reference one or more Spring beans in this file -->
         <property name="myMgr" ref="myManager"/>
  </bean>

  <bean id="myManager" class="org.myorg.MyManager">
         <property name="fooList">
            <list>
                 <value>foo</value>
                 <value>foo2</value>
            </list>
         </property>
  </bean>

</beans>
```

You can also call out plugin's from within the <plugin> element as this example illustrates.

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:amq="http://activemq.org/config/1.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
       http://activemq.apache.org/camel/schema/spring
       http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

<!-- Allows us to use system properties as variables in this configuration file -->
<bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

 <broker xmlns="http://activemq.org/config/1.0" brokerName="localhost"
           dataDirectory="${activemq.base}/data">

 <!-- The transport connectors ActiveMQ will listen to -->
 <transportConnectors>
   <transportConnector name="openwire" uri="tcp://localhost:61616" />
 </transportConnectors>

 <plugins>
    <bean id="myPlugin" class="org.myorg.MyPlugin">
                              …
   </bean>
 </plugins>

 </broker>

</beans>
```

At startup, the broker's main processing engine calls your plug-in's installPlugin() method. This method creates and returns an object that extends org.apache.activemq.broker.BrokerFilter.

```
import org.apache.activemq.broker.Broker;
import org.apache.activemq.broker.BrokerPlugin;

public class MyPlugin implements BrokerPlugin {

        public Broker installPlugin(Broker broker) throws Exception {
            return new MyBroker(broker);
        }

}
```

The BrokerFilter class is a convenience class that implements the org.apache.activemq.broker.Broker interface. The Broker interface defines all the main engine operations (e.g., addConnection, addSession, etc.) that your implementation can intercept.

The class that extends BrokerFilter overrides any of the methods that are defined in the Broker interface so that it can intercept the corresponding core engine's operations. Here's an example of a class that extends BrokerFilter and intercepts/overrides the addConnection() and addSession() Broker methods/operations.

```
import org.apache.activemq.broker.Broker;
import org.apache.activemq.broker.BrokerFilter;
import org.apache.activemq.broker.ConnectionContext;
import org.apache.activemq.command.ConnectionInfo;

public class MyBroker extends BrokerFilter {

     public MyBroker(Broker next) {
        super(next);
    }

    public void addConnection(ConnectionContext context, ConnectionInfo info)
            throws Exception {

        // Your code goes here
    // Then call your parent
    super.addConnection(context, info);
  }

  public void addSession(ConnectionContext context, SessionInfo info)
        throws Exception {

     //  Your code goes here…

     // Then call your parent
     super.addSession(context, info);
  }
}
```

The last thing your code does in each of the method implementations is to call the BrokerFilter parent class so that it can call the next plug-in in the processing chain.

15 Destination Policies

ActiveMQ supports a number of different policies which can be assigned to individual destinations (queues, topics) or to wildcards of queue/topic hierarchies. This makes it easy to configure how different regions of the JMS destination space are handled.

The following are examples of different policies that can be customized on a per destination basis

```
<beans>

  <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
/>

  <broker persistent="false" brokerName="${brokername}"
xmlns="http://activemq.org/config/1.0">

    <!--  lets define the dispatch policy -->
    <destinationPolicy>
      <policyMap>
        <policyEntries>

          <policyEntry topic="FOO.>">
            <dispatchPolicy>
              <roundRobinDispatchPolicy />
            </dispatchPolicy>
            <subscriptionRecoveryPolicy>
              <lastImageSubscriptionRecoveryPolicy />
            </subscriptionRecoveryPolicy>
          </policyEntry>

          <policyEntry topic="ORDERS.>">
            <dispatchPolicy>
              <strictOrderDispatchPolicy />
            </dispatchPolicy>
            <!--  1 minutes worth -->
            <subscriptionRecoveryPolicy>
              <timedSubscriptionRecoveryPolicy recoverDuration="60000" />
            </subscriptionRecoveryPolicy>
          </policyEntry>

          <policyEntry topic="PRICES.>">
            <!--  10 seconds worth -->
            <subscriptionRecoveryPolicy>
              <timedSubscriptionRecoveryPolicy recoverDuration="10000" />
            </subscriptionRecoveryPolicy>

           <!-- lets force old messages to be discarded for slow consumers -->
            <pendingMessageLimitStrategy>
              <constantPendingMessageLimitStrategy limit="10"/>
            </pendingMessageLimitStrategy>

          </policyEntry>
        </policyEntries>
      </policyMap>
    </destinationPolicy>
  </broker>

</beans>
```

16 Authentication and Authorization Services

This section discusses the authentication and authorization (A&A) services provided by an ActiveMQ message broker. When authentication services are enabled for a particular message broker, all connection requests made to that message broker must provide the proper credentials (i.e., user name and password). The message broker will reject any connection request that does not provide the proper credentials. When enabled, authorization services controls access to the destinations that are managed by the message broker. To enable authorization services, you must also enable authentication services.

Before you begin to use ActiveMQ's A&A services, ensure that you have cleared out the broker's data directory. The broker's data directory is typically found in ../activemq-data/<*broker name*>. If you enable A&A services and their exists artifacts in the data directory that were not created whileA&A services were enabled, then this will likely lead to exceptions being thrown by the message broker.

The default activemq.xml configuration file comes with three optional and enabled elements: <commandAgent>, <camelContext>, and <jetty>. If you enable A&A services, these enabled elements will more than likely cause the broker to throw security-related exceptions. This is because these elements represent functionality that is essentially represented by clients that need to connect to the broker and the connections are made without security credentials. If you do not require the functionality behind these elements, you should disable or comment-out the elements.

16.1 Authentication

To provide authentication, ActiveMQ comes with a simple Java Authentication and Authorization Services (JAAS)-based plug-in module. It is considered "simple", because it is based on three basic files and not on something more sophisticated like a replicated directory service (e.g., LDAP or Active Directory). To enable the JAAS plug-in module and authentication services, you must first add the <plugins> element and its <jaasAuthenticationPlugin> sub-element to the message brokers XML configuration file as this XML snippet illustrates.

```
<plugins>
    <!--  use JAAS to authenticate using the login.config file -->
    <jaasAuthenticationPlugin configuration="activemq-domain" />
</plugins>
```

You configure the simple JAAS plug-in module by setting the **java.security.auth.login.config** system property to point to a configuration file that must adhere to the format depicted below.

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
activemq-domain {
        org.apache.activemq.jaas.PropertiesLoginModule required
        debug=true
        org.apache.activemq.jaas.properties.user="org/apache/activemq/security/users.properties"
```

```
    org.apache.activemq.jaas.properties.group="org/apache/activemq/security/groups.properties";
};
```

If the java.security.auth.login.config system property is not set, ActiveMQ's JAAS plug-in module will search for the file called "login.config", which adheres to the above format, in the message broker's CLASSPATH. The broker's default CLASSPATH is $ACTIVEMQ_HOME/conf. The "login.config" file points to two additional files that are used to define users and user groups. These two files should be located in the same directory that contains the "login.config" file.

The users.properties file is used to define each user along with the user's password. This is an example users.properties file.

```
## ---------------------------------------------------------------------------
## Comments
##
## user-name = password
## ---------------------------------------------------------------------------
system=manager
user=password
guest=password
```

The groups.properties file is used to define what groups(s) each user is assigned to and is used by the authorization services, which is discussed in the following section. This is an example groups.properties file.

```
## ---------------------------------------------------------------------------
## Comments
##
## group=<list of comm- separated users pertianing to this group>
## ---------------------------------------------------------------------------
admins=system
users=system,user
tempDestinationAdmins=system,user
guests=guest
```

With all of the above in-place, the ActiveMQ message broker will accept only those connections, from either clients or brokers, having the proper credentials (i.e., user name, password).

If you don't wish to use the JAAS plug-in module and its corresponding external configuration files, you can instead rely on the <simpleAuthenticationPlugin> element to provide all authentication information through the brokers XML configuration file. This is illustrated in the following XML snippet.

```
<plugins>
 <!--  use the simpleAuthenticationPlugin instead of JAAS -->
 <simpleAuthenticationPlugin>
   <!-- Define all users along with their passwords and the groups -->
  <users>
    <authenticationUser
          username="system"
          password="manager"
          groups="users,admins"/>
   <authenticationUser
          username="user"
          password="password"
        groups="users"/>
    <authenticationUser
          username="guest"
          password="password"
          groups="guests"/>
  </users>
 </simpleAuthenticationPlugin>
</plugins>
```

This sample code snippet illustrates how a JMS client would create a connection with the proper credentials.

```
// Get a connection factory from the JNDI InitialContext
javax.jms.ConnectionFactory factory = (javax.jms.ConnectionFactory)
            ctx.lookup("localConnectionFactory");

// Get a connection from the connection factory's createConnection method and
// pass it the proper user name and password.
try {
      conn = factory.createConnection("user", "password");
      //Start the connection
      conn.start();
 }
 catch(JMSException e){
     System.out.println("Caught this exception: "+ e.getClass());
     System.out.println("Caught this security exception: "+ e.getMessage());
     System.exit(1);
}
```

16.1.1 Authentication and Authorization between Message Brokers

If you have enabled authentication for a particular message broker, then other brokers that wish to connect to that broker must provide the proper authentication credentials via their <networkConnector> element.

For example, let's suppose that we have a network of brokers (NoB) with the following configuration:

- The NoB comprises two brokers (BrokerA and BrokerB)

- We have enabled authentication for BrokerA via the example <simpleAuthenticationPlugin> given in the previous section.
- Authentication for BrokerB has **not** been enabled.
- BrokerA only listens for connections. In other words, BrokerA has a <transportConnector> element, but no <networkConnector> elements.

In order for BrokerB to connect to BrokerA, the corresponding <networkConnector> element in BrokerB's XML configuration file must be set up as follows.

```
<networkConnectors>
      <!-- A connector used for connecting to brokerA -->
      <networkConnector name="brokerAbridge"
                                      userName="user"
                                      password="password"
                                      uri="static://(tcp://brokerA:61616)"/>

    </networkConnectors>
```

Note how BrokerB's <networkConnector> element must provide the proper credentials in order to connect to BrokerA. . The userName assigned to that <networkConnector> element must also have the proper 'authorization' credentials if 'authorization' has been enabled on BrokerA. Messages cannot be forwarded from BrokerB to BrokerA if BrokerA has authorization enabled and BrokerB's corresponding <networkConnector> element's userName has not been given the proper authorization credentials.

Also, if BrokerA is given a <networkConnector> element so that it can initiate a connection to BrokerB, then that <networkConnector> must be given a userName/password combination that is defined in the <simpleAuthenticationPlugin> element; this is required even though BrokerB does not have authentication services enabled.

16.2 Authorization

Authorization services are used to grant user groups access rights to destinations. Access rights are granted in the form of three operations: read, write, and admin.

Operation	**Description**
read	You can browse and consume from the destination
write	You can send messages to the destination
admin	You can lazily create the destination if it does not yet exist. This allows you fine grained control over which new destinations can be dynamically created

	in what part of the queue/topic hierarchy

Access rights are configured through the <authorizationPlugin> XML element, which is located in the broker's XML configuration file. The following is an example broker XML configuration file that uses the JAAS plug-in for authentication and the <authorizationPlugin> element to assign access rights to the groups defined in the groups.properties file. Note how wildcards (">") are used for assigning access rights to a hierarchy of destination names. Full access rights should always be given to the ActiveMQ.Advisory topics, else your client will receive an exception stating it does not have access rights to these series of topics.

```
<beans>
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

  <broker xmlns="http://activemq.org/config/1.0">

    <plugins>
      <!--  use JAAS to authenticate using the login.config file on the CLASSPATH to configure JAAS -->
      <jaasAuthenticationPlugin configuration="activemq-domain" />

      <!--  configure a destination based authorization mechanism -->
      <authorizationPlugin>
        <map>
          <authorizationMap>
            <authorizationEntries>
              <authorizationEntry queue=">" read="admins" write="admins" admin="admins" />
              <authorizationEntry queue="USERS.>" read="users" write="users" admin="users" />
              <authorizationEntry queue="GUEST.>" read="guests" write="guests,users"
                                  admin="guests,users" />

              <authorizationEntry topic=">" read="admins" write="admins" admin="admins" />
              <authorizationEntry topic="USERS.>" read="users" write="users" admin="users" />
              <authorizationEntry topic="GUEST.>" read="guests" write="guests,users"
                                  admin="guests,users" />

              <authorizationEntry topic="ActiveMQ.Advisory.>" read="guests,users"
                                  write="guests,users" admin="guests,users,admins"/>
            </authorizationEntries>

            <!-- let's assign roles to temporary destinations. comment this entry if we don't want
                 any roles assigned to temp destinations  -->
            <tempDestinationAuthorizationEntry>
              <tempDestinationAuthorizationEntry read="tempDestinationAdmins"
                        write="tempDestinationAdmins" admin="tempDestinationAdmins"/>
           </tempDestinationAuthorizationEntry>
          </authorizationMap>
        </map>
    </authorizationPlugin>

   </plugins>
  </broker>

</beans>
```

CAUTION: If authorization has been enabled, then all groups must be granted all access rights to the ActiveMQ.Advisory topics.

16.2.1 Controlling Access To Temporary Destinations

To control access to temporary destinations, you will need to add a <tempDestinationAuthorizationEntry> element to the authorizationMap. Through this element, you control access to all temporary destinations. If this element is not present, read, write, and admin privileges for temporary destinations will be granted to all. In the example below, read, write, and admin privileges for temporary destinations are only granted to those clients that have been assigned to the 'admin' group.

```
<broker>
  ..
   <plugins>
      ..
   <authorizationPlugin>
       <map>
         <authorizationMap>
           <authorizationEntries>
             <authorizationEntry queue="TEST.Q" read="users"
                   write="users" admin="users" />
             <authorizationEntry topic="ActiveMQ.Advisory.>" read="all"
                   write="all" admin="all"/>
           </authorizationEntries>
           <tempDestinationAuthorizationEntry>
             <tempDestinationAuthorizationEntry read="admin" write="admin"
                   admin="admin"/>
           </tempDestinationAuthorizationEntry>
        </authorizationMap>
      </map>
   </authorizationPlugin>
     ..
  </plugins>
  ..
</broker>
```

16.3 Camel

If you intend to use Camel against a 'secure' ActiveMQ broker, then refer to the following URL for information on how to pass a username and password to the ActiveMQ broker when using Camel.

http://activemq.apache.org/camel/jms.html

In particular, the section titled, "Configuring different JMS providers". Note how you can configure the ActiveMQ connection factory via the Spring XML file as follows. The example below is the same as found on that web page, but a userName and password property have been added.

```
<camelContext id="camel"
  xmlns="http://activemq.apache.org/camel/schema/spring">
</camelContext>

<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL"
          value="vm://localhost?broker.persistent=false"/>
      <property name="userName" value="john"/>
      <property name="password" value="secret"/>
    </bean>
  </property>
</bean>
```

16.4 TTM's Security Plugins

16.4.1 File Based Security Plugin

If you're interested in a dynamically reconfigurable security plugin, check out TTM's simple security plugin (SSP). The SSP is an ActiveMQ plugin module that provides dynamically reconfigurable authentication and authorization security services. Dynamic reconfiguration allows you to update the SP's configuration without having to stop and restart the ActiveMQ message broker. This is also referred to as "hot deployment" or "dynamic reloading". The SSP is free and can be downloaded from this web page.

http://www.ttmsolutions.com/amqsec.php4

The SP package also includes the following:

- MD5-obfuscated passwords. The SP's file security realm supports passwords that have been obfuscated via the MD5 (Message-Digest algorithm 5) cryptographic hash function. This provides the added security of not having to store passwords in clear text. For example, the following entry in the security realm defines a user with a name of 'mary' and a password for 'mary' that has been obfuscated via the MD5 hash function. The entry also assigns 'mary' to the groups, 'users' and 'guests'.

```
username="mary" password="MD5:12735683dee9c7d59a54d30251bb29d0" groups="users, guests"
```

ActiveMQ’s simpleAuthenticationPlugin does not support obfuscated passwords.

- An extended version of the ActiveMQ CommandAgent called, “TtmCommandAgent”. Unlike the CommandAgent, the TtmCommandAgent works within a secured ActiveMQ message broker environment. In other words, TtmCommandAgent can be given authentication credentials (username and password), which it uses to connect with a broker that requires such credentials.
- Administrative MBean. Through this JMX MBean, you can view the SP’s properties and issue commands to the SP.

16.4.2 LDAP Based Security Plugin

TTM’s LDAP Security Plugin (LSP) is an ActiveMQ plugin module that uses a LDAP directory server (DS) to provide dynamically reconfigurable authentication and authorization (A&A) security services.

The primary benefit of using a LDAP DS is that all A&A information, which pertains to ActiveMQ clients and resources (i.e., topics and queues), is centrally and securely stored and managed. This is especially attractive for large enterprise class environments that employ 10’s if not 100’s of ActiveMQ message brokers. The combination of LSP and DS also provides for dynamic runtime configuration. This feature allows you to make modifications to the A&A information in the DS and not have to stop and restart the ActiveMQ message broker(s) to have those modifications take effect.

The SP connects or binds to a central LDAP directory service to:

1. Authenticate clients
2. Retrieve clients’ security credentials
3. Retrieve the access control lists (ACLs) assigned to a broker’s resources (destinations).

This package has been certified for the officially released version of ActiveMQ 5.1. It has not been certified with prior releases or SNAPSHOT versions of 5.1.

For more information, please download the plugin’s user guide via the following web page.

http://www.ttmsolutions.com/amqldapsec.php4

17 Performance

This section touches on some important performance factors to take into consideration when using ActiveMQ.

17.1 Persistent vs. Transient Messages

When a producer sends a *persistent* message to the ActiveMQ message broker, the underlying JMS session will block until the message broker sends the session an acknowledgement that it has received the message and has written the message out to secondary storage. This is sometimes referred to as a *synchronous* send, which guarantees a high-level of reliability; however, it also introduces a latency penalty, because the session must block until it receives the acknowledgement from the message broker.

When a producer sends a transient or non-persistent message, the underlying session does not have to wait for the broker to acknowledge that it has written the message to secondary storage; therefore, the latency associated with sending a transient message is much less than that for sending a persistent message. However, unlike a persistent message, a transient message cannot survive a message broker failure.

The following sections will provide some tips on how to improve performance when using persistent messaging.

17.1.1 Asynchronous Sends

If your producer must use persistent messages, but has been designed to tolerate a few lost messages in failure scenarios, then you can disable synchronous sends and enable *asynchronous* sends. Enabling asynchronous sends for persistent messages will preclude the producer from blocking or waiting on the broker's acknowledgement message thus reducing the latency penalty associated with the synchronous sends. See section 13.1 to learn more about asynchronous sends and how to enable this feature.

17.1.2 Transactions

If your producer must send persistent messages and cannot tolerate any lost messages, then one option is to group multiple send() methods within the context of a single transaction. When a send() is part of a transaction, it will automatically operate in asynchronous mode, because it is the transaction's commit() method that will block to wait for the acknowledgment from the broker. In this particular case, the acknowledgment back to the commit method tells the session that all the messages sent within the context of the transaction have been safely written out to secondary storage. In effect, a transaction allows you to consolidate or batch many synchronous sends, and their corresponding writes to secondary storage, down to just one synchronous send.

17.2 Prefetch Limit

The prefetch limit that is assigned to a consumer is an indication of the message load the consumer can adequately process. The higher the limit, the greater the message loads that the broker can stream to the consumer. A higher prefetch limit also ensures that the consumer always has messages in its local message buffer and does not have to wait for messages to be read in from the broker thus enhancing the overall message throughput of both the consumer and message broker.

The default configuration for ActiveMQ is set up for environments that require very high performance and message throughput. To address these types of environments, the default prefetch limit (see section 12.1) for consumers is set to a high value (1000 messages); therefore, the default dispatch policy is one that will attempt to fill the consumer's prefetch buffer as quickly as possible. With an environment that utilizes a small average message size, a potential side-effect of this default configuration is that a small number of consumers, which comprise a larger cluster of consumers, may end up receiving the majority of the message load. This side-effect may not pose an issue if the processing of each message requires a relatively small amount of time. However, if the message processing time is high, then this default configuration would not be the optimum configuration, because you would want to more evenly disperse the message processing/load across the cluster of consumers.

The following describes how ActiveMQ disperses the message load under a couple of different scenarios.

1. This first scenario is one where a destination has a cluster of active consumers (i.e., more than one consumer blocked waiting for a message). As producers send messages to that destination, ActiveMQ will evenly disperse the messages across the active consumers in the cluster; no matter what the value of the prefetch limit.

2. This second scenario is one where a destination has many pending messages (i.e., messages in the destination, ready to be delivered), but there are no active consumers to receive those pending messages. Let's suppose the destination has 1000 pending messages and through some sort of activation script, a cluster of 10 consumers is then automatically activated for that destination. What will happen is that the broker will stream, up to the consumer's prefetch limit, the messages to the consumer that first connects to the broker. In other words, the broker will latch on to that first consumer and stream as many messages, as the prefetch limit allows, to that consumer. Because the default prefetch limit is 1000, that first consumer will receive all 1000 pending messages. This will raise a serious issue if each message requires a lot of processing time.

To address the potential issue described in scenario #2 above, you can lower the prefetch limit for the consumers. If, for example, you lowered the limit to 10, the broker will stream only 10 messages to the first consumer and will not stream any more messages to that consumer until it receives an acknowledgement from the consumer. This will make it more likely that the broker will dispatch messages to the rest of the consumers, as they connect with the broker.

17.3 Threads

By default, ActiveMQ makes use of several threads to process messages and isolate producers from consumers. Under certain conditions, you can increase the overall scalability/throughput of ActiveMQ by reducing this default number of threads. The following web page describes how threads are used within ActiveMQ and the options (async, dispatchAsync, optimizedDispatch) that are used to influence the number of threads used w/in ActiveMQ.

http://open.iona.com/wiki/display/ProdInfo/Understanding+the+Threads+Allocated+in+ActiveMQ

17.4 DUPS_OK_ACKNOWLEDGE

The DUPS_OK_ACKNOWLEDGE is a JMS session-specific option that instructs the session to *lazily* acknowledge the delivery of messages. For example, with DUPS_OK_ACKNOWLEDGE mode, the JMS session acknowledges the consumption of a batch of messages. Also, unlike the two other JMS session acknowledgement modes (AUTO_ACKNOWLEDGE and CLIENT_ACKNOWLEDGE), the session does not block waiting for a broker acknowledgement to the session's acknowledgement, because no broker acknowledgement is requested in the DUPS_OK_ACKNOWLEDGE mode. These two characteristics of DUPS_OK_ACKNOWLEDGE mode will generally improve overall message throughput. However, with DUPS_OK_ACKNOWLEDGE mode there is no guarantee that messages are delivered and consumed only once. In general, messages will not be redelivered very often; they are redelivered only in cases of failure, where the broker has not received a client acknowledgement for a message it has delivered. Consumers should use DUPS_OK_ACKNOWLEDGE mode if they don't care about duplicate delivery.

17.5 Optimized Acknowledge

When using a JMS session that is in AUTO_ACKNOWLEDGE mode, ActiveMQ will acknowledge receipt of messages in batches (similar to DUPS_OK_ACKNOWLEDGE) to improve performance. The batch size is 50% of the prefetch limit for the consumer. You can turn off this batch acknowledgment by setting the ActiveMQ connection factory property called, "**optimizeAcknowledge"** to be **false.**

17.6 Asynchronous Dispatch

Asynchronous dispatch allows the message broker to efficiently address ‘slow’ consumers; however, it does increase context switching between threads. For more information, see section 12.3.

17.7 Embedded Brokers

The use of embedded or intra-VM brokers greatly reduces message latency because there is no network hop between client and broker. A client communicates with its embedded broker via direct method invocation as opposed to streaming ActiveMQ command objects through a network connection. More information on embedded brokers can be found in sections 2 and 3.2.1.

17.8 Message Copy

In order to be JMS compliant, ActiveMQ’s Producer.send() method copies the corresponding message to a new message object. However, if your producer does not reuse the message object or does not mutate the message after it is sent, then it can disable the copying of the message, which improves throughput. To disable message copying, set the ‘copyMessageOnSend’ URI connection property to ‘false’. This is an example of setting it for a tcp connector URI.

```
tcp://localhost:61616?jms.copyMessageOnSend=true
```

Or you can use several methods for assigning it to a connection factory in your jndi.properties file as follows

```
connection.<factory name>.brokerURL= tcp://localhost:61616?
                    jms.useAsyncSend=true

connection.<factory name>.copyMessageOnSend’ = false
```

17.9 OpenWire Loose Encoding

By default, ActiveMQ uses a “tight encoding” format for the OpenWire protocol. This type of encoding produces smaller protocol packets, but requires more CPU utilization. For those environments where network bandwidth is not an issue, but CPU utilization is, switching to the “loose encoding” format should help reduce CPU usage.

To disable the tight encoding format, set the ‘tightEncodingEnabled’ wire format property to ‘false’ in the connection URI as follows.

```
tcp://localhost:61616?wireFormat.tightEncodingEnabled=false
```

1. *JAR File Requirements*

The following table lists jar files that are relevant to an ActiveMQ client application and describes when the jar file must be in the client's CLASSPATH.

JAR Files	Description
apache-activemq-<*version*>.jar	This is the main ActiveMQ jar file and is always required. This jar file is typically found in the ACTIVEMQ_HOME directory. If you're using a SNAPSHOT release of ActiveMQ, the jar file will typically be called, activemq-all-<*version*>-SNAPSHOT.jar
spring-<*version*>.jar xbean-spring-<*version*>.jar	These jar files are required if your client invokes an 'embedded' broker and configures the embedded broker through an external XML configuration file. More information on embedded brokers can be found in sections 2 and 3.2.1.
jmdns-1.0-RC2.jar	This jar file is required if your client application is using the 'rendezvous' based discovery agent.
activemq-optional-<*version*>.jar commons-httpclient-<*version*>.jar xstream-<*version*>.jar xmlpull-<*version*>.jar	These jar files are required if you client uses the http and/or https transport connector URIs. The jar files can all be found in the ACTIVEMQ_HOME/lib/optional directory.