Hewlett Packard Enterprise

HPE StoreOnce Catalyst Developers Implementation Guide v9.0

Contents

Implementation Guide Introduction

This Implementation Guide contains information for Independent Software Vendors (ISVs) wishing to integrate HPE StoreOnce Catalyst into their backup application software using the HPE StoreOnce Catalyst Software Development Kit (SDK).

It provides information on the StoreOnce Catalyst concepts and technology and has example code snippets as well as integration requirements.

Example code is syntactically highlighted for readability and is displayed in a blue box as below.

```
int main(int argc, char *argv[])
{
    printf("Hello Catalyst Application!")
}
```

To ensure readability some code examples will require other sections in the guide to fully implement.

The SDK includes `OSCLT_Interface.h`, `OSCMN_Types.h` and `OSCLI_ErrorCodes.h` with `ExampleCode.c` for a working demonstration of the SDK.

Section 1: StoreOnce Catalyst Concepts

1.1 Introduction to StoreOnce Catalyst

The StoreOnce Catalyst API provides an advanced RPC based interface through which applications can have a rich interaction with StoreOnce appliances. The benefits of using an RPC based interface instead of SCSI, CIFS or NFS is that the calling application can have a much richer interaction with the storage device.

Capabilities that were previously beyond view of the backup application, such as data deduplication and replication are now presented in a manner which allows the backup applications to present their users with more information and also to make decisions based on the information returned.

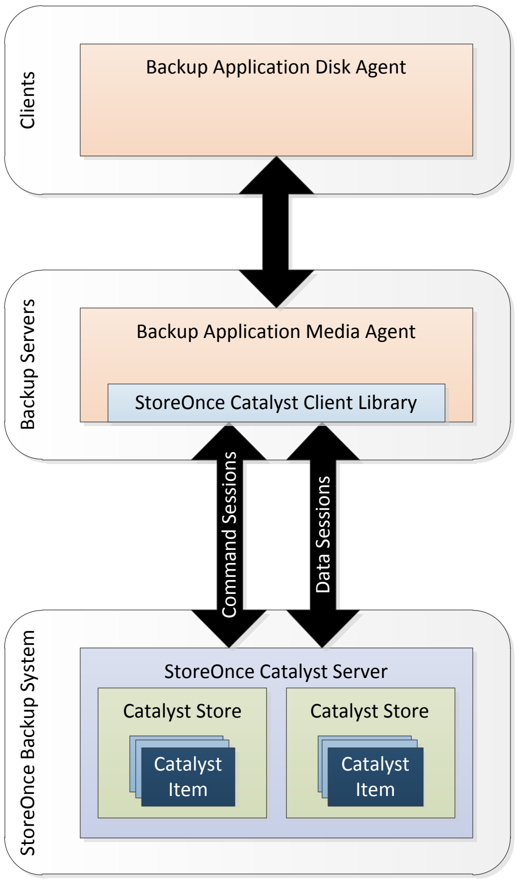


Figure 1 - HPE StoreOnce Catalyst Client Deployment

The Catalyst Client Library should be distributed as part of the backup application and will typically be included in any backup application components that would interact directly with a target storage device. Figure 1 shows, for example, how the client library is embedded within the backup application media agent, allowing the media agent to communicate directly with the Catalyst Stores on the StoreOnce Backup System.

1.1.1 Overview of ISV Integration with HPE StoreOnce Catalyst

StoreOnce Catalyst targets should be thought of as a new device type that has been optimized for the purposes of data protection. StoreOnce Catalyst should not be used as general purpose storage as the data deduplication data paths will most commonly have sub-optimal throughput performance for more random access workloads.

For data protection there are four basic use cases:

- Backup
  - Create objects on the storage target and populate them with data.
  - Perform bandwidth efficient backups over LAN, WAN and Fibre Channel.
- Clone
  - Clone data from a previous object into a new backup object
- Restore
  - Recover data from objects on the storage target.
- Copy
  - Create independent redundant copies of data on multiple storage targets.
  - Perform bandwidth efficient copies over both LAN and WAN.
  - Copies are managed by 3rd party software but data transfer is directly between the storage targets.

Despite StoreOnce Catalyst targets being a disk based solution, when designing and integrating with 3rd party software it is often useful to think of the Catalyst Item as an advanced tape drive. This means that you should aim to write to the Catalyst Item in a sequential manner with a single data stream per Catalyst Item. Unlike tape however, Catalyst Items have additional properties such as metadata, descriptive tags, data deduplication and the ability to perform bandwidth efficient copies between storage targets.

Copying Catalyst Items between storage targets is controlled by the 3rd party software, this allows the software to control when copies are made and how many copies are made. The key benefit being that the 3rd party software is aware of all the copies of the data that exist and can therefore control their lifecycles.

Additionally the 3rd party software may specify the copy to be a region within a Catalyst Item rather than the whole Catalyst Item. This allows the 3rd party software, for example, to create copies that have unique header information. An example flow chart showing the copy process can be found in page 104.

StoreOnce Catalyst integration is commonly achieved in one of two ways:

1) By creating a specific new type of storage target within the 3rd party software that presents the full range of StoreOnce Catalyst functionality to the user. The benefit of this approach is that a much richer user experience where data management is viewed in a consistent way through both the 3rd party software and the storage target.
2) By creating a generic storage target type within the 3rd party software and integrating with StoreOnce Catalyst through a plug-in architecture. This caters for frameworks where the 3rd party software integrates with multiple storage target vendors, such as Symantec OpenStorage.

1.2 StoreOnce Catalyst Terminology

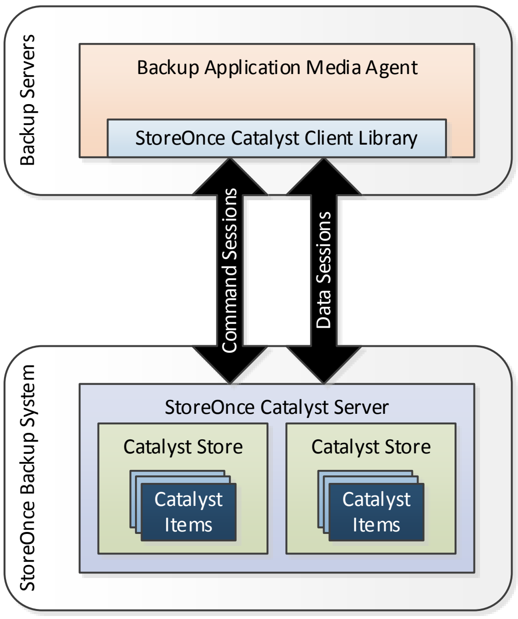


Figure 2 - StoreOnce Catalyst

1.2.1 Catalyst Client

A Catalyst Client is the name used to describe any application used to connect to the Catalyst Server using the Catalyst Client API. A typical Catalyst Client would be a data backup application that is using the Catalyst Server as a target for the backup data.

Catalyst Clients are described in more detail in the Catalyst Clients section on page 51.

1.2.2 Catalyst Server

A Catalyst Server is the name used to describe any target appliance that has StoreOnce Catalyst capabilities, for example the HPE StoreOnce 6500. The Catalyst Server is responsible for the management of Catalyst Client connections to Catalyst Stores and Catalyst Items, as well as the control of asynchronous copy operations between Catalyst Stores.

A Catalyst Server provides access to state information that a backup application may request. This includes the number of Catalyst Client sessions that can be opened to the Catalyst Server, the number of Catalyst Stores configured and capacity utilisation. Each Catalyst Server may host one or more Catalyst Stores.

Catalyst Servers are described in more detail in the Catalyst Servers section on page 54.

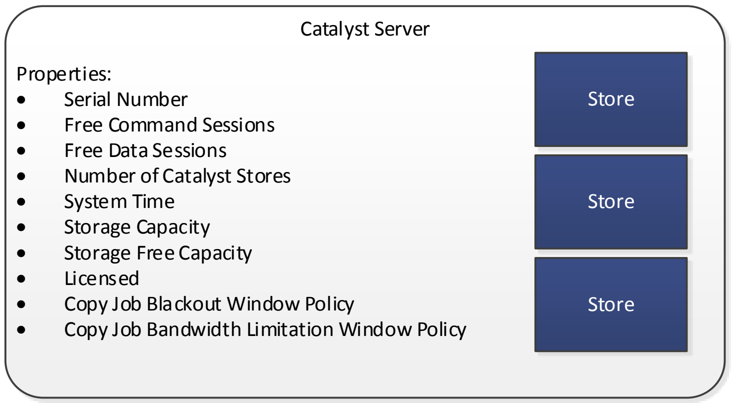


Figure 3 - Catalyst Server overview

1.2.3 Catalyst Store

A Catalyst Store is a container in which multiple Objects may be stored. If the Catalyst Store supports deduplication, then all Objects written to this Catalyst Store will have their data content deduplicated to reduce the amount of storage capacity required for the Catalyst Item. Deduplication only occurs across Objects within the same Catalyst Store.

Each Catalyst Store has a unique identifier known as a Key. The Key must be unique within the scope of the Catalyst Server and cannot be altered after creation. An optional description may also be provided.

Information such as the number of Objects within the Catalyst Store, the deduplication efficiency within the Catalyst Store and the number of running data and copy jobs are all reported when the Catalyst Store is queried.

Catalyst Stores are described in more detail in the Catalyst Stores section on page 65.

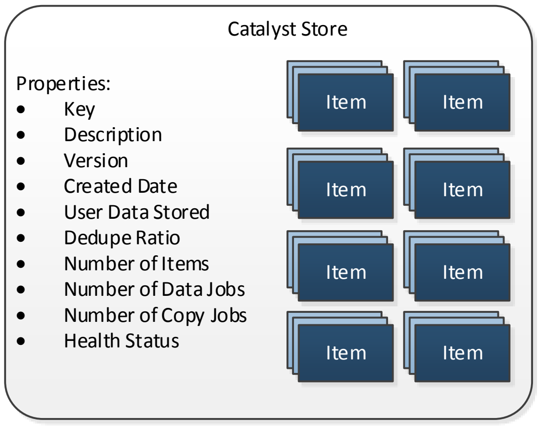


Figure 4 - Catalyst Store overview

1.2.4 Catalyst Item

Items (also referred to as Objects in the SDK) consist of three elements:

- Catalyst Item Properties
- Meta Data
- Catalyst Item Data

Each Catalyst Item within a Catalyst Store must have a unique identifier, known as a Key, which shall be supplied when the Catalyst Item is created. Once a Catalyst Item is created, the Key may not be modified. The Catalyst Item Key must be specified whenever a data connection is opened.

Information such as the Created Date and the Last Modified Date are set by the Catalyst Server and are stored as UTC timestamps. These timestamps may not be manually modified by the Catalyst Client.

A Tag List property allows series of text identifiers to be associated with a Catalyst Item. Tags may be specified at Catalyst Item creation time and modified at any time using the osCltCmd_ModifyObjectMeta operation. For more details see Tag Lists on page 77.

The State property indicates the current state of a Catalyst Item and under normal operation shall always have a state of `OSCMN_CONFIG_STATE_CONFIGURED`. When creating or deleting a Catalyst Item the state will be `OSCMN_CONFIG_STATE_CREATING` or `OSCMN_CONFIG_STATE_DELETING` respectively. States when the Catalyst Item is non-operational are `OSCMN_CONFIG_STATE_UNCONFIGURED`, `OSCMN_CONFIG_STATE_FAILED_TO_CREATE` and `OSCMN_CONFIG_STATE_FAILED_TO_DELETE`. The State property is fixed by the Catalyst Server and may not be modified by the Catalyst Client.

The Storage Mode provides information about whether the Catalyst Item was stored using variable block deduplication or fixed block deduplication. When creating a Catalyst Item the storage mode selected must match a storage mode supported by the Catalyst Store. This Storage Mode information is important when copying data between Objects as the Objects must have the same Storage Mode. For more details see Storage Mode on page 77.

Meta Data may be stored in a Catalyst Item by a client caller. For more information about Meta data see Metadata on page 78.

The Catalyst Item data part of a Catalyst Item is where the user data is located and can only be accessed using a data session. There is no limited to the amount of data that can be stored in a Catalyst Item but a Catalyst Store may have a storage quota applied.

When listing Items in a Catalyst Store, all of the properties will be returned in an array. The caller may also optionally request metadata be included in the response but Catalyst Item data is not returned during a list operation.

Catalyst Items are described in more detail in the

Catalyst Items section on page 76.

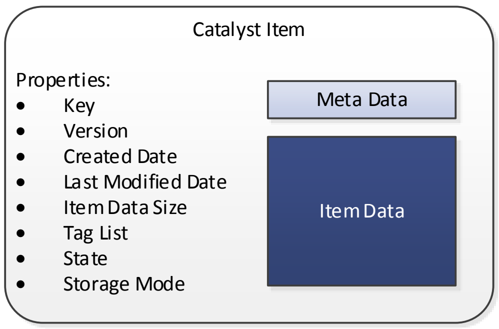


Figure 5 - Catalyst Item overview

1.2.5 Command Sessions

A command session is a socket connection established between a Catalyst Client and a Catalyst Server for the purposes of managing Catalyst Servers, Catalyst Stores and Catalyst Items.

Command Sessions are described in more detail in the

Command Sessions section on page 19.

1.2.6 Data Sessions

A data session is a socket connection established between a Catalyst Client and a Catalyst Item for the purposes of data transfer, such as writing and/or reading data.

Data Sessions are described in more detail in the Data Sessions section on page 30.

1.2.7 Data Job Records

For every data session opened, a new data job record will be created. A data job record details the meaningful events that were performed during a data session. Their purpose is to allow Catalyst users the ability to review information about their data session. An example use would be for a Catalyst object which is created and then written to via a data session. A data job record will be created listing the object name, client IP address from which the session was written, the amount of data written, dedupe ratio; among others. Data job records are also created for read data operations.

Data jobs records have an expiration period. This expiration period is defined at a per Catalyst store level. The client should can set the retention period via `OSCMN_sJobRetentionPolicyType`. It should be set to the default `OSCMN_JOB_RETENTION_POLICY_DEFAULT_DAYS` (90 days) unless the user selects otherwise. The Retention periods must be between 1 and `MaximumJobRetentionPeriod` in Server Capabilities.

Data Jobs are described in more detail on page 94.

1.2.8 Copy Job Records

Similar to data job records, every Catalyst Copy operation will create a copy job record. A copy job record details the meaningful information about a Catalyst copy session. Their purpose is to allow Catalyst users the ability to review information about their Catalyst copy session. An example use would be for a Catalyst Item which is copied from source to target. Both the source and target Catalyst Items will have copy job records created. The source will create an “Outbound Copy Job” record and the target an “Inbound Copy Job” record. The information recorded is similar to the above Data Job Records, including: name, client IP address from which the session was written, the amount of data written, dedupe ratio; among others.

Copy jobs records have an expiration period. This expiration period is defined at a per Catalyst store level. The client can set the retention period via `OSCMN_sJobRetentionPolicyType`. It should be set to the default `OSCMN_JOB_RETENTION_POLICY_DEFAULT_DAYS` (90 days) unless the user selects otherwise. The Retention periods must be between 1 and `MaximumJobRetentionPeriod` in Server Capabilities.

Copy Jobs are described in more detail on page 98.

1.2.9 Federated Catalyst

Within a multi-node StoreOnce appliance, such as the HPE StoreOnce 6500, the maximum size of a Catalyst Store will be limited by the capacity available behind each couplet of the cluster. This means that users must configure multiple Catalyst Stores across the cluster and define within their backup software which backups will be directed to which Catalyst Stores, in order to address all available storage.

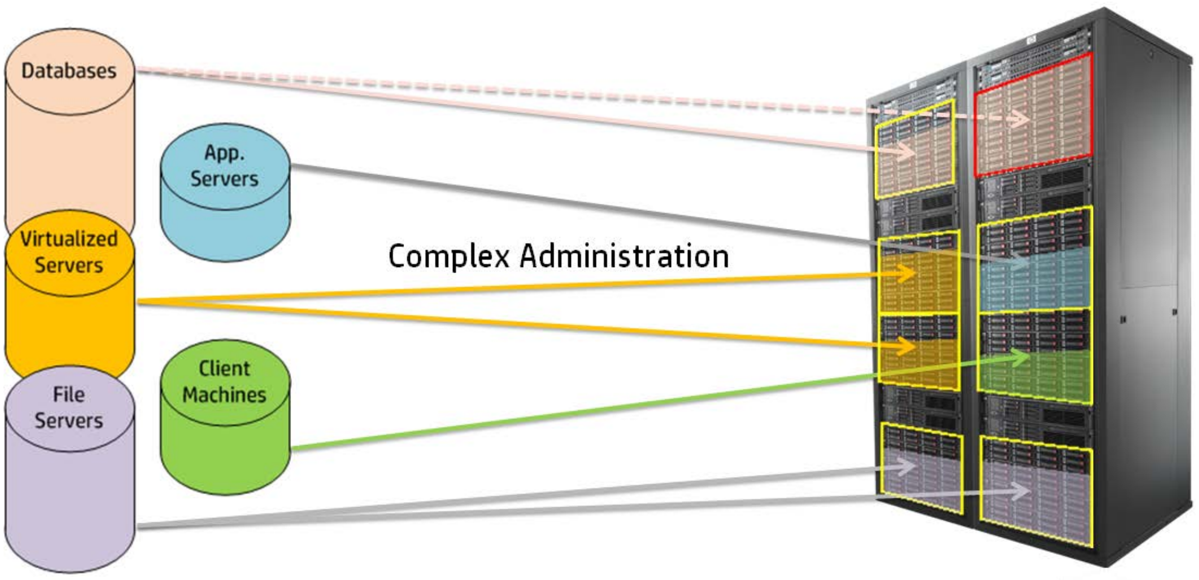


Figure 6 - Individual Pools of Storage in a Cluster

Federated Catalyst introduces a new capability to the StoreOnce Catalyst API whereby a logical Catalyst Store may now be created across multiple nodes within a cluster. This removes the need for customer to configure multiple Catalyst Stores across the cluster and complex policies within their backup software.

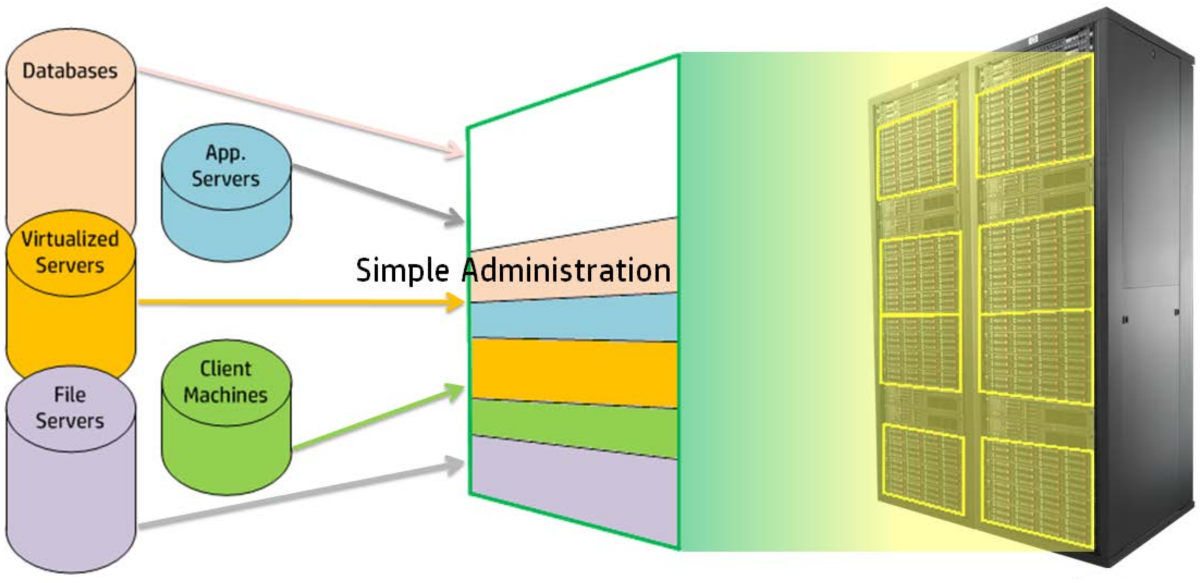


Figure 7 - Federated Pools of Storage in a Cluster

Backup applications integrating with StoreOnce Catalyst are able to support Federated Catalyst Stores with only a few minor changes, described in more detail later in this document.

1.2.10 Federated Catalyst Store

A Federated Catalyst Store comprises of individual Catalyst Stores configured across multiple nodes. Each of the individual Catalyst Stores that make up the federation will store a portion of the data written to the Federated Catalyst Store. Storage utilisation across the nodes is then managed by the Catalyst Client.

The Catalyst Client interacts with the Federated Catalyst Store by establishing connections to each individual Catalyst Stores and routing requests to the appropriate nodes on behalf of the backup application. This means that a backup application does not need to manage the complexities of device management but instead can continue to address a single target device.

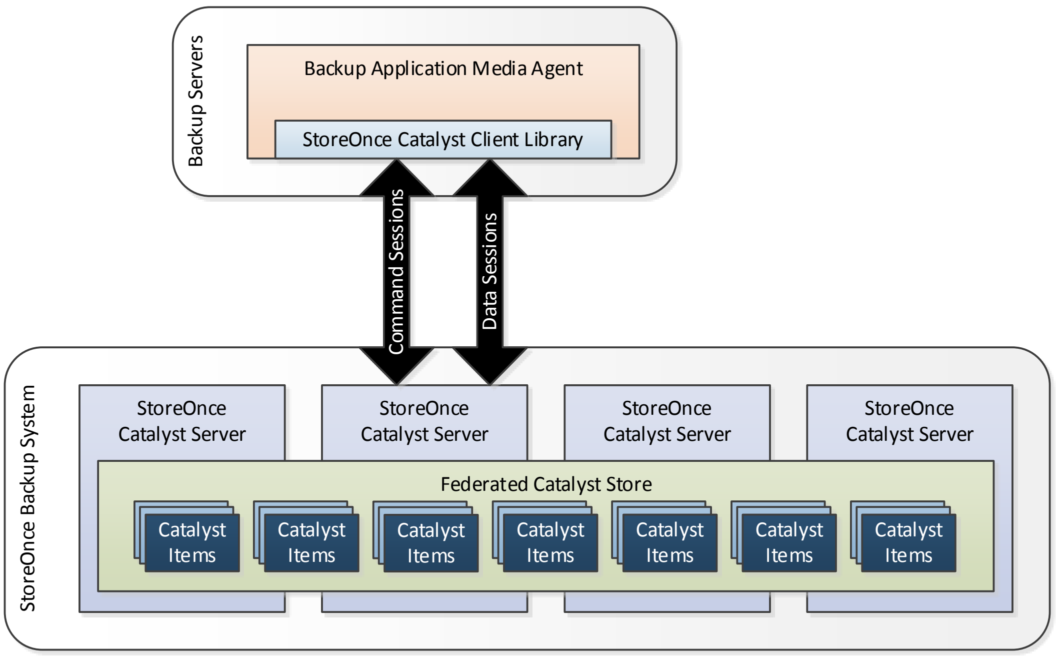

Backup Servers
Backup Application Media Agent
StoreOnce Catalyst Client Library
Command Sessions
Data Sessions
StoreOnce Backup System
StoreOnce Catalyst Server
StoreOnce Catalyst Server
StoreOnce Catalyst Server
StoreOnce Catalyst Server
Federated Catalyst Store
Catalyst Items

Figure 8 - Federated Catalyst

More detail on how to integrate with Federated Catalyst Stores can be found in the Catalyst Stores on page 65.

1.2.11 Federated Catalyst Item

Federated Catalyst Items (also referred to as Objects in the SDK) are the same as regular Catalyst Items with the exception that they exist in Federated Catalyst Stores. In most cases, when a Catalyst Item is created in a Federated Catalyst Store it will be created on each individual federation member. Each of these Catalyst Items will then hold a portion of the data written to the Federated Catalyst Item.

Federated Catalyst Items are described in more detail in Federated Catalyst Items on page 88.

1.2.12 Catalyst Over Fibre Channel

Catalyst Over Fibre Channel allows a backup application to use the Catalyst software over a fibre channel connection by opening a command or data session using a Catalyst Over Fibre Channel identifier (prepended with "COFC-") rather than an IP address. The Catalyst Client from protocol v6 automatically handles the fibre channel protocol internally and so the same client functions can be used as Catalyst over Ethernet except for Copyjobs that are allowed over FC from protocol v9 onwards.

1.2.13 StoreOnce Catalyst Customer and API Reference

Table 1 provides a cross reference between the StoreOnce Catalyst terminology used in the customer viewable interface such as the appliance management interface and the internal ObjectStore API that is used by integrators in the SDK.

Table 1 – Terminology references

Customer Terminology	**SDK Terminology**
HPE StoreOnce Catalyst Server	ObjectStore Server
HPE StoreOnce Catalyst Store	ObjectStore
HPE StoreOnce Catalyst Item	Object
HPE StoreOnce Federated Catalyst Store	Teamed ObjectStore
HPE StoreOnce Federated Catalyst Item	Teamed Object

Section 2: StoreOnce Catalyst Command Sessions

2.1 Command Sessions

A command session is established between the Catalyst Client and the Catalyst Server using the `osCltCmd_OpenCommandSession` operation. This will return a session handle that should then be passed with subsequent command operations. The session is closed by calling the `osCltCmd_CloseCommandSession` operation.

The number of concurrent command sessions that a Catalyst Server can support is not without limit and therefore a Catalyst Client should take care to minimize the number of open command sessions open to a given Catalyst Server. This is best achieved by only maintaining a command session for the duration of a single command operation.

A typical command session would be as follows:

- osCltCmd_OpenCommandSession
- osCltCmd_GetServerProperties
- osCltCmd_CloseCommandSession

It should also be noted that inactive command sessions will timeout after five minutes.

A command session handle must not be shared between overlapping command session operations.

The command operations available are:

Table 2 - Command session supported protocols

Command	Protocol v4	Protocol v5	Protocol v6	Protocol v7	Protocol v8	Protocol v9
`osCltCmd_OpenCommandSession`	✓	✓	✓	✓	✓	✓
`osCltCmd_CloseCommandSession`	✓	✓	✓	✓	✓	✓
`osCltCmd_GetServerProperties`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListObjectStores`	✓	✓	✓	✓	✓	✓
`osCltCmd_CreateObjectStore`	✓	✓	✓	✓	✓	✓
`osCltCmd_ModifyObjectStore`	✓	✓	✓	✓	✓	✓
`osCltCmd_DeleteObjectStore`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListObjects`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListObjectsCount`		✓	✓	✓	✓	✓
`osCltCmd_CreateObject`	✓	✓	✓	✓	✓	✓
`osCltCmd_ModifyObjectMeta`	✓	✓	✓	✓	✓	✓
`osCltCmd_DeleteObject`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListObjectDataJobs`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListObjectDataJobsCount`		✓	✓	✓	✓	✓
`osCltCmd_QueueObjectCopyJob`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListOriginObjectCopyJobs`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListOriginObjectCopyJobsCount`		✓	✓	✓	✓	✓
`osCltCmd_ListDestinationObjectCopyJobs`	✓	✓	✓	✓	✓	✓
`osCltCmd_ListDestinationObjectCopyJobsCount`		✓	✓	✓	✓	✓
`osCltCmd_CancelOriginObjectCopyJob`	✓	✓	✓	✓	✓	✓
`osCltCmd_CancelDestinationObjectCopyJob`	✓	✓	✓	✓	✓	✓
`osCltCmd_SetServerProperties`		✓	✓	✓	✓	✓
`osCltCmd_SetClientPermissionsChecking`		✓	✓	✓	✓	✓
`osCltCmd_ListClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_CreateClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_ModifyClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_ModifyClientPassword`		✓	✓	✓	✓	✓

`osCltCmd_DeleteClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_ListObjectStoreClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_AddObjectStoreClientPermission`		✓	✓	✓	✓	✓
`osCltCmd_RemoveObjectStoreClientPermission`		✓	✓	✓	✓	✓
`osCltCmd_SetServerProperties`		✓	✓	✓	✓	✓
`osCltCmd_SetClientPermissionsChecking`		✓	✓	✓	✓	✓
`osCltCmd_ListClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_CreateClientPermissions`		✓	✓	✓	✓	✓
`osCltCmd_OpenTeamedCommandSession`		✓	✓	✓	✓	✓
`osCltCmd_TeamedExpandObjectStore`			✓	✓	✓	✓
`osCltCmd_TeamedContractObjectStore`			✓	✓	✓	✓
`osCltCmd_TeamedExpandObject`			✓	✓	✓	✓
`osCltCmd_TeamedContractObject`			✓	✓	✓	✓

* Protocol v5 returns Federated (Teaming) values in structures but they will always be false.

Full details of the command session operations can be found in `OSCLT_Interface.h`.

2.1.1 Opening a Command Session

Command Sessions should be opened on demand, to issue a command or short burst of commands, and then closed again. In many cases it will be preferential to create a wrapper function around the `osCltCmd_OpenCommandSession` and `osCltCmd_OpenTeamedCommandSession` operations which are able to manage the resource allocation and retry logic.

When opening a command session via `osCltCmd_OpenCommandSession` the client should specify logging information. The client should specify the log session ID (`OSCLT_sSessionLogIdType`), log path (`OSCLT_sPathnameType`), log level (`osClt_SessionLogLevel`) and maximum log file size. Error Logging is described in more detail in the Error Logs section on page 45.

2.1.1.1 Code Example – Open a Command Session

```
#define OBJSDK_CMD_SESSION_ID "OBJSDK_CMD"
int objSDKPrv_OpenCmdSession(const char *pServerAddress,
OSCLT_sSessionHandleType *pCmdSessionHandle)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;

    int retryCount = 0;
    int sleepSeconds = 2;

    OSCLT_sSessionLogIdType sessionLogID;
    OSCMN_sAddressPortType serverAddr;
    OSCLT_sPathnameType logPath;
    OSCMN_sCredentialsType cred;
    OSCLT_sSessionLogDescriptorType sessionLogDescriptor;
    OSCLT_sSessionOptionsType sessionOptions;

    // << Memset all structures>>

    strncpy(sessionLogID.String,
            OBJSDK_CMD_SESSION_ID,
            sizeof(sessionLogID.String));

    if (sessionLogID.String[sizeof(sessionLogID.String) - 1])
    {
        returnStatus = OSCLT_ERR_INTERNAL_ERROR;
        return returnStatus;
    }
    sessionLogID.StringSize = strlen(sessionLogID.String);

    strncpy(serverAddr.Address.String,
            pServerAddress,
            sizeof(serverAddr.Address.String));

    if (serverAddr.Address.String[sizeof(serverAddr.Address.String) - 1])
    {
        returnStatus = OSCLT_ERR_INTERNAL_ERROR;
        return returnStatus;
    }
    serverAddr.Address.StringSize = strlen(serverAddr.Address.String);
    serverAddr.PortNumber = OSCLT_DEFAULT_COMMAND_PORT;
    // <<Set credentials | See ‘Client Authentication’ section>>
    // <<Set logging | See ‘Error Logging’ section>>

    while (retryCount < 33)
    {
        callStatus = osCltCmd_OpenCommandSession(&sessionLogDescriptor,
                                                 &sessionOptions,
                                                 &serverAddr,
                                                 &cred,
                                                 pCmdSessionHandle);

        // <<If connection unsuccessful, retry | See 'Command Session
        //Retries' section>>
        if (callStatus == OSCLT_ERR_MAXIMUM_SESSIONS)
        {
            sleep(sleepSeconds);
            if (sleepSeconds < 30)
            {
                sleepSeconds = sleepSeconds * 2;
                sleepSeconds = (sleepSeconds > 30)? 30 : sleepSeconds;
            }
        }
        else
        {
            break;
        }

        retryCount++;
    }

    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Unable to open command session, error : %s.\n",
               objSDKPrv_ErrorNumToString(callStatus));
        returnStatus = callStatus;
        return returnStatus;
    }
    return returnStatus;
}
```

Within the shipping application, for debug / support purposes the backup application is recommended to expose a means of increasing the log level and log file size, on a temporary basis. This may be within a configuration file or within the application interface. The levels of available logging are:

```
OSCLT_LOG_LEVEL_EXTENDED_DEBUG
OSCLT_LOG_LEVEL_DEBUG
OSCLT_LOG_LEVEL_TRACE
OSCLT_LOG_LEVEL_INFO
OSCLT_LOG_LEVEL_QUIET
OSCLT_LOG_LEVEL_ERROR
```

During application development it may be useful for the client to run in OSCLT_LOG_LEVEL_INFO mode, but the level should be set to OSCLT_LOG_LEVEL_ERROR before shipping to customers. Debug and extended debug are very verbose and should only be used if instructed by HP.

If an Open Session operation does not return OSCLT_ERR_SUCCESS then it will free up all resources it may have used (sockets, mallocs, etc). Close Session does not need to be called for such session, and must not be called as the Session Handle (out parameter) will be invalid (see above). Using the session handle parameter will almost certainly result in a segv.

2.1.2 Command Session Retries

Catalyst Servers support a limited number of concurrent command sessions. An indicator to the number of available command sessions is returned by osCltCmd_GetServerProperties in FreeCommandSessions. This value should only be treated as a guide however due to the very short life of command sessions and therefore the backup application should include logic to retry the opening of command sessions if the open fails with an error of OSCLT_ERR_MAXIMUM_SESSIONS.

It is recommended that a retry mechanism be written such that successive retry would progressive increase the delay between retry attempts. For example, on the first retry attempt the delay would be 2 seconds. On the second retry the delay would increase to 4 seconds and one the third retry the delay would increase to 8 seconds. The increasing delay should be capped at 30 second intervals, as shown in Table 3.

Table 3 - Recommended Command Session Retry Delays

Retry Attempt	Retry Delay (Sec)
1	2
2	4
3	8
4	16
5 – 33	30

After all the retries have been attempted (15 minutes), the operation should fail and the backup application should make a decision about how to proceed. For example, the backup application could reschedule the backup jobs to run at another time.

2.1.3 Closing a Command Session

Users of the API are expected to be responsible citizens and close sessions when they have finished with them. This way the Server is able to service many more clients than it has worker threads available as clients will time slice on a Command Session by Command Session basis.

For both Command Sessions and Federated Command Sessions, osCltCmd_CloseCommandSession should be called to close the session.

If osCltCmd_CloseSession returns an error, it will still free up all resources used by that session. osCltCmd_CloseSession does not need to be called again for that session, and must not be called as the Session Handle will have been invalidated by the initial osCltCmd_CloseSession operation. Using the session handle parameter after a close will almost certainly result in a segv.

Command Sessions which are not explicitly closed by the Catalyst Client will timeout after 5 minutes of inactivity and will be forcibly closed by the Catalyst Server.

2.1.3.1 Code Example – Close a Command Session

```
int objSDKPrv_CloseCmdSession(OSCLT_sSessionHandleType *pCmdSessionHandle)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;

    if (pCmdSessionHandle)
    {
        callStatus = osCltCmd_CloseCommandSession(pCmdSessionHandle);
        if (callStatus != OSCLT_ERR_SUCCESS)
        {
            printf("Error : Unable to close command session.\n");
            return returnStatus;
        }
    }
    return returnStatus;
}
```

2.1.4 Federated Command Sessions

A Federated Command Session provides the Catalyst Client with a method of performing the same command session task or query across a federation of Catalyst Servers. This ensures consistency across the federation when performing such tasks as creating Catalyst Items, as well as proving consolidated results when performing lists of Catalyst Items or Data Jobs.

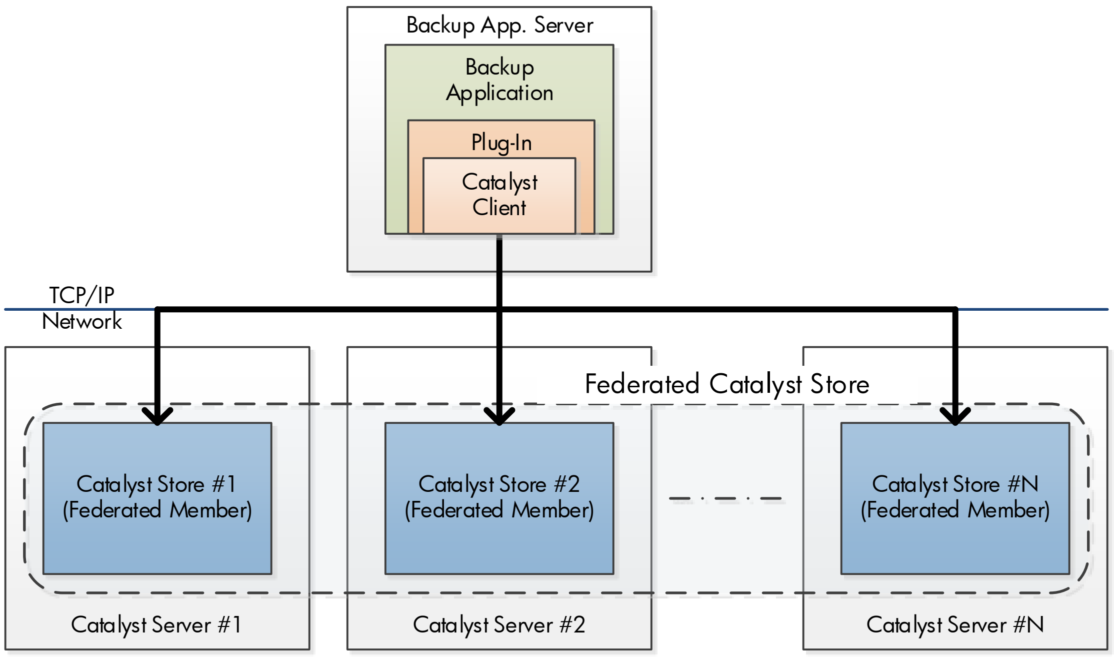


Figure 9 - Federated Members

Only a subset of commands may be run from a Federated Command Session:

- osCltCmd_GetServerProperties

- osCltCmd_ListObjectStores (only if key filter is provided with ExactMatchOnly)
- osCltCmd_CreateObjectStore
- osCltCmd_ModifyObjectStore
- osCltCmd_DeleteObjectStore
- osCltCmd_ListObjects
- osCltCmd_CreateObject
- osCltCmd_ModifyObjectMeta
- osCltCmd_DeleteObject
- osCltCmd_ListObjectDataJobs

When calling any of the commands above, the request will be applied to all Federated Members and any response will be a merged response from all Federated Members. This means, for example, when listing data jobs in a Federated Catalyst Store the job details will show the combined data job records from each of the Federated Members.

Unlike a non-federated command session, a Federated Command Session is bound to the Federated Store, so the caller must have permission to access the Federated Store to be able to open the session. Federated command sessions can be opened in two modes:

- Bootstrap Mode
- Normal Mode

2.1.4.1 Bootstrap Mode

'Bootstrap' mode is where a blank store key is provided to the open Federated Command Session operation. It is used for Federated Store creation, expansion, and contraction operations (and can be used for store modification).

A Federated policy is provided in the create/expand/contract operation, which contains, amongst other things, the address for all Federation Members. This is needed because a backup application typically only allows for a single address to be provided for a backup target device, which is the server address passed into the open Federated session operation. The Federation policy can then be consulted by the client API so it can internally connect to all the other Federation members. Once the create/modify/expand/contract operation is completed the session becomes a normal command session. If the operation fails, the bootstrap session is locked down, and should be closed using `osCltCmd_CloseCommandSession`.

2.1.4.2 Normal Mode

'Normal' mode is where a non-blank Federated Store key is provided in the open Federated command session operation. It is used for all Federated command session operations other than store creation, expansion and contraction.

The client API pulls the Federation policy from the Federated member whose server address is provided in the open, and then consults this so it can internally connect to all the other Federated Members.

Store modification is usually done in a ‘normal’ Federated Command Session, but can optionally be done in bootstrap mode if a Federated Member has changed its address (so opening a normal Federated Command Session won't work as the Federated Member address details stored in the Federate Policy on the server are wrong).

There is no bootstrap Federated store delete. Instead, each individual single Federate Member store can be deleted via a normal command session.

2.1.4.3 Code Example – Open a Teamed Command Session

```
int objSDKPrv_OpenTeamedCmdSession(const char *pServerAddress,
OSCLT_sSessionHandleType *pCmdSessionHandle, const char *pStoreKey)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;

    int retryCount = 0;
    int sleepSeconds = OBJSDK_CMD_SESSION_RETRY_SLEEP_SECONDS_MIN;

    //<<Setup as in objSDKPrv_OpenCmdSession>>

    OSCMN_sKeyType teamedObjectStoreKey;
    OSCLT_sTeamMemberConnectionStatusType teamMemberConnectionStatus;

    if (pStoreKey == NULL) // Open connection in bootstrap mode
    {
        memset(&teamedObjectStoreKey, 0, sizeof(teamedObjectStoreKey));
    }
    else // Storekey is defined - open connection in non-bootstrap mode
    {
        memset(&teamedObjectStoreKey,
               0,
               sizeof(teamedObjectStoreKey));
        strncpy(teamedObjectStoreKey.String,
                pStoreKey,
                sizeof(teamedObjectStoreKey.String));
        if (teamedObjectStoreKey.String[sizeof(teamedObjectStoreKey.String)
                                                                       - 1])
        {
            printf("Error : Unable to set Teamed Store Key %s\n",
                    teamedObjectStoreKey.String);
            returnStatus = OSCLT_ERR_INTERNAL_ERROR;
            return returnStatus;
        }
        teamedObjectStoreKey.StringSize = strlen
            (teamedObjectStoreKey.String);
    }

    while (retryCount < 33)
    {
        callStatus = osCltCmd_OpenTeamedCommandSession(
                                    &sessionLogDescriptor,
                                    &sessionOptions,
                                    &serverAddr,
                                    &cred,
                                    &teamedObjectStoreKey,
                                    pCmdSessionHandle,
                                    &teamMemberConnectionStatus);
        //<<Use same retry logic as objSDKPrv_OpenCmdSession>>
    }

    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Unable to open teamed command session for "
            "server, error : %s.", objSDKPrv_ErrorNumToString(callStatus));
        returnStatus = callStatus;
        return returnStatus;
    }
    return returnStatus;
}
```

2.2 Section Summary and recommendated best practises

- The number of available Catalyst command sessions is limited and is different across the StoreOnce appliance range. Implement command session retry logic when `OSCLT_ERR_MAXIMUM_SESSIONS` is received and do not keep command sessions open for excessive periods. They should be short lived.
- Always close any command session opened by the application.
- There are 2 types of command sessions; teamed command sessions and non-teamed command sessions.
- Set an appropriate logging level when opening a command session.
- Expose a method within the backup application to increase log levels and log sizes

Section 3: StoreOnce Catalyst Item Data Sessions

3.1 Data Sessions

A data session is established between the Catalyst Client and a Catalyst Item using the `osCltData_OpenObjectDataSession` operation. The Catalyst Item must therefore already exist before the data session is opened. On open, a session handle is returned that should then be passed with subsequent data operations. The session is closed by calling the `osCltData_CloseObjectDataSession` operation.

Prior to StoreOnce 3.15, each Catalyst Item can only have a single data session open to it at any one time. Multi-read support of Catalyst items is supported post StoreOnce 3.15. Data sessions are typically maintained for the duration of a backup or restore operation however to avoid orphaned sessions caused by such things as loss of connectivity, inactive sessions will timeout after a period of four hours.

A data session handle must not be shared between overlapping data session operations.
A typical data session would be as follows:

```
    osCltData_OpenObjectDataSession
    osCltData_SeekToWriteBytes
    osCltData_WriteBytes
    osCltData_WriteBytes
    osCltData_WriteBytes
    osCltData_Flush
    osCltData_WriteBytes
    osCltData_WriteBytes
    osCltData_Flush
    …
    osCltData_CloseObjectDataSession
```

The supported data session operations are as follows:

Table 4 - Data Session supported protocols

Command	Protocol v4	Protocol v5	Protocol v6	Protocol v7	Protocol v8	Protocol v9
osCltData_OpenObjectDataSession	✓	✓	✓	✓	✓	✓
osCltData_CloseObjectDataSession	✓	✓	✓	✓	✓	✓
osCltData_SeekToWriteBytes	✓	✓	✓	✓	✓	✓
osCltData_SeekToReadBytes	✓	✓	✓	✓	✓	✓
osCltData_WriteBytes	✓	✓	✓	✓	✓	✓
osCltData_ReadBytes	✓	✓	✓	✓	✓	✓
osCltData_Flush	✓	✓	✓	✓	✓	✓
osCltData_SeekToCloneExtents		✓	✓	✓	✓	✓
osCltData_OpenParent		✓	✓	✓	✓	✓
osCltData_CloseParent		✓	✓	✓	✓	✓
osCltData_CloneExtent		✓	✓	✓	✓	✓
osCltData_SeekToWriteSections		✓	✓	✓	✓	✓
osCltData_SeekToReadSections		✓	✓	✓	✓	✓
osCltData_MatchSectionManifest		✓	✓	✓	✓	✓
osCltData_StoreSection		✓	✓	✓	✓	✓
osCltData_GetSectionManifest		✓	✓	✓	✓	✓
osCltData_GetSectionChunks		✓	✓	✓	✓	✓
osCltData_OpenTeamedObjectDataSession		✓	✓	✓	✓	✓
osCltData_TeamedObjectGetExtentRegions		✓	✓	✓	✓	✓
osCltData_GetClientThroughputStatistics		✓	✓	✓	✓	✓
osCltData_SeekToReadParentExtents				✓	✓	✓
osCltData_ReadParentExtent				✓	✓	✓

Protocol v5 returns Federated (Teaming) values in structures but they will always be false.

3.1.1 Opening a Catalyst Item Data Session

Catalyst Item Data Sessions are required when writing to or reading from Catalyst Items. In many cases it will be preferential to create wrapper functions around the `osCltData_OpenObjectDataSession` and `osCltData_OpenTeamedObjectDataSession` operations, which are able to manage the resource allocation and retry logic.

Before opening a Catalyst Item Data Session it is recommended that the backup application checks the `IsTeamed` flag for the Catalyst Store being accessed. If the `IsTeamed` flag is true, the backup application must open a command session using `osCltData_OpenTeamedObjectDataSession` whereas if the `IsTeamed` flag is false the backup application should open the command session using `osCltData_OpenTeamedObjectDataSession`.

When opening a data session via `osCltData_OpenObjectDataSession` the client must specify logging parameters. The client should specify the log session ID (`OSCLT_sSessionLogIdType`), log path (`OSCLT_sPathnameType`), log level (`osClt_SessionLogLevel`) and maximum log file size.

3.1.1.1 Code Example – Open an Item

```
int objSDKPrv_OpenObject(const char *pServerAddress, const char *pStoreKey,
const char *pObjectKey, bool writeMode, uint64_t seekToOffset,
OSCLT_sSessionHandleType *pDataSessionHandle, OSCMN_IdType *pJobID)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;

    OSCMN_sKeyType objKey;
    OSCMN_sKeyType storeKey;
    OSCMN_sAddressPortType serverAddr;
    OSCMN_sCredentialsType cred;
    OSCMN_sServerPropertiesType serverProperties;
    OSCMN_sObjectStoreType objectStore;
    OSCMN_sObjectType object;
    OSCLT_sSessionLogDescriptorType sessionLogDescriptor;
    OSCLT_sSessionOptionsType sessionOptions;
    OSCMN_sStringType jobReference;

    // <<Memset structs>>
    // <<Set logging | See 'Error Logging' section>>
    // <<Set credentials | See 'Client Authentication' section>>
    // <<Set server address | Setting data port>>
    serverAddr.PortNumber = OSCLT_DEFAULT_DATA_PORT;

    strncpy(jobReference.String,
            "A Job Reference Id",
            sizeof(jobReference.String));
    if (jobReference.String[sizeof(jobReference.String) - 1])
    {
        printf("Error : Unable to set Job Reference %s\n",
            jobReference.String);
        returnStatus = OSCLT_ERR_INTERNAL_ERROR;
        return returnStatus;
    }
    jobReference.StringSize = strlen(jobReference.String);

    strncpy(storeKey.String,
            pStoreKey,
            sizeof(storeKey.String));
    if (storeKey.String[sizeof(storeKey.String) - 1])
    {
        printf("Error : Unable to set Store Key %s\n", storeKey.String);
        returnStatus = OSCLT_ERR_INTERNAL_ERROR;
        return returnStatus;
    }
    storeKey.StringSize = strlen(storeKey.String);

    strncpy(objKey.String,
            pObjectKey,
            sizeof(objKey.String));
    if (objKey.String[sizeof(objKey.String) - 1])
    {
        printf("Error : Unable to set Object Key %s\n", objKey.String);
        returnStatus = OSCLT_ERR_INTERNAL_ERROR;
        return returnStatus;
    }
    objKey.StringSize = strlen(objKey.String);

    // Open data session
    callStatus = osCltData_OpenObjectDataSession (&sessionLogDescriptor,
                                                  &sessionOptions,
                                                  &serverAddr,
                                                  &cred,
                                                  &storeKey,
                                                  &objKey,
                                                  &jobReference,
                                                  pDataSessionHandle,
                                                  &serverProperties,
                                                  &objectStore,
                                                  &object,
                                                  pJobID);
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Unable to open data session to object '%s' in "
            "store '%s', error : %s.\n", pObjectKey, pStoreKey,
        objSDKPrv_ErrorNumToString(callStatus));
        returnStatus = callStatus;
        return returnStatus;
    }
    return returnStatus;
}
```

For debug purposes the client is recommended to expose a means of increasing the log level and log file size, on a temporary basis. The levels of available log debug are:

```
OSCLT_LOG_LEVEL_EXTENDED_DEBUG
OSCLT_LOG_LEVEL_DEBUG
OSCLT_LOG_LEVEL_TRACE
OSCLT_LOG_LEVEL_INFO
OSCLT_LOG_LEVEL_QUIET
OSCLT_LOG_LEVEL_ERROR
```

During development it is recommended the client be run in OSCLT_LOG_LEVEL_INFO mode, but the level should be set to OSCLT_LOG_LEVEL_ERROR before shipping to customers. See comment on page 24.
If an Open Catalyst Item Data Session operation does not return OSCLT_ERR_SUCCESS then it will free up all resources it may have used (sockets, mallocs, etc). osCltData_CloseObjectDataSession does not need to be called for that session, and must not be called as the Session Handle (out parameter) will be invalid (see above). Using the session handle parameter will almost certainly result in a segv.

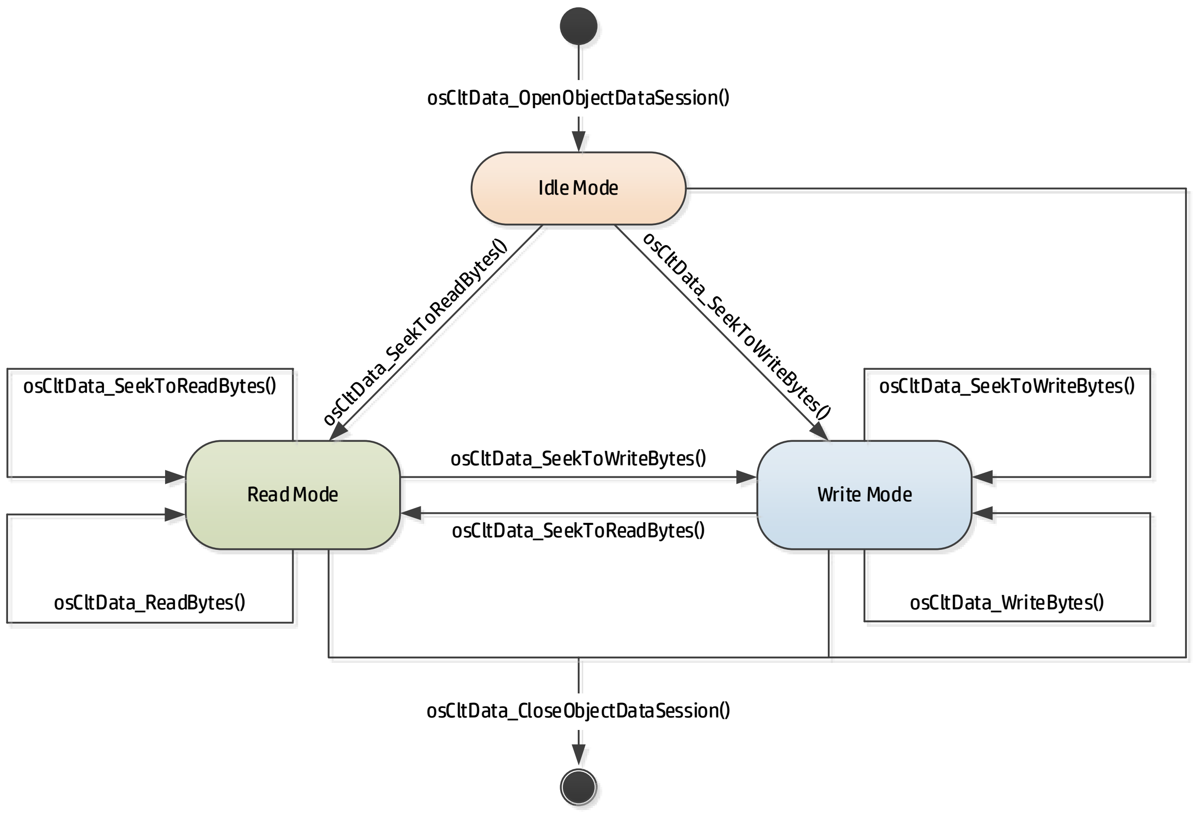

osCltData_OpenObjectDataSession()
Idle Mode
osCltData_SeekToReadBytes()
osCltData_SeekToWriteBytes()
osCltData_SeekToReadBytes()
osCltData_SeekToWriteBytes()
Read Mode
osCltData_SeekToWriteBytes()
osCltData_SeekToReadBytes()
Write Mode
osCltData_ReadBytes()
osCltData_WriteBytes()
osCltData_CloseObjectDataSession()

Figure 10 - Catalyst Item Data Session State Diagram

Full details of the Catalyst Item data session operations can be found in OSCLT_Interface.h.

3.1.2 Catalyst Item Data Session Retries

Catalyst Servers support a limited number of concurrent Catalyst Item data sessions. An indicator to the number of available Catalyst Item data sessions is returned by osCltCmd_GetServerProperties in FreeDataSessions and additionally the ObjectLockResourcesUnavailable flag will indicate whether the Catalyst Server has available resources. These values should only be treated as a guide however as their state could change before the Catalyst Item data session is successfully opened.

If a Catalyst Item data session fails to open with an error of either OSCLT_ERR_MAXIMUM_SESSIONS or OSCLT_ERR_MAXIMUM_LOCKS, it is recommended that the backup application should internally queue the backup job to run again at a later time. Additional logic could be included in the backup application to retry the job as it detects other backup jobs have completed or when the `FreeDataSessions` property and `ObjectLockResourcesUnavailable` flag indicate resources are now available.

3.1.3 Closing a Catalyst Item Data Session

Users of the API are expected to be responsible citizens and close sessions when they have finished with them. Only once a Catalyst Item Data Session to a Catalyst Item is closed is the Catalyst Item lock freed so that it may be accessed by another process, such as the copy engine.

For both Catalyst Item Data Sessions and Federated Catalyst Item Data Sessions, `osCltData_CloseObjectDataSession` should be called to close the session.

If `osCltData_CloseObjectDataSession` returns an error, it will still free up all resources used by that session. Close Session does not need to be called again for that session, and must not be called as the Session Handle will have been invalidated by the Close Session operation. Using the session handle parameter after a close will almost certainly result in a segv.

Catalyst Item Data Sessions which are idle and are not closed will timeout after 4 hours of inactivity and will be forcibly closed by the Catalyst server. This timeout ensures that Catalyst Items do not become locked indefinitely. If the backup application is expected to have known idle periods of longer than 4 hours then they should implement keep alive functionality to keep the data session active. An example keep alive would be to issue an `osCltData_Flush` after 30 minutes of data session inactivity. The application is responsible for maintaining such a counter between operations.

3.1.3.1 Code Example – Close an Item

```
int objSDKPrv_CloseObject(OSCLT_sSessionHandleType *pDataSessionHandle)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;
    OSCMN_sObjectDataJobType objectDataJob;

    memset(&objectDataJob,
           0,
           sizeof(objectDataJob));

    callStatus = osCltData_CloseObjectDataSession(pDataSessionHandle,
                                                   &objectDataJob);
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Unable to close data session, error : %s.\n",
            objSDKPrv_ErrorNumToString(callStatus));
        returnStatus = callStatus;
        return returnStatus;
    }
    return returnStatus;
}
```

3.1.4 Federated Catalyst Item Data Sessions

A Federated Catalyst Item data session provides the Catalyst client with a data path to Catalyst Items in Federated Catalyst Stores. Using this connection the backup software can write and read in the same way as a regular Catalyst Store.

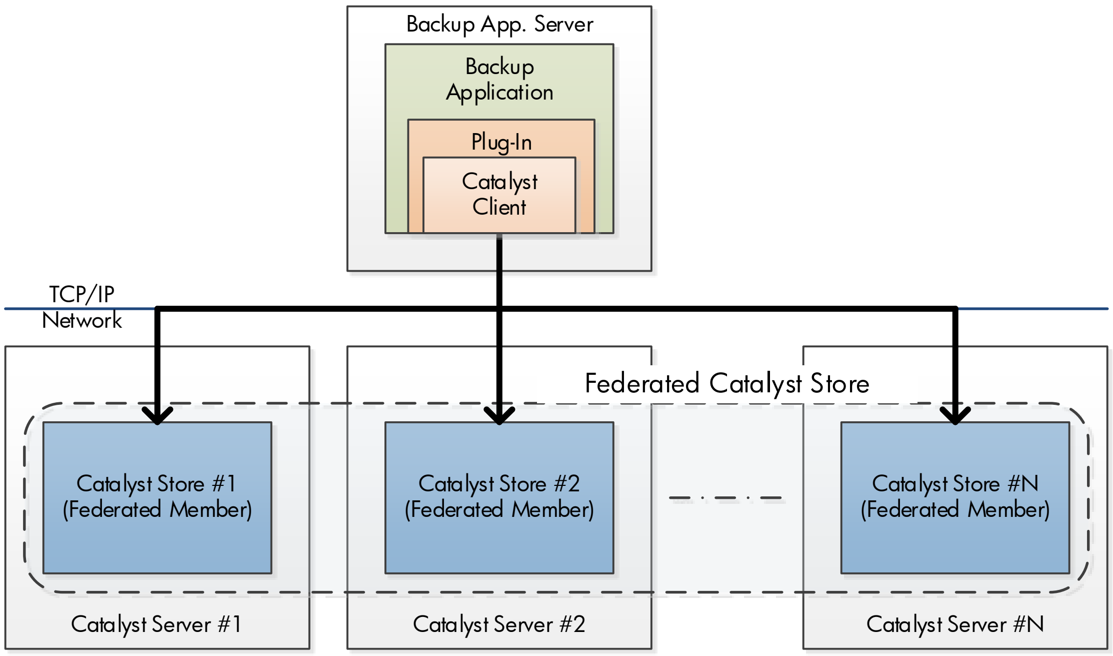


Figure 11 - Federated Catalyst Data Session to all Federated Members

3.1.4.1 Bidding and Routing

Each team member is an independent Catalyst Store with its own backend deduplication store and associated sparse index. In order to maintain deduplication across multiple independent stores a suitable Catalyst routing algorithm must be used to ensure similar data is always sent to the same backend StoreOnce node and that unmatched data is load balanced across the remaining storage. This is handled within the Catalyst Client and is not the responsibility of a backup application.

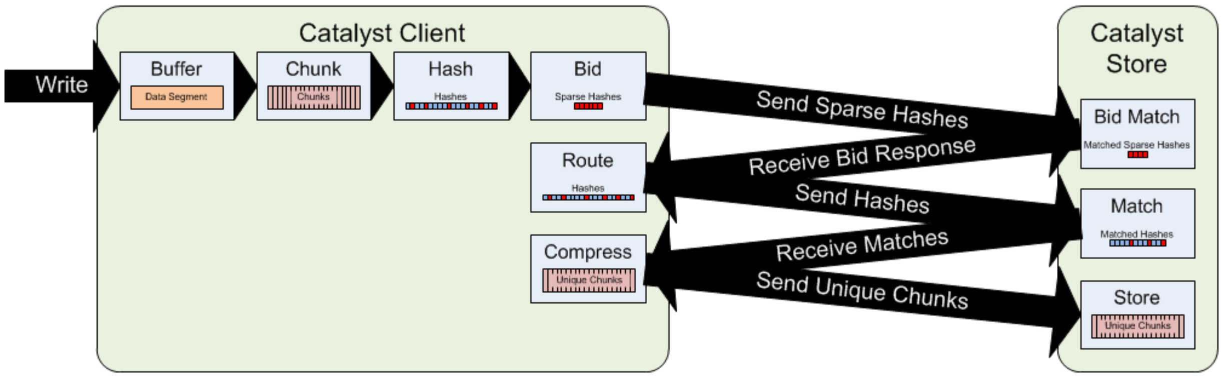


Figure 12 - Bidding and Routing

3.1.4.2 Code Example – Open a Teamed Item

Federated Data Sessions are similar in their implementation to regular Data Sessions with additional structures for `OSCMN_sObjectDataJobTeamingType` and `OSCLT_sTeamMemberConnectionStatusType`.

```
int objSDKPrv_OpenTeamedObject(const char *pServerAddress, const char
*pStoreKey, const char *pObjectKey, bool writeMode, uint64_t seekToOffset,
OSCLT_sSessionHandleType *pDataSessionHandle)
{
    //<<Set up as in objSDKPrv_OpenObject>>
    OSCMN_sObjectDataJobTeamingType objectDataJobTeaming;
    OSCLT_sTeamMemberConnectionStatusType teamMemberConnectionStatus;

    memset(&objectDataJobTeaming,
           0,
           sizeof(objectDataJobTeaming));

    memset(&teamMemberConnectionStatus,
           0,
           sizeof(teamMemberConnectionStatus));

    callStatus = osCltData_OpenTeamedObjectDataSession(
                                        &sessionLogDescriptor,
                                        &sessionOptions,
                                        &address,
                                        &cred,
                                        &objStoreKey,
                                        &objKey,
                                        &jobReference,
                                        pDataSessionHandle,
                                        &teamMemberConnectionStatus,
                                        &serverProperties,
                                        &objectStore,
                                        pObject,
                                        &objectDataJobTeaming);
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Unable to open data session to object '%s' in "
               "store '%s', error : %s.\n", pObjectKey, pStoreKey,
                objSDKPrv_ErrorNumToString(callStatus));
        returnStatus = callStatus;
        return callStatus;
        // Can also check the teamMemberConnectionStatus structure to
        // determine on which team member(s) the error occurred
    }
    return returnStatus;
}
```

3.1.4.3 Multi-reader Functionality

From StoreOnce 3.15 it is possible to have multiple read concurrent data sessions open to a single Catalyst object. Prior to this release, a request to open a subsequent data session would have resulted in `OSCLT_ERR_OBJECT_LOCK_FAILED`. Multiple sessions can only be opened when only read operations have been performed on an object. Once a clone or write operation is performed the object will be exclusively locked to that single data session.

Multi-reader functionality is provided to support the use case where multiple copies or multiple restores need to be performed from an object.

Applications can query whether a Catalyst store supports multiple readers by querying the `SupportMultipleObjectOpeners` in `OSCMN_sObjectStoreCapabilitiesType` which is returned by a call to `osCltCmd_ListObjectStores`.

3.2 Section Summary and recommendated best practises

- The number of available Catalyst data sessions is limited and is different across the StoreOnce appliance range. Implement data session retry logic when `OSCLT_ERR_MAXIMUM_SESSIONS`.
- Always close any data session opened by the application.
- There are 2 types of data sessions; teamed data sessions and non-teamed data sessions.
- Set an appropriate logging level when opening a data session.
- A data session will be closed on the Catalyst server if it is idle for 4 hours. Catalyst integrators should implement a keep alive mechanism if it is expected to have long periods of idle operation.

Section 4: StoreOnce Catalyst Transfer Policies

4.1 Bandwidth Optimized Data Transfer

One of the many features of StoreOnce Catalyst is the ability to be able to specify that data transferred between a Catalyst Client and a Catalyst Server is done in a bandwidth efficient way. Bandwidth efficiency is achieved by performing some of the deduplication process within the Catalyst Client. This processing of the data ensures that only unique chunks of data are transferred between the Catalyst Client and the Catalyst Server, thus reducing network load and potentially increasing overall throughput.

Data transfer that does not utilise the bandwidth efficient transfer mode is known as High Bandwidth and data transfer that does utilise the bandwidth efficient transfer mode is known as Low Bandwidth.

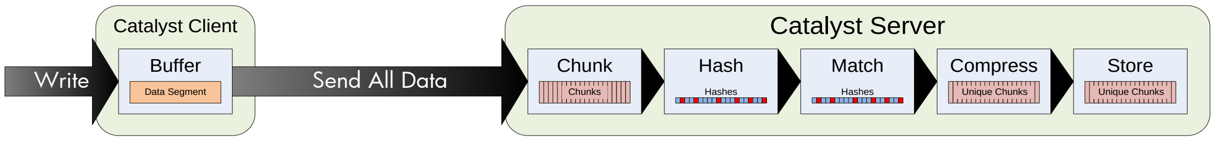


Figure 13 - High Bandwidth Data Transfer Process

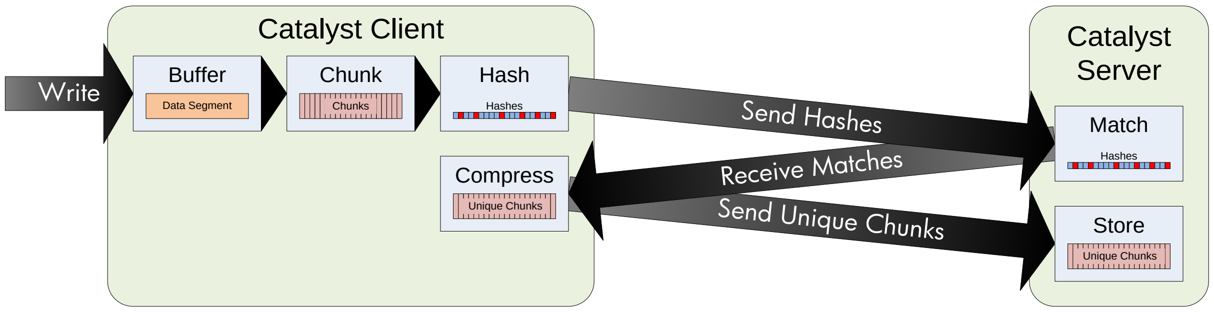


Figure 14 - Low Bandwidth Data Transfer Process

Catalyst Stores are configured with a primary and secondary transfer policy. These policies specify which data transfer modes may be used by Catalyst Clients that open data sessions to Objects within a Catalyst Store. Table 5 shows that if a transfer mode is not selected in either the primary or secondary transfer policy, the Catalyst Client will not be permitted to utilise this transfer mode.

Table 5 – Permitted Transfer Modes

Catalyst Store Settings		**Catalyst Client Data Transfer**	
Primary Transfer Policy	**Secondary Transfer Policy**	**High Bandwidth**	**Low Bandwidth**
High Bandwidth	Low Bandwidth	✓	✓
Low Bandwidth	High Bandwidth	✓	✓
High Bandwidth	High Bandwidth	✓	✗
Low Bandwidth	Low Bandwidth	✗	✓

Bandwidth optimized data transfer between the Catalyst Client and the Catalyst Server is only available when performing the `osCltData_WriteBytes` operation. During the `osCltData_ReadBytes` operation, all data transfer is done in a high bandwidth mode. Selecting either high bandwidth or low bandwidth data transfer is done each time the `osCltData_SeekToWriteBytes` operation is called. This allows callers to select a data transfer mode appropriately for the data to be transferred and also the ability to change the mode based on other factors, such as load on the media server, data transfer statistics or data type.

If the client application does not expose options allowing the user to specify a bandwidth option the application should query the Catalyst store properties (`OSCMN_sObjectStoreType`) to get the `OSCMN_sWritePolicyType`. This will query the transfer policy value set within the StoreOnce GUI. Setting the transfer policy via the StoreOnce GUI is the recommended approach,

The available OSCMN_eWriteBandwidthModeType enums are:

- OSCMN_WRITEBANDWIDTHMODE_HIGH_ONLY
- OSCMN_WRITEBANDWIDTHMODE_LOW_ONLY
- OSCMN_WRITEBANDWIDTHMODE_BOTH_DEFAULT_HIGH
- OSCMN_WRITEBANDWIDTHMODE_BOTH_DEFAULT_LOW

Bandwidth optimized data transfer also caters for performing data transfer over a WAN connection, for example between a remote small office and a data centre. When performing data transfer over a WAN, it is recommended that the maximum data buffer size is used. This maximum permissible size is reported as `MaximumLowBandwidthDataBufferSize` when calling `osCltCmd_GetClientProperties`.

```
OSCLT_sSessionOptionsType sessionOptions;
memset(&sessionOptions,
       0,
       sizeof(sessionOptions));
sessionOptions.PayloadChecksumsDisabled = true;
```

It should be noted that Federated Catalyst only supports Low Bandwidth.

```
OSCMN_sWritePolicyType writePolicy;
memset(&writePolicy,
       0,
       sizeof(writePolicy));
writePolicy.WriteBandwidthMode = OSCMN_WRITEBANDWIDTHMODE_BOTH_DEFAULT_LOW;
```

4.1.1 Target Deduplication

Target Deduplication is a High Bandwidth Data Transfer where all data is sent from the Media Server to the StoreOnce Backup System to be deduplicated. As deduplication has high resource requirements it may be better suited to offload the deduplication process to the StoreOnce Backup System in certain use cases with lower powered Clients and Media Servers or clients already under load.

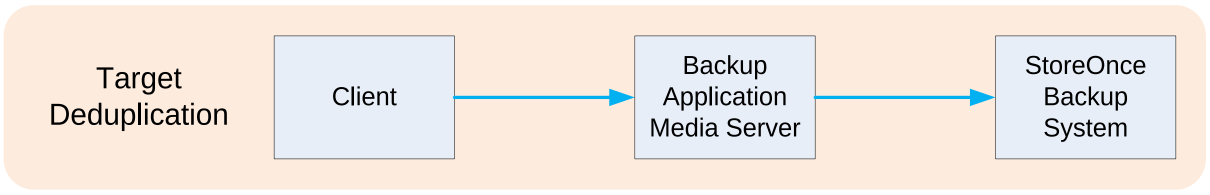


Figure 15 - Target Deduplication process

4.1.2 Backup Server Deduplication

Backup Server Deduplication is a High Bandwidth Data Transfer from the Client to the Backup Application Media Server where the deduplication occurs before performing a Low Bandwidth Data Transfer between the Media Server and the StoreOnce Backup System. As deduplication has high resource requirements it may be better suited to offload the deduplication process to a Media Server which can then transfer only unique data to a StoreOnce Backup System.

This low bandwidth transfer policy is the recommended default for Catalyst integrations.

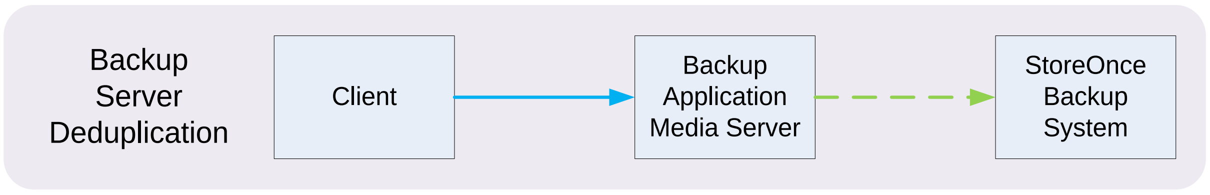


Figure 16 - Backup Server Deduplication process

4.1.3 Application Source Deduplication

Application Source Deduplication is a Low Bandwidth Data Transfer where the Client is capable of performing the deduplication process itself and sending the deduplicated data to the StoreOnce Backup System, removing the need for multiple Media Servers.

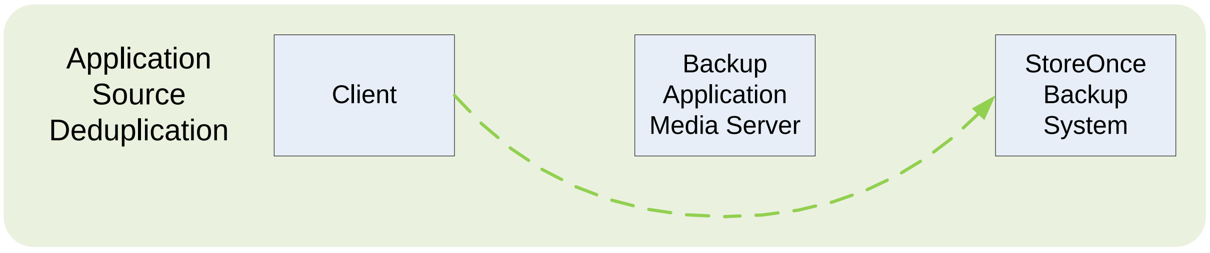

Application Source Deduplication
Client
Backup Application Media Server
StoreOnce Backup System

Figure 17 - Application Source Deduplication

4.2 Catalyst Network Compression and Checksums

By default, data sent via a Catalyst data session will be compressed when sent over the network and will have checksums created. This is to account for the worst case scenario where customers are backing up over a Wide Area Network (WAN), since bandwidth will be limited and WAN checksums have low levels of integrity. It is recommended to expose the ability to enable and disable payload checksums to customers, for example via a config file or via a backup application GUI option.

The recommended values are Catalyst compression and checksums are as follows:

	Compression	Network Checksums
Catalyst over FC	DISABLE	DISABLE
Low Bandwidth (WAN)	ENABLE	ENABLE
Low Bandwidth (LAN)	DISABLE	DISABLE
High Bandwidth (LAN only)	DISABLE	DISABLE

Where possible, the simplest way to achieve the above is to expose a method to allow the user to specify whether they are transferring over WAN; if not, this checksums and compression can be disabled.

4.3 Section Summary and recommendated best practises

- Low bandwidth and High bandwidth data transfer policies are available
- Select high or low bandwidth based on the Catalyst store property in the StoreOnce GUI
- Expose a method for Catalyst network compression and checksums to be enabled disabled

Section 5: StoreOnce Catalyst Error Reporting

5.1 Error Logs

5.1.1 Catalyst Client Log Files

The Catalyst Client provides a logging facility that must be configured for each command or data session. The logging facility takes the following parameters:

- Log File Location
- Maximum Log File Size
- Log Level
- Session Log ID

5.1.1.1 Log File Location

The log file is written in a thread safe manner and therefore a single log file may be specified for all command and data sessions of an individual client. It is recommended that the Catalyst Client log file be co-located with any other log files being generated by the calling application.

5.1.1.2 Maximum Log file Size

A maximum size for the log file should be provided and the Catalyst Client will ensure that the log file does not exceed this size. The maximum log file size must be consistent between all callers who specify the same log file. Failure to do so will result in the existing contents of the log file being truncated. The recommended log file size during development is 200MB. In a production environment it is recommended that the maximum log file size is reduced to 10MB.

Log file size should be consistent within a session to avoid unexpected truncation.

The log file size provided for the log file should be externally configurable to allow for greater verbosity in the unlikely event that a failure cannot be identified using the default log file size.

The the backup application creates a support ticket if should include the Catalyst log files, including the .cofc file.

5.1.1.3 Log Level

The verbosity of the information included in the log file should be specified. The recommended log level during development is `OSCLT_LOG_LEVEL_INFO` as this will contain enough information to assist with debug investigations without logging unnecessary additional information. In a production environment it is recommended that the log level is changed to `OSCLT_LOG_LEVEL_ERROR` so that logging only occurs during unexpected behaviour.

The log level provided for the log file should be externally configurable to allow for greater verbosity in the unlikely event that a failure cannot be identified using the default log level. It is recommended to expose the ability to configure Catalyst log levels independently of the backup application log level. All log levels exposed in the Catalyst API should be configurable.

Increasing the log level to OSCLT_LOG_LEVEL_DEBUG and above will have a significant impact on performance.

5.1.1.4 Session Log ID

With multiple operations being logged to the same Catalyst Client log file it is important to be able to correlate between external operations and the API calls logged. To do this, the Session Log ID has been provided. The Session Log ID is a string passed by the caller that will prepend any log entries written during the duration of the session. If the caller provides a Session Log ID that can be associated to an operation within the calling application, it will be possible to align log file entries.

For example, if the calling application were to have a unique identifier for a backup job such as a Job ID. This Job ID could be passed as the Session Log ID, and therefore all entries in the Catalyst Client log file that are associated with the backup job will be prepended with the unique Job ID of the backup.

5.1.2 Catalyst over Fibre Channel Log File

Events logged whilst using Catalyst over Fibre Channel are saved to a separate log file with the log file name appended with '.cofc' which is identical in format and use to a normal log file. The .cofc file will be created next to the default standard Catalyst log.

If the backup application creates a support ticket if should include the Catalyst log files, including the .cofc file.

5.1.3 Complete Code Example – Error Logging

Logging parameters are defined via an OSCLT_sSessionLogDescriptorType struct, which is passed in when opening command and data sessions. The following is an example of constructing the struct:

```
#define OBJSDK_DEFAULT_LOG_SIZE ( 10 * MB)
#define OBJSDK_DEFAULT_LOG_PATH "ObjectStoreSDK.log"

OSCLT_eLogLevelType OBJSDK_LOG_LEVEL = OSCLT_LOG_LEVEL_INFO;
OSCLT_sPathnameType logPath;
OSCLT_sSessionLogDescriptorType sessionLogDescriptor;

memset(&sessionLogID,
       0,
       sizeof(sessionLogID));

memset(&sessionLogDescriptor,
       0,
       sizeof(sessionLogDescriptor));

memset(&logPath,
       0,
       sizeof(logPath));
```

```
strncpy(sessionLogID.String,
        OBJSDK_CMD_SESSION_ID,
        sizeof(sessionLogID.String));

if (sessionLogID.String[sizeof(sessionLogID.String) - 1])
{
    returnStatus = OSCLT_ERR_INTERNAL_ERROR;
    return returnStatus;
}
sessionLogID.StringSize = strlen(sessionLogID.String);

strncpy(logPath.String,
        OBJSDK_DEFAULT_LOG_PATH,
        sizeof(logPath.String));

if (logPath.String[sizeof(logPath.String) - 1])
{
    returnStatus = OSCLT_ERR_INTERNAL_ERROR;
    return returnStatus;
}
logPath.StringSize = strlen(logPath.String);

sessionLogDescriptor.SessionLogId = sessionLogID;
sessionLogDescriptor.LogLevel = osClt_SessionLogLevel(OBJSDK_LOG_LEVEL);
// Should be run time configurable for customers to alter
sessionLogDescriptor.LogFile = logPath;
sessionLogDescriptor.MaxLogfileSize = OBJSDK_DEFAULT_LOG_SIZE;
```

5.1.4 Backup application Error Reporting

The aim of client error reporting and checking should be to allow customers to quickly and easily self-diagnose failures. By doing so, customers can quickly rectify mistakes or if technical support is required turnaround time can be reduced when errors are easily identifiable. The Catalyst client therefore implements a set of functions (see Table 6 – Error reporting supported protocols) which can be used to convert error numbers into error codes that should be translated to the customer as a human readable error message. Where errors are logged to file it is recommended that the error code is passed through these funtions and the error code be logged along with the error number.

The supported error convertion functions are as follows:

Table 6 – Error reporting supported protocols

Command	Protocol v4	Protocol v5	Protocol v6	Protocol v7	Protocol v8	Protocol v9
`osClt_ErrorNumToString`	✓	✓	✓	✓	✓	✓
`osClt_JobReasonToString`				✓	✓	✓
`osClt_ConfigStateToString`				✓	✓	✓

5.1.4.1 Code Example – Converting Error Numbers to Strings

```
const char * objSDKPrv_ErrorNumToString(int errorNum)
{
      return osClt_ErrorNumToString(errorNum);
}
// Example output
printf("Error number: %d, error string: %s\n", -1404,
        objSDKPrv_ErrorNumToString(-1404));
printf("Error number: %d, error string: %s\n", -1900,
        objSDKPrv_ErrorNumToString(-1900));
printf("Error number: %d, error string: %s\n", -2004,
        objSDKPrv_ErrorNumToString(-2004));
```

See `OSCLT_ErrorCodes.h` for the full error numbers/codes.

Within the shipping application, for debug / support purposes the client is recommended to expose a means of increasing the log level and log file size, on a temporary basis. This may be within a configuration file or within the application interface. The levels of available logging are:

- OSCLT_LOG_LEVEL_EXTENDED_DEBUG
- OSCLT_LOG_LEVEL_DEBUG
- OSCLT_LOG_LEVEL_TRACE
- OSCLT_LOG_LEVEL_INFO
- OSCLT_LOG_LEVEL_QUIER
- OSCLT_LOG_LEVEL_ERROR

During development it would be useful for the client to run in `OSCLT_LOG_LEVEL_INFO` mode, but the level should be set to `OSCLT_LOG_LEVEL_ERROR` before shipping to customers. See comment on page 24.

5.2 Section Summary and recommendated best practises

- The client application should set the default logging level as OSCLT_LOG_LEVEL_ERROR
- A method of increasing the log level should be exposed within the client application in order allow triage of errors. The log file size should also be exposed as a customer configurable parameter.

Section 6: StoreOnce Catalyst Client and Server

6.1 Catalyst Clients

6.1.1 Getting Client Properties

Details of the client library being used by the backup application may be acquired by calling `osClt_GetClientProperties`. This includes version information and details of the client libraries capabilities. Unlike all other command session operations, `osClt_GetClientProperties` does not require a command session handle be passed when calling.

It is recommended that backup applications retrieve this information from the client when they first load the client library. The returned information includes for example the minimum and maximum data buffer size that can be used with the client library. These capabilities should be stored in an internal structure so that they may be directly referenced by other operations during use.

The capabilities returned via a call to `osClt_GetClientProperties` over the different protocol versions:

Table 7 - Client Capabilities Protocol Support

Client Capability (OSCLT_sClientCapabilitiesType)	**Protocol v4**	**Protocol v5**	**Protocol v6**	**Protocol v7**	**Protocol v8**	**Protocol v9**
`SupportUserDataSizeQuotas`		✓	✓	✓	✓	✓
`SupportDedupedDataSizeOnDiskQuotas`		✓	✓	✓	✓	✓
`SupportPerStoreEncryptedStorage`		✓	✓	✓	✓	✓
`SupportClientPermissions`		✓	✓	✓	✓	✓
`SupportDedupeStoreSelection`		✓	✓	✓	✓	✓
`SupportObjectCopy`		✓	✓	✓	✓	✓
`SupportObjectCopyDestinationCredentials`						✓
`SupportDataSessionModesForStorageModeVariableBlock`		✓	✓	✓	✓	✓
`SupportDataSessionModesForStorageModeFixedBlock`		✓	✓	✓	✓	✓
`SupportDataSessionModesForStorageModeNoDedupe`		✓	✓	✓	✓	✓
`SupportJobReference`		✓	✓	✓	✓	✓
`SupportClientCredentialsPassword`		✓	✓	✓	✓	✓
`SupportListIncreasingOrder`		✓	✓	✓	✓	✓
`SupportKeyTaglistExtendedCharSet`		✓	✓	✓	✓	✓
`SupportNegativeTaglistFilter`		✓	✓	✓	✓	✓
`SupportModifyServerProperties`		✓	✓	✓	✓	✓
`SupportClientPermissionsManagement`		✓	✓	✓	✓	✓
`SupportListCounts`		✓	✓	✓	✓	✓
`SupportSecureErase`				✓	✓	✓
`SupportTeaming`		✓	✓	✓	✓	✓
`SupportTeamingObjectCopy`		✓	✓	✓	✓	✓
`SupportTeamingDataSessionModesForStorageModeVariableBlock`		✓	✓	✓	✓	✓
`SupportTeamingDataSessionModesForStorageModeFixedBlock`		✓	✓	✓	✓	✓
`SupportTeamingDataSessionModesForStorageModeNoDedupe`		✓	✓	✓	✓	✓
`SupportTeamingAddTeamMembers`		✓	✓	✓	✓	✓
`SupportTeamingRemoveTeamMembers`		✓	✓	✓	✓	✓

`SupportTeamingDegradedOperation`		✓	✓	✓	✓	✓
`SupportTeamingRedundancy`		✓	✓	✓	✓	✓
`MinimumHighBandwidthDataBufferSize`	✓	✓	✓	✓	✓	✓
`MaximumHighBandwidthDataBufferSize`	✓	✓	✓	✓	✓	✓
`MinimumLowBandwidthDataBufferSize`	✓	✓	✓	✓	✓	✓
`MaximumLowBandwidthDataBufferSize`	✓	✓	✓	✓	✓	✓
`MinimumRawReadWriteNominalSectionSize`		✓	✓	✓	✓	✓
`MaximumRawReadWriteNominalSectionSize`		✓	✓	✓	✓	✓
`MinimumRawReadWriteOverlappingSections`		✓	✓	✓	✓	✓
`MaximumRawReadWriteOverlappingSections`		✓	✓	✓	✓	✓
`MinimumProtocolVersion`		✓	✓	✓	✓	✓
`MaximumProtocolVersion`		✓	✓	✓	✓	✓
`MinimumSessionThreads`		✓	✓	✓	✓	✓
`DefaultSessionThreads`		✓	✓	✓	✓	✓
`MaximumSessionThreads`		✓	✓	✓	✓	✓
`SupportObjectCopyUsingRawReadWrite`		✓	✓	✓	✓	✓
`SupportRawReadWrite *`	✓					
`SupportObjectCopy *`	✓					
`SupportCloneExtent *`	✓					
`SupportLowBWForObjectStorageModeVariableBlockDedupe *`	✓					
`SupportLowBWForObjectStorageModeFixedBlockDedupe *`	✓					
`MinimumLowBandwidthChunkerWriteThreads *`	✓					
`MaximumLowBandwidthChunkerWriteThreads *`	✓					

* Deprecated in v5. Protocol v5 will return Federated (Teaming) values in structures but they will always be false.

For a full list of reported client capabilities, see `osCLT_sClientCapabilitiesType` in `OSCLT_Interface.h`

`osClt_GetClientProperties` can also be used to query the client software version, as shown in the code example below.

6.1.1.1 Code Example – Getting Client Properties

```
OSCLT_sClientPropertiesType clientProp;
memset(&clientProp,
       0,
       sizeof(clientProp));

callStatus = osClt_GetClientProperties(&clientProp);
if (callStatus != OSCLT_ERR_SUCCESS)
{
    printf("Error : Failed to get client properties : %s.\n",
        objSDKPrv_ErrorNumToString(callStatus));
    return 0;
}
clientCap = clientProp.ClientCapabilities;

printf("Client Software Version \t\t: '%s'\n",
    clientProp.ClientSoftwareVersion.String);

printf("Minimum High Bandwidth Data Buffer Size : %u\n",
    clientCap.MinimumHighBandwidthDataBufferSize);

printf("Maximum High Bandwidth Data Buffer Size : %u\n",
    clientCap.MaximumHighBandwidthDataBufferSize);
```

6.2 Catalyst Servers

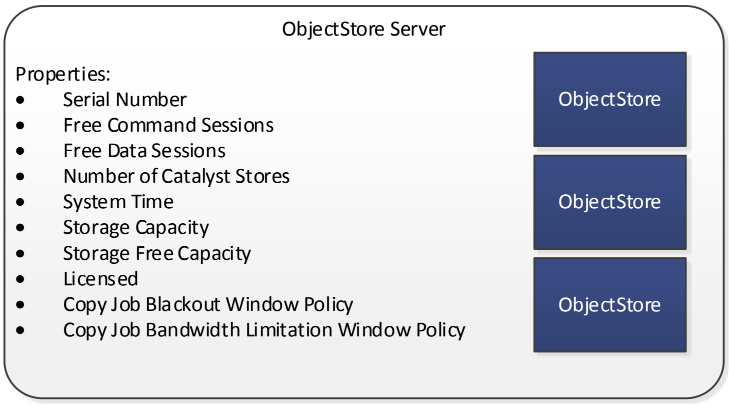


Figure 18 – Catalyst Server Properties

6.2.1 Getting Server Properties

The Catalyst Server properties provide a wealth of information about a storage device, including its capabilities, status and capacity.

When a Catalyst Server is first contacted by a backup application, it is recommended that server properties are requested so that the backup application can identify the capabilities and limits of the storage device. For example, the maximum number of objects that can be listed on a single `osCltCmd_ListObjects` operation. These capabilities should be stored in an internal structure so that they may be directly referenced by other operations during use.

Server properties returned via a call to `osCltCmd_GetServerProperties` include:

Table 8 - Server Properties Protocol Support

Server Property (OSCMN_sServerPropertiesType)	**Protocol v4**	**Protocol v5**	**Protocol v6**	**Protocol v7**	**Protocol v8**	**Protocol v9**
`SoftwareVersion`	✓	✓	✓	✓	✓	✓
`SerialNumber`	✓	✓	✓	✓	✓	✓
`ManagementAddress`			✓	✓	✓	✓
`NegotiatedProtocolVersion`	✓	✓	✓	✓	✓	✓
`FreeCommandSessions`	✓	✓	✓	✓	✓	✓
`FreeOriginCopySessions`		✓	✓	✓	✓	✓
`OriginCopySessionsLimit`		✓	✓	✓	✓	✓
`FreeDataSessions`	✓	✓	✓	✓	✓	✓
`DataSessionsLimit`		✓	✓	✓	✓	✓
`ObjectLockResourcesUnavailable`	✓	✓	✓	✓	✓	✓
`NumObjectStores`	✓	✓	✓	✓	✓	✓
`FreeObjectStores`	✓	✓	✓	✓	✓	✓
`FreeDedupeStores`	✓	✓	✓	✓	✓	✓
`SystemTimeUTC`	✓	✓	✓	✓	✓	✓
`SystemTimeLocal`	✓	✓	✓	✓	✓	✓
`DiskCapacity`	✓	✓	✓	✓	✓	✓
`DiskSpaceFree`	✓	✓	✓	✓	✓	✓
`IsClientLicensed`	✓	✓	✓	✓	✓	✓
`IsClientTeamedLicensed`			✓	✓	✓	✓
`IsClientCoFCLicensed`			✓	✓	✓	✓
`CanClientCreateObjectStores`		✓	✓	✓	✓	✓
`CanClientSetServerProperties`		✓	✓	✓	✓	✓
`CanClientManageClientPermissions`		✓	✓	✓	✓	✓
`ServerCapabilities`	✓	✓	✓	✓	✓	✓
`CopyJobBlackoutPolicy`	✓	✓	✓	✓	✓	✓

CopyJobThrottlePolicy	✓	✓	✓	✓	✓	✓
IsCurrentlyInCopyJobBlackout	✓	✓	✓	✓	✓	✓
CurrentCopyJobThrottleBytesPerSecond	✓	✓	✓	✓	✓	✓
IsClientPermissionsCheckingEnabled		✓	✓	✓	✓	✓
IsServerStorageEncrypted		✓	✓	✓	✓	✓

Capabilities returned in the ServerCapabilities structure include:

Table 9 - Server Capabilities Protocol Support

Server Capabilities (OSCMN_sServerCapabilitiesType)	**Protocol v4**	**Protocol v5**	**Protocol v6**	**Protocol v7**	**Protocol v8**	**Protocol v9**
SupportUserDataSizeQuotas	✓	✓	✓	✓	✓	✓
SupportDedupedDataSizeOnDiskQuotas	✓	✓	✓	✓	✓	✓
SupportPerStoreEncryptedStorage (v4 : SupportEncryptedStorage)	✓	✓	✓	✓	✓	✓
SupportClientPermissions	✓	✓	✓	✓	✓	✓
SupportDedupeStoreSelection	✓	✓	✓	✓	✓	✓
SupportObjectCopy	✓	✓	✓	✓	✓	✓
SupportObjectCopyDestinationCredentials						✓
SupportDataSessionModesForStorageModeVariableBlock		✓	✓	✓	✓	✓
SupportDataSessionModesForStorageModeFixedBlock		✓	✓	✓	✓	✓
SupportDataSessionModesForStorageModeNoDedupe		✓	✓	✓	✓	✓
SupportJobReference		✓	✓	✓	✓	✓
SupportClientCredentialsPassword		✓	✓	✓	✓	✓
SupportListIncreasingOrder		✓	✓	✓	✓	✓
SupportKeyTaglistExtendedCharSet		✓	✓	✓	✓	✓
SupportNegativeTaglistFilter		✓	✓	✓	✓	✓
SupportModifyServerProperties		✓	✓	✓	✓	✓
SupportClientPermissionsManagement		✓	✓	✓	✓	✓
SupportListCounts		✓	✓	✓	✓	✓
SupportSecureErase				✓	✓	✓

SupportTeaming		✓	✓	✓	✓	✓
SupportTeamingObjectCopy		✓	✓	✓	✓	✓
SupportTeamingDataSessionModesForStorageModeVariableBlock		✓	✓	✓	✓	✓
SupportTeamingDataSessionModesForStorageModeFixedBlock		✓	✓	✓	✓	✓
SupportTeamingDataSessionModesForStorageModeNoDedupe		✓	✓	✓	✓	✓
SupportTeamingAddTeamMembers		✓	✓	✓	✓	✓
SupportTeamingRemoveTeamMembers		✓	✓	✓	✓	✓
SupportTeamingDegradedOperation		✓	✓	✓	✓	✓
SupportTeamingRedundancy		✓	✓	✓	✓	✓
MinimumHighBandwidthDataBufferSize		✓	✓	✓	✓	✓
MaximumHighBandwidthDataBufferSize		✓	✓	✓	✓	✓
MinimumLowBandwidthDataBufferSize		✓	✓	✓	✓	✓
MaximumLowBandwidthDataBufferSize		✓	✓	✓	✓	✓
MinimumRawReadWriteNominalSectionSize		✓	✓	✓	✓	✓
MaximumRawReadWriteNominalSectionSize		✓	✓	✓	✓	✓
MinimumRawReadWriteOverlappingSections		✓	✓	✓	✓	✓
MaximumRawReadWriteOverlappingSections		✓	✓	✓	✓	✓
MinimumProtocolVersion		✓	✓	✓	✓	✓
MaximumProtocolVersion		✓	✓	✓	✓	✓
MaximumObjectStoresPerListIteration	✓	✓	✓	✓	✓	✓
MaximumObjectsPerListIteration	✓	✓	✓	✓	✓	✓
MaximumObjectsPerObjectStore	✓	✓	✓	✓	✓	✓
MaximumJobsPerListIteration	✓	✓	✓	✓	✓	✓
MaximumMetadataSizePerObject	✓	✓	✓	✓	✓	✓
MaximumMetadataSizePerListIteration	✓	✓	✓	✓	✓	✓
MaximumDatabaseSizePerObjectStore	✓	✓	✓	✓	✓	✓
MaximumJobRetentionPeriod	✓	✓	✓	✓	✓	✓
MaximumClientsPermissionsPerListIteration		✓	✓	✓	✓	✓
MaximumFixedChunkSize		✓	✓	✓	✓	✓
MaximumOpenParentsPerCloneSession		✓	✓	✓	✓	✓

`MaximumParentObjectExtentsPerCloneRequest`		✓	✓	✓	✓	✓
`MaximumRegionsPerGetExtentRegionsIteration`		✓	✓	✓	✓	✓
`MaximumCommandSessions`		✓	✓	✓	✓	✓
`MaximumOriginCopySessions`		✓	✓	✓	✓	✓
`MaximumDataSessions`		✓	✓	✓	✓	✓

* Protocol v5 will return Federated (Teaming) values in structures but they will always be false.

The resources available including storage utilisation are periodically updated and therefore should not be requested more frequently than once every 5 minutes. If the information is required on a more frequent basis then a cached structure should be held in memory and updated only when the information has aged 5 minutes or a change has been made that will knowingly have made a change to the Catalyst Server Properties.

6.2.2 Setting Server Properties

Server settings are most commonly managed through the web management interface of the StoreOnce appliance. However a StoreOnce appliance, which supports StoreOnce Catalyst v5 or above, may be configured so that some server settings may be remotely managed by software that integrates StoreOnce Catalyst.

The server settings that can be modified are:

- Maximum Concurrent Data and Inbound Copy Jobs
- Maximum Concurrent Outbound Copy Jobs
- Outbound Copy Job Blackout Windows and Blackout Override
- Outbound Copy Job Bandwidth Throttling Windows and General Bandwidth Throttling Limit

Software wishing to provide this remote management of server will need to support the `osCltCmd_SetServerProperties` function.

Support of this function in a backup application is considered advanced integration and is not undertaken by the majority of integrators.

6.2.2.1 Checking Remote Management Capability

A backup application can check whether it has been granted permission to remotely manage server settings by checking the `CanClientSetServerProperties` Boolean returned by `osCltCmd_GetServerProperties`.

If `CanClientSetServerProperties` is true, then the software is able to use the command listed above, otherwise the function will return permission denied when called.

6.2.2.2 Updating Server Properties

A backup application is able to update Catalyst Server properties by modifying the policy structs or setting valid integer values. Please see `OSCMN_sBlackoutPolicyType` and `OSCMN_sThrottlePolicyType` in `OSCMN_Types.h` for more information.

As the `osCltCmd_SetServerProperties` function will update each property, if there is a setting which should remain unchanged it is necessary to pull the existing server properties and resend them with the newly modified properties, merged.

6.2.2.3 Complete Code Example – Setting Server Properties

```
int objSDKPrv_SetServerProp(const char *pServerAddress)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = NULL;
      OSCMN_sServerPropertiesType serverProp;
      OSCMN_sBlackoutPolicyType blackoutPolicy;
      OSCMN_sThrottlePolicyType throttlePolicy;
      uint64_t originCopySessionsLimit = 0;
      uint64_t dataSessionsLimit = 0;

      // <<Memset policy structs>>
```

```
       // <<Open command sessions>>
       // <<Get current server properties>>

       // Get current server properties
       blackoutPolicy = serverProp.CopyJobBlackoutPolicy;
       throttlePolicy = serverProp.CopyJobThrottlePolicy;
       originCopySessionsLimit =
            serverProp.ServerCapabilities.MaximumOriginCopySessions;
       dataSessionsLimit =
            serverProp.ServerCapabilities.MaximumDataSessions;

       // Change blackout policy to enable
       blackoutPolicy.IsBlackoutNowOverrideInPlace = true;
       // Change data sessions limit to 1
       dataSessionsLimit = 4;

       // Set server properties (including unchanged properties)
       callStatus = osCltCmd_SetServerProperties (pCmdSessionHandle,
                                                   &blackoutPolicy,
                                                   &throttlePolicy,
                                                   originCopySessionsLimit,
                                                   dataSessionsLimit);
       if (callStatus != OSCLT_ERR_SUCCESS)
       {
             printf("Error : Failed to set server properties : %s.\n",
             objSDKPrv_ErrorNumToString(callStatus));
             // <<Close command session>>
             return callStatus;
       }
       printf("Blackout Now Override In Place set.\nMax number of Data"
              "And Destination CopyJobs set to 4.\n");
       // <<Close command session>>
       return returnStatus;
}
```

6.3 ObjectStores

This section only gives details about the ObjectStores capabilities. ObjectStores are described in more detail in Section 7: StoreOnce Catalyst Stores.

6.3.1 Getting ObjectStore Capabilities

Details of the ObjectStores use by the backup application may be acquired by calling `osCltCmd_ListObjectStores`. This includes details of the ObjectStores capabilities.

It is recommended that backup applications retrieve this information before starting a backup in order to verify the supported capabilities of the ObjectStore used for the backups. These capabilities should be stored in an internal structure so that they may be directly referenced by other operations during use.

The capabilities returned via a call to `osCltCmd_ListObjectStores` over the different protocol versions:

Table 10 – ObjectStore Capabilities Protocol Support

ObjectStore Capabilities (OSCMN_sObjectStoreCapabilitiesType)	**Protocol v4**	**Protocol v5**	**Protocol v6**	**Protocol v7**	**Protocol v8**	**Protocol v9**
`SupportStorageModeVariableBlockDedupe`	✓	✓	✓	✓	✓	✓
`SupportStorageModeFixedBlockDedupe`	✓	✓	✓	✓	✓	✓
`SupportStorageModeNoDedupe`	✓	✓	✓	✓	✓	✓
`SupportWriteSparse`	✓	✓	✓	✓	✓	✓
`SupportWriteInPlace`	✓	✓	✓	✓	✓	✓
`SupportRawReadWrite`	✓	✓	✓	✓	✓	✓
`SupportMultipleObjectOpeners`	✓	✓	✓	✓	✓	✓
`SupportMultipleObjectWriters`	✓	✓	✓	✓	✓	✓
`SupportCloneExtent`	✓	✓	✓	✓	✓	✓

6.4 Merging Client, Server and ObjectStore Capabilities

The overall capabilities available to the backup application will be the most restrictive combination of both the Catalyst Client capabilities and the Catalyst Server capabilities. For example, if the Catalyst Client capabilities set `SupportClientPermissions` to true but the Catalyst Server capabilities set `SupportClientPermissions` to false, the combined capability would be false; Client Permissions would not be supported. These capabilities can be different depending on the ObjectStore policy, i.e. low bandwidth transfer policy might support some datapath operation that high bandwith doesn't, that's why the ObjectStore capabilities and transfer policy should also be taken into consideration.

The backup application should always merge the client, server and ObjectStore capabilities in order to determine this most restrictive combination. Failure to merge cababilities can result in the client attempting to perform an operation which the server does not support.

6.4.1 Merging Client and Server Capabilities

In older version of the Catalyst Client API the merge operation was implemented on ISV space, from v7 a new function `osClt_CombineCapabilities` is provided to merge these capabilities.
Please note that this function will merge all the capabilities and provide a merge version of them except for the following ObjectStore capabilities:

- `SupportWriteSparse`
- `SupportWriteInPlace`
- `SupportMultipleObjectOpeners`
- `SupportMultipleObjectWriters`

6.4.2 Code Example – Merging Client and Server Capabilities (for versions older than v7)

```
int main(int argc, char *argv[])
{
    const char *pServerIP = "127.0.0.1";
    int callStatus = OSCLT_ERR_SUCCESS;
    OSCLT_sClientPropertiesType clientProp;
    OSCMN_sServerPropertiesType serverProp;
    bool teamingSupported = false;
    bool permissionChecking = false;

    //<<Memset structs>>
    callStatus = osClt_GetClientProperties(&clientProp);
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Failed to get client properties : %s.\n",
                objSDKPrv_ErrorNumToString(callStatus));
        return 0;
    }
    callStatus = objSDKPrv_GetServerProp(pServerIP, &serverProp);
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
        printf("Error : Failed to get server properties : %s.\n",
                objSDKPrv_ErrorNumToString(callStatus));
        return 0;
    }
    printf("----- Comparing Client and Server Capabilities -----\n");
    printf("\t\t\tClient\t\tServer\n");
    printf("Min Protocol \t:\t%"PRIu32"\t\t%"PRIu64"\n",
           clientProp.ClientCapabilities.MinimumProtocolVersion,
           serverProp.ServerCapabilities.MinimumProtocolVersion);

    printf("Max Protocol \t:\t%"PRIu32"\t\t%"PRIu64"\n",
           clientProp.ClientCapabilities.MaximumProtocolVersion,
           serverProp.ServerCapabilities.MaximumProtocolVersion);

   printf("Object Copy \t:\t%s\t\t%s\n",
           clientProp.ClientCapabilities.SupportObjectCopy ? "Yes" : "No",
           serverProp.ServerCapabilities.SupportObjectCopy ? "Yes" : "No");

printf("Support Teaming :\t%s\t\t%s\n",
          clientProp.ClientCapabilities.SupportTeaming ? "Yes" : "No",
          serverProp.ServerCapabilities.SupportTeaming ? "Yes" : "No");

    if (clientProp.ClientCapabilities.SupportTeaming &&
        serverProp.ServerCapabilities.SupportTeaming)
    { // Teaming is supported - Store internally for later use
        teamingSupported = true;
    }

   if(clientProp.ClientCapabilities.SupportClientPermissions &&
      serverProp.IsClientPermissionsCheckingEnabled)
    { // Client permissions are supported - Store for later use
        permissionChecking = true;
    }
    printf("\n");
}
```

6.4.3 Other Supported Capabilities

The reason for having some capabilities that are not merged is because they don't depend on the client version. If one of these capabilities is supported by the specific ObjectStore then it can be used without taking into consideration the version of the client.

Out of all the mentioned capabilities on 6.4.1 the only that is supported in any of the HPE StoreOnce product versions is `SupportMultipleObjectOpeners`, support for this feature was added in server version 3.15.0.

6.5 Section Summary and recommendated best practises

- Always merge the client and server properties to obtain the properties which are common between the server and client versions
- Use default client and server properties rather than overriding defaults unless advised to do so by your integration HPE contact.

Section 7: StoreOnce Catalyst Stores

7.1 Catalyst Stores

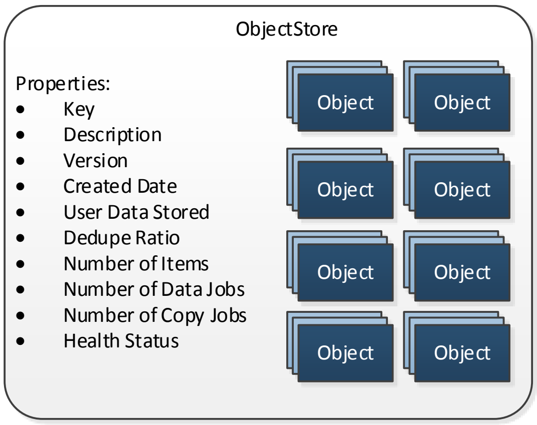


Figure 19 - Catalyst Store properties

7.1.1 Properties

Each Catalyst store has associated properties. These include:

Table 11 - Store Properties Supported Protocols

Store Property (OSCMN_sObjectStoreType)	Protocol v4	Protocol v5	Protocol v6	Protocol v7	Protocol v8	Protocol v9
`Id`	✓	✓	✓	✓	✓	✓
`ConfigState`	✓	✓	✓	✓	✓	✓
`Key`	✓	✓	✓	✓	✓	✓
`Description`	✓	✓	✓	✓	✓	✓
`Version`	✓	✓	✓	✓	✓	✓
`SecureEraseOverwriteCount`				✓	✓	✓
`CreationDateUTC`	✓	✓	✓	✓	✓	✓
`LastModifiedDateUTC`		✓	✓	✓	✓	✓
`Capabilities`	✓	✓	✓	✓	✓	✓
`WritePolicy`	✓	✓	✓	✓	✓	✓
`QuotaPolicy`	✓	✓	✓	✓	✓	✓
`JobRetentionPolicy`	✓	✓	✓	✓	✓	✓
`StorageEncryptionPolicy`		✓	✓	✓	✓	✓
`TeamingPolicy`		✓	✓	✓	✓	✓
`Status`	✓	✓	✓	✓	✓	✓
`ObjectStoreTeaming`		✓	✓	✓	✓	✓

* Protocol v5 will return Federated (Teaming) values in structures but they will always be false.

7.1.2 Listing Catalyst Stores

When listing the available Catalyst Stores on a Catalyst Server the backup application can specify a subset of the Catalyst Stores by specifying a filter. High level information is returned for each Catalyst Store in the list. Only Catalyst Stores to which the client is allowed access by the Catalyst Server will be returned in the list and the list will be returned in order of the Catalyst Store creation date.

Depending on the number of Catalyst Stores it might not be possible to return the information for all requested Catalyst Stores in one operation. The maximum number supported is based on the number of Catalyst Store records the client, the Client API, and the server can handle in one Operation. If the full list cannot be returned in one operation, PartialData will be returned as true. The caller can call again with the ListHandle set to the value returned in the previous call to continue the returned List from where the previous called reached. If this is not a follow-on call, the ListHandle should be blank. To keep the requests to a minimum it is recommended that when calling osCltCmd_ListObjectStores the objectStoreArraySize should be set to MaximumObjectStoresPerListIteration as reported by osCltCmd_GetServerProperties. The list returned will only contain Catalyst Stores for which the Client has access.

All stores/objects/datajob records in V2 Catalyst Stores are assigned partially ordered UUIDs and these are used for list result ordering in Federated Command Sessions. These UUIDs are assigned during creation and when the record is modified. Bitmasks of the Federated Members involved in the creation/modification are also assigned at the same time.

The UUIDs enable consistency to be checked and the bitmasks enable completeness to be checked. UUIDs are retrieved by the client API from the team member it initially connected to. In order to ensure that the time element of UUIDs retrieved from different team members does not skew too much, the client API checks on the open teamed command/data session that the UTC clocks on the time members are within 5 seconds of each other. The expectation is that all team members will be NTP synchronised.

Stores which return IsTeamed should be relisted using a Federated command session to get rolled up statistics. Integrators are not recommended to store details of whether a Catalyst store is Federated or non-federated, because it is possible to expand a non-federated Catalyst store to be Federated, and vice versa, a Federated store can be contracted to become a non-Federated store. The application should instead query whether a store IsTeamed or not, when opening sessions.

7.1.2.1 Code Example – List All Stores from Server

```
int objSDKPrv_ListAllStoresFromServer(const char *pServerAddress)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sServerPropertiesType serverProp;
      bool partialData = true;
      uint32_t i, numStoresReturned = 0;
      OSCLT_sCompoundListHandleType listHandle;
      OSCMN_sObjectStoreListFilterType objStoreListFilter;
      OSCMN_sObjectStoreType *pStoreList = 0;
      OSCLT_sObjectStoreTeamingListConsistencyStatusType
        *pObjStoreTeamingListConsistencyStatusArray = NULL;

      //<<Memset structs - Setting empty filter will match all stores>>
      callStatus = objSDKPrv_GetServerProp(pServerAddress,
                                           &serverProp);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Unable to get server '%s' properties.\n",
                    pServerAddress);
            returnStatus = callStatus;
            return returnStatus;
      }
      // Allocate space for list stores
      if (pStoreList = (OSCMN_sObjectStoreType *) calloc(
                        serverProp.ServerCapabilities.
                            MaximumObjectStoresPerListIteration,
                                 sizeof(OSCMN_sObjectStoreType)) == NULL)
      {
            printf("Error : Unable to allocate space for '%"PRIu64
                   "' stores.\n",
                    serverProp.ServerCapabilities.
                          MaximumObjectStoresPerListIteration);
            returnStatus = callStatus;
            return returnStatus;
      }
      printf("List all stores from server : %s\n", pServerAddress);
      //<<Open command session>>
      do
      {
            callStatus = osCltCmd_ListObjectStores(pCmdSessionHandle,
            &objStoreListFilter,
            serverProp.ServerCapabilities.
                    MaximumObjectStoresPerListIteration,
            pStoreList,
            pObjStoreTeamingListConsistencyStatusArray,
            &listHandle,
            &partialData,
            &numStoresReturned);
            if (callStatus != OSCLT_ERR_SUCCESS)
            {
                printf("Error : Listing stores from server '%s' failed"
                       "with error : %s.\n", pServerAddress,
                        objSDKPrv_ErrorNumToString(callStatus));
                //<<Close command session>>
                returnStatus = callStatus;
                if (pStoreList)
                {
                     free(pStoreList);
                }
                return returnStatus;
            }
            for (i = 0; i < numStoresReturned; i++)
            {
                  // Print Store Properties
                  printf("\tName        : %s\n", pStoreList[i].Key.String);
                  printf("\tDescription : %s\n",
                      pStoreList[i].Description.String);
                  printf("\tTeamed      : %s\n",
                      pStoreList[i].TeamingPolicy.IsTeamed ? "Yes" : "No");
                  printf("\n");
            }// Loop until partial data set to true
      } while (partialData);
      //<<Close command session>>
      //<<Free memory allocation>>
      return returnStatus;
}
```

7.1.3 Creating a Catalyst Store

Creating a Store in a backup application is considered advanced integration and is not undertaken by the majority of integrators.

In addition to the user creating Catalyst Stores through the HPE StoreOnce web interface, the Catalyst Client library provides the capability to create Catalyst Stores on the Catalyst Server directly from the backup application.

When using this feature the backup application must ensure that the client identifier strings used to open the command session has authorisation on the Catalyst Server to create Catalyst Stores. See the Client Authentication section of this guide.

The creation of Catalyst Stores is an asynchronous task and therefore once the request has been made the backup application should monitor for the Catalyst Store coming into an online state. This is done by periodically calling `osCltCmd_ListObjectStores` using an `ObjectStoreKeyFilter` for the Catalyst Store to be monitored. When `OSCMN_sObjectStoreType.Status.IsOnline` is TRUE, the Catalyst Store is online and is ready to be used. The recommended polling frequency when waiting for a store to come online is once per minute.

7.1.3.1 Code Example – Create a Store

```
int objSDKPrv_CreateStore(const char *pServerAddress, const char
*pStoreKey, const char *pStoreDescription)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sKeyType storeKey;
      OSCMN_sStringType storeDescription;
      OSCMN_sWritePolicyType writePol;
      OSCMN_sQuotaPolicyType quotaPol;
      OSCMN_sJobRetentionPolicyType jobReten;
      OSCMN_sStringType dedupeStore;
      OSCMN_sStorageEncryptionPolicyType storageEncryptionPolicy;
      OSCMN_sTeamingPolicyType teamingPolicy;

      // <<Memset structs>>
      // <<Set Store Key and Store Description Strings>>
      writePol.WriteBandwidthMode =
            OSCMN_WRITE_POLICY_DEFAULT_BANDWIDTHMODE;
      jobReten.DataJobRetentionDays =
            OSCMN_JOB_RETENTION_POLICY_DEFAULT_DAYS;
      jobReten.OriginCopyJobRetentionDays =
            OSCMN_JOB_RETENTION_POLICY_DEFAULT_DAYS;
      jobReten.DestinationCopyJobRetentionDays =
            OSCMN_JOB_RETENTION_POLICY_DEFAULT_DAYS;
      // <<Open command session>>
      callStatus = osCltCmd_CreateObjectStore (pCmdSessionHandle,
                                          &storeKey,
                                          &storeDescription,
                                          &writePol,
                                          &quotaPol,
                                          &jobReten,
                                          &storageEncryptionPolicy,
                                                        &teamingPolicy,
       if (callStatus != OSCLT_ERR_SUCCESS) // Failed to create store
       {
           if (callStatus != OSCLT_ERR_DUPLICATE_OBJECTSTORE_KEY)
           { // Unexpected error
               printf("Error : Failed to create store '%s'. Error : %s.\n",
                       pStoreKey, objSDKPrv_ErrorNumToString(callStatus));
               // <<Close command session>>
               returnStatus = callStatus;
               return returnStatus;
           }
           if (callStatus == OSCLT_ERR_DUPLICATE_OBJECTSTORE_KEY)
           { // Store already exists on server
               printf("Store '%s' exists on server '%s'.\n",
                       pStoreKey, pServerAddress);
           }
       }
       else
       { // Store created
           printf("Created store '%s' in server '%s'.\n",
                   pStoreKey, pServerAddress);
       }
       // <<Close command session>>
       // Wait for store to come online
       callStatus = objSDKPrv_WaitForStore(pServerAddress,
                                            pStoreKey,
                                            true);
       if (callStatus != OSCLT_ERR_SUCCESS)
       {
             printf("Error : Store '%s' failed to come online.\n",
                     pStoreKey);
             returnStatus = callStatus;
             return returnStatus;
       }
       return returnStatus;
}
```

7.1.3.2 Catalyst Store Encryption

When creating a Catalyst Store it is possible to set the Store to encrypt data that is stored in it. A backup application should check that the Catalyst Server supports encryption by getting the Catalyst Server properties and checking its capabilities. If `IsServerStorageEncrypted` is set to true then the server is already encrypting the data for the Catalyst Store and setting the Catalyst Store Encryption Policy to true would be superfluous. Please see the Getting Client Properties section on page 51.

A Catalyst Store can only be encrypted when it is first created, if after creation the customer wishes to encrypt the data it should be copied to a newly created encrypted Catalyst Store.

7.1.4 Modifying a Catalyst Store

Modifying a Catalyst Store in a backup application is considered advanced integration and is not undertaken by the majority of integrators.

After a Catalyst Store has been created the Catalyst Client library provides the capability to further modify it using the `osCltCmd_ModifyObjectStore` method. If Client

Permissions checking is enabled on the Catalyst Server then the backup application must have permission to access the Catalyst Store otherwise an error will be returned. Client Permissions are explained further in the Client Authentication section on page 109.

The Catalyst Store Properties that can be modified are:

- Store description
- Write policy
- Quote policy
- Job log retention period

When a Catalyst Store is to be modified the backup application should get the capabilities of the Catalyst Server to ensure invalid values are not set. Please see Getting Client Properties section on page 51.

Federated Stores can be modified using a normal or bootstrap command session. Store modification using bootstrap command session is less secure, so it should only be used to update a Federated Member Address. All the Federated Member Stores must be configured but do not have to be online before modification can occur. See Federated Command Sessions section on page 25 for more information on normal and bootstrap sessions.

As the `osCltCmd_ ModifyObjectStore` function will update each property if there is a setting which should remain unchanged it is necessary to pull the existing store properties and resend them with the newly modified properties, merged.

7.1.4.1 Modifying Description

A backup application is able to update the description of a Catalyst Store by modifying the `OSCMN_sStringType` struct.

7.1.4.2 Modifying Write Policy

A backup application is able to update the write bandwidth mode in the write policy of a Catalyst Store by updating the `OSCMN_sWritePolicyType` struct.

7.1.4.3 Modifying Quota Policy

A backup application is able to update the data size quotas applied to a Catalyst Store by updating the `OSCMN_sQuotaPolicyType` struct.

7.1.4.4 Complete Code Example – Modify an existing Store

```
int objSDKPrv_ModifyStore(const char *pServerAddress, const char
*pStoreKey)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = NULL;
      OSCMN_sObjectStoreType store;
      OSCMN_sKeyType storeKey;
      OSCMN_sStringType storeDescription;
      OSCMN_sWritePolicyType writePol;
      OSCMN_sQuotaPolicyType quotaPol;
      OSCMN_sJobRetentionPolicyType jobReten;
      OSCMN_sTeamingPolicyType teamingPolicy;

      memset(&storeKey,
             0,
             sizeof(storeKey));

      memset(&storeDescription,
             0,
             sizeof(storeDescription));

      memset(&writePol,
             0,
             sizeof(writePol));

      memset(&quotaPol,
             0,
             sizeof(quotaPol));

      memset(&jobReten,
             0,
             sizeof(jobReten));

      memset(&teamingPolicy,
             0,
             sizeof(teamingPolicy));

      // <<Get Store | See 'Listing Catalyst Stores'>>
      // <<Set Store Key>>
      // <<Set new values>>
      writePol.WriteBandwidthMode = OSCMN_WRITEBANDWIDTHMODE_LOW_ONLY;

      // Use old valyes
      storeDescription = store.Description;
      jobReten = store.JobRetentionPolicy;
      teamingPolicy = store.TeamingPolicy;

      // <<Open command session>>
      callStatus = osCltCmd_ModifyObjectStore(pCmdSessionHandle,
                                              &storeKey,
                                              &storeDescription,
                                              &writePol,
                                              &quotaPol,
                                              &jobReten,
                                              &teamingPolicy);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Unable to modify store : %s.\n",
                    objSDKPrv_ErrorNumToString(callStatus));
            returnStatus = callStatus;
            // <<Close command session>>
            return returnStatus;
      }
      // <<Close command session>>
      return returnStatus;
}
```

7.1.5 Deleting a Catalyst Store

Deleting a Store is considered advanced integration. Backup applications supporting this functionality should ensure that appropriate warning is given to the user if they request to delete a store. Store deletion is non-reversible and will result in dataloss.

The deletion of Catalyst Stores is an asynchronous task and therefore once the request has been made the backup application should monitor for the Catalyst Store no longer being listed in response to `osCltCmd_ListObjectStores`. The Catalyst Store deletion may take many minutes as the data is removed from the Catalyst Server. Therefore the recommended polling frequency when waiting for a Catalyst Store to be deleted is once every minute.

7.1.5.1 Code Example – Delete a Store

```
int objSDKPrv_DeleteStore(const char *pServerAddress, const char
*pStoreKey)
{
	int returnStatus = OSCLT_ERR_SUCCESS;
	int callStatus = OSCLT_ERR_SUCCESS;
	OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
	OSCMN_sKeyType storeKey;

	//<<Set store key>>
	//<<Open command session>>

	callStatus = osCltCmd_DeleteObjectStore(pCmdSessionHandle,
	                                        &storeKey);
	if (callStatus != OSCLT_ERR_SUCCESS)
	{
		if (callStatus != OSCLT_ERR_OBJECTSTORE_KEY_NOT_FOUND)
		{
			printf("Error : Failed to delete store '%s'. "
			       "Error : %s.\n", pStoreKey,
			        objSDKPrv_ErrorNumToString(callStatus));
		}
		else
		{
			printf("Error : Store '%s' does not exist. "
			       "Error : %s.\n", pStoreKey,
			        objSDKPrv_ErrorNumToString(callStatus));
		}
		//<<Close command session>>
		returnStatus = callStatus;
		return returnStatus;
	}
	//<<Close command session>>
	printf("Deleted store '%s' from server '%s'.\n",
	        pStoreKey, pServerAddress);
	// Wait for store to go offline to ensure we report on success
	callStatus = objSDKPrv_WaitForStore(pServerAddress,
	                                    pStoreKey,
	                                    false);
	if (callStatus != OSCLT_ERR_SUCCESS)
	{
		printf("Error : Failed to delete Store '%s'.\n", pStoreKey);
		returnStatus = callStatus;
		return returnStatus;
	}
	return returnStatus;
}
```

7.1.6 Wait for a Catalyst Store

When creating or deleting a Catalyst Store it is recommended to monitor the status to ensure the action is completed successfully.

7.1.6.1 Code Example – Wait for a Catalyst Store

```
int objSDKPrv_WaitForStore(const char *pServerAddress, const char
*pStoreKey, bool waitForOnline)
{
      int returnStatus = OBJSDK_ERR_SUCCESS;
      int callStatus = OBJSDK_ERR_SUCCESS;
      int waitCounter = 0;
      OSCMN_sObjectStoreType store;
      printf("Wait for store '%s' to %s.\n", pStoreKey, waitForOnline?
             "come online" : "be deleted");
      do
      {
            callStatus = objSDKPrv_GetStoreByKey(pServerAddress,
                                                 pStoreKey,
                                                 &store);
            if (callStatus != OBJSDK_ERR_SUCCESS)
            {
                  // Check if we are waiting for store to be deleted
                  if (callStatus == OBJSDK_ERR_KEY_NOT_EXIST &&
                                                      !waitForOnline)
                  {
                        // Store was deleted successfully, return success
                        break;
                  }
                  // Error while waiting for store to come online
                  printf("Error : Failed to get store '%s' properties from"
                         " server '%s'. Error : %s\n", pStoreKey,
                         pServerAddress,
                         objSDKPrv_ErrorNumToString(callStatus));
                  returnStatus = callStatus;
                  return returnStatus;
            }
            // Check if we are waiting for store to be created
            if (waitForOnline && store.Status.IsOnline)
            {
                  break;
            }
            // Keep waiting
            waitCounter++;
            printf(".");
            fflush(stdout);
            // Sleep one second
            sleep(1);
      } while (waitCounter < 60);
      printf("\n");
      if (waitCounter < 60)
      {
            printf("Store '%s' is %s.\n", pStoreKey, waitForOnline?
                   "ready" : "gone");
      }
      else
      {
            printf("Error : Store '%s' failed to %s.\n", pStoreKey,
                    waitForOnline? "come online" : "be deleted");
      }
```

```
        return returnStatus;
}
```

7.2 Section Summary and recommendated best practises

- Catalyst stores cannot to renamed after creation
- Whether a Catalyst store is teamed or non-teamed should not be persisted to an application config because stores can be contracted / expanded to be teamed / non-teamed.
- Defer store creation to the StoreOnce GUI rather than exposing store creation within the backup application.

Section 8: StoreOnce Catalyst Items

8.1 Catalyst Items

Catalyst Items are stored in Catalyst Stores and are containers for storing customers backed up data, typically from one backup job. Each Item can only have a maximum of one concurrent write data job runningto it; otherwise an error will be returned. To allow customers to perform backups using multiple streams it is necessary to create multiple Items and perform the backup to those multiple items.

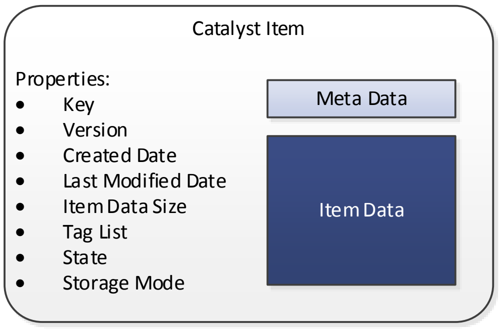


Figure 20 - Catalyst Item properties

Catalyst Items consist of three elements:

- Properties
- Meta Data
- Data

8.1.1 Properties

Each Catalyst Item has the following properties:

Table 12 - Item Properties Supported Protocol

Object Property (OSCMN_sObjectType)	Protocol v4	Protocol v5	Protocol v6	Protocol v7	Protocol v8	Protocol v9
`Id`	✓	✓	✓	✓	✓	✓
`ConfigState`	✓	✓	✓	✓	✓	✓
`Key`	✓	✓	✓	✓	✓	✓
`Version`	✓	✓	✓	✓	✓	✓
`DataSize`	✓	✓	✓	✓	✓	✓
`DataStored`		✓	✓	✓	✓	✓
`CreationDateUTC`	✓	✓	✓	✓	✓	✓
`MetaLastModifiedDateUTC`		✓	✓	✓	✓	✓
`DataLastModifiedDateUTC`	✓ (`LastModifiedDateUTC`)	✓	✓	✓	✓	✓
`StorageMode`	✓	✓	✓	✓	✓	✓
`TagList`	✓	✓	✓	✓	✓	✓
`ObjectMeta`	✓	✓	✓	✓	✓	✓
`ObjectTeaming`		✓	✓	✓	✓	✓

* Protocol v5 will return Federated (Teaming) values in structures but they will always be false.

8.1.1.1 Name

The name of a Catalyst Item must be unique within a Catalyst Store and therefore it is referred to as the Key, it cannot be altered after creation. The Key must be provided whenever a session is interacting with a Catalyst Item. Valid characters are:

- v4: ASCII char. Only [a-z][A-Z][0-9][_-.+]
- v5: ASCII char. Only [a-z][A-Z][0-9][_-.+(){}:#$*;=?@[]^|~]

See SupportKeyTaglistExtendedCharSet capability to verify if v5 charset is supported.

8.1.1.2 Version

The elements and properties of a Catalyst Item may change in the future and therefore the version number refers to the structure used for the Catalyst Item.

8.1.1.3 Data Size

There are two data sizes recorded for each Catalyst Item:

- Data Size
- Data Stored

The Data Size is the total size of the Catalyst Item including any sparse regions. The Data Stored is the amount of user data stored in the Item (not including any sparse regions). Catalyst Items stored in Federated Stores return the sizes of the Item across all Federated Members when opened in a Federated Session.

8.1.1.4 Timestamps

There are three timestamps recorded for each Catalyst Item:

- Creation Date
- Metadata Last Modified Date
- Data Last Modified Date

The timestamps are localised to the UTC time zone. The Creation Date is the timestamp when the Catalyst Item was first created and will not change. The timestamps for Metadata Last Modified and Data Last Modified will be updated when either metadata or user data is updated.

8.1.1.5 Storage Mode

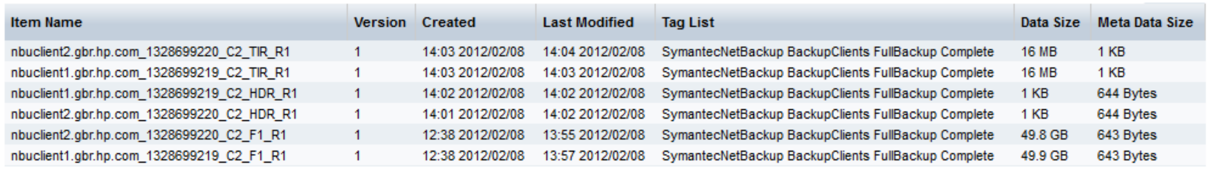
The storage mode of a Catalyst Item is the method of deduplication that will be applied to any data stored in the data element of the Catalyst Item. There are three potential storage modes, although not all Catalyst Stores supports all storage modes:

- No Deduplication
- Variable Block Deduplication
- Fixed Block Deduplication

For more information on Storage Modes see the Data information below (page 80)

8.1.1.6 Tag Lists

Objects have a tag list property which is a space separated list of strings used to describe the Catalyst Item. The strings included in the tag list are displayed in the StoreOnce GUI and therefore it is recommended that user interpretable tags are used whenever possible.



Item Name	Version	Created	Last Modified	Tag List	Data Size	Meta Data Size
nbuclient2.gbr.hp.com_1328699220_C2_TIR_R1	1	14:03 2012/02/08	14:04 2012/02/08	SymantecNetBackup BackupClients FullBackup Complete	16 MB	1 KB
nbuclient1.gbr.hp.com_1328699219_C2_TIR_R1	1	14:03 2012/02/08	14:03 2012/02/08	SymantecNetBackup BackupClients FullBackup Complete	16 MB	1 KB
nbuclient1.gbr.hp.com_1328699219_C2_HDR_R1	1	14:02 2012/02/08	14:02 2012/02/08	SymantecNetBackup BackupClients FullBackup Complete	1 KB	644 Bytes
nbuclient2.gbr.hp.com_1328699220_C2_HDR_R1	1	14:01 2012/02/08	14:02 2012/02/08	SymantecNetBackup BackupClients FullBackup Complete	1 KB	644 Bytes
nbuclient2.gbr.hp.com_1328699220_C2_F1_R1	1	12:38 2012/02/08	13:55 2012/02/08	SymantecNetBackup BackupClients FullBackup Complete	49.8 GB	643 Bytes
nbuclient1.gbr.hp.com_1328699219_C2_F1_R1	1	12:38 2012/02/08	13:57 2012/02/08	SymantecNetBackup BackupClients FullBackup Complete	49.9 GB	643 Bytes

Examples of useful tags could include:

- Application (e.g. SymantecNetBackup)
- Source of backup data (e.g. mssqlserver.hp.com)
- Source data type (e.g. SQLBackup)
- Backup type (e.g. FullBackup)

- Backup policy name (e.g. BackupClients)

Additionally it is also recommended that a further tag of either “Incomplete” or “Complete” also be used. The intention being that a Catalyst Item has a tag of “Incomplete” when it is first created and once all the intended data has been written to the Item, the tag is updated to be “Complete”. Having this status tag will provide users with the ability to identify items which are not yet completed backups. In the rare scenario where a backup fails and the Catalyst Item is orphaned, the user will be able to identify the Catalyst Item and manual delete it through a management interface, such as the StoreOnce GUI.

The tag list is first defined when creating a Catalyst Item using the `osCltCmd_CreateObject` operation. The tag list may be modified at any time by calling the `osCltCmd_ModifyObjectMeta` operation. Valid characters for tags are `[a-z][A-Z][0-9][_-.+(){}:#$*;=?@[]^|~]` with space being used as the delimiter between tags.

Tag list strings are returned for each Catalyst Item when the `osCltCmd_ListObjects` operation is called. The `osCltCmd_ListObjects` operation also allows for a Tag List Filter to be supplied which may be used to specifically limit the returned items to those with particular tags. A tag list filter may be applied in one of two ways, either Objects with any matching tags are returned or Objects where only all tags match are returned. The method of matching is specified using the `AllMatch` Boolean when defining the Tag List Filter.

It is recommended that as a minimum, the Application tag be used for all items. And when performing the `osCltCmd_ListObjects` a Tag List Filter is used with the Application tag defined. This will ensure that the calling application only ever lists Objects that it has created.

8.1.2 Metadata

Each Catalyst Item may also optionally contain associated metadata for the purposes of the calling application. Example uses of this metadata could include:

- Application Revisions
- Individual Component Revisions
- Byte offsets for catalogue information

Metadata may be written to a Catalyst Item when it is first created using the `osCltCmd_CreateObject` operation and it may be subsequently updated by calling the `osCltCmd_ModifyObjectMeta` operation. In both cases the operation should be passed the buffer location and size of the metadata to be stored within the Catalyst Item.

Metadata for a Catalyst Item is optionally returned when the `osCltCmd_ListObjects` operation is called. For metadata to be returned a buffer must be supplied into which the metadata will be copied and the `includemeta` Boolean flag must be set when the `osCltCmd_ListObjects` operation is called.

To allow HPE Support to be able to identify which applications have created particular Objects it is recommended that all metadata that is written uses JSON encoding using the following structure:

```
{
        "Common": {
                "FormatVersion": 0,
                "ClientSoftwareVersion": "WIN_x64_5_4.0.4",
                "ISVInfo": {
                        "Name": "HPE Data Protector",
                        "Version": "7.00",
                        "Platform": "Microsoft Window 2008 R2 (x86_64)"
                }
        },
        "ISV": {
                "FormatVersion": 0,
                "Data": {
                        <Base64 encoded data>
                }
        }
}
```

This JSON data structure consists of two distinct sections; Common metadata and ISV metadata.

8.1.2.1 Common Metadata

The Common metadata structure should be used to track information about the Catalyst Client and backup application that created the Catalyst Item. The Catalyst Client Software Version that should be included is the value returned by the `osClt_GetClientProperties` operation. The backup application name, version and platform should be provided by the calling application. Backup applications are advised to keep this information up to date as new versions are published.

8.1.2.2 ISV Metadata

The ISV metadata may be used as the calling application requires. For basic string information standard key:value pairs should be used but binary data should be string encoded using base64 or similar. It is recommended that a version number is also applied to the backup application metadata to allow for future metadata structure enhancements.

The maximum size of metadata that can be written per Catalyst Item is defined as the `MaximumMetadataSizePerObject` value as returned in `OSCMN_sServerCapabilitiesType` by the `osCltCmd_GetServerProperties` operation. The maximum metadata size returned does not account for JSON or other encoding and so the caller should allow for such overheads in their internal data structures.

8.1.2.3 Code Example

Printing metadata is demonstrated in the List all Items (objSDKPrv_ListAllObjectsFromStore) example on page 85.

8.1.3 Data

Data written to a Catalyst Item can be deduplicated using Variable Block chunks or Fixed Block chunks depending on the Object Storage mode setting for the Item and the modes supported by the Server and Store. Where Fixed Block Deduplication breaks the stream based on a ISV specified offset starting at the end of the file, Variable Block Deduplication splits the data stream into chunks based on the data itself by performing a small rolling calculation across a window of the data. If the integrator knows that data is Fixed block aligned, such as certain databases, then it is highly recommended to use Fixed block chunking; this method will result in better deduplication ratios, performance and client side resource utilisation. Items should be created as using either the `OSCMN_DEDUPEMODE_FIXED_BLOCK_DEDUPE` mode or `OSCMN_DEDUPEMODE_VARIABLE_BLOCK_DEDUPE` of `OSCMN_eDedupeModeType`. When creating as fixed block integrators must ensure that write offsets, write sizes and write buffers are block aligned. Also, when performing read operations the operations must also be block aligned to the alignment size used to write the Catalyst item. It is advisable to store the block size within the item meta.

Data must be written to an Item in a sequential manner as write in place is unsupported at this time. It is also recommended to read data sequentially to improve performance.
Catalyst Items do not have a maximum data size; however the Catalyst Store containing the Item can have a quota applied to it for both physical data stored and logical data stored for Data Size and Data Stored so this setting should be taken into account.

It is highly recommended behaviour to have separate Catalyst Items for each data session, i.e. – each backup job should be saved to a new Item and so a backup application should default to this.

8.1.4 Creating a Catalyst Item

Before a backup application can write backup data to a Catalyst Store is must create a Catalyst Item into which the data will be written. A backup job may create one or more Objects to store the backup data but it is recommended that multiple backup jobs do not share Objects as this allows for improved resolution when expiring data.

When creating a Catalyst Item the backup application should set the initial tag list and metadata. The tag list should contain a tag of "Incomplete" and as minimum the metadata should include the Common information.

When calling the `osCltCmd_CreateObject` operation the Catalyst Store into which the Catalyst Item will created, must be supplied. If the client does not have access to that store the request will fail with error `OSCLT_ERR_PERMISSION_DENIED`.

Additionally the backup application must supply the storage method which the Catalyst Item will be stored. Possible options are reported in the Catalyst Store capabilities returned by osCltCmd_ListObjectStores. Once a Catalyst Item has been created the storage mode may not be modified.

When creating the Catalyst Item it is also possible for the backup application to provide metadata to associate with the Catalyst Item. The metadata may be modified later using the osCltCmd_ModifyObjectMeta operation.

8.1.4.1 Code Example – Create an Item

```
int objSDKPrv_CreateObject(const char *pServerAddress, const char
*pStoreKey, const char *pObjectKey, const char *pObjectTags, char
*pMetaData, uint32_t metaDataSize)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sKeyType storeKey;
      OSCMN_sKeyType objKey;
      OSCMN_sObjectStorageModeType objectStorageMode;
      OSCMN_sTagListType objectTagList;
      OSCMN_sObjectMetaType objectMetaBuffer;

      //<<Memset all structs>>

      //<<Set Store Key>>
      //<<Set Object Key>>

      // Set dedupe mode
      objectStorageMode.DedupeMode = OSCMN_DEDUPEMODE_VARIABLE_BLOCK_DEDUPE;

      if (pObjectTags)
      {
            memset(&objectTagList,
                   0,
                   sizeof(objectTagList));

            strncpy(objectTagList.String,
                    pObjectTags,
                    sizeof(objectTagList.String));

            if (objectTagList.String[sizeof(objectTagList.String) - 1])
            {
                  printf("Error : Unable to set Object Tags %s\n",
                         objectTagList.String);
                  returnStatus = OSCLT_ERR_INTERNAL_ERROR;
                  return returnStatus;
            }
            objectTagList.StringSize = strlen(objectTagList.String);
      }
      // Metadata provided?
      if (pMetaData && metaDataSize) {
            objectMetaBuffer.MetaSize = metaDataSize;
            objectMetaBuffer.pMetaData = (uint8_t *) pMetaData;
      }
      //<<Open command session>>
      callStatus = osCltCmd_CreateObject(pCmdSessionHandle,
                                         &storeKey,
                                         &objKey,
                                                  &objectStorageMode,
                                                  &objectTagList,
                                                  &objectMetaBuffer);
      if (callStatus != OSCLT_ERR_SUCCESS) {
            printf("Error : Creating object '%s' in store '%s' failed "
                   "with error : %s.\n",
                    pObjectKey, pStoreKey,
                    objSDKPrv_ErrorNumToString(callStatus));
            returnStatus = callStatus;
            //<<Close command session>>
            return returnStatus;
      }
      printf("Created object '%s' in store '%s'.\n", pObjectKey,
              pStoreKey);
      //<<Close command session>>
      return returnStatus;
}
```

8.1.5 Listing Catalyst Items

Catalyst Stores may contain millions of Catalyst Items and although it is possible to list every Catalyst Item within a Catalyst Store, during normal usage it is not recommended. When the purpose of the list is to retrieve Catalyst Item properties or metadata for a specific Catalyst Item, the `osCltCmd_ListObjects` operation should be called with an `ObjectKeyFilter` provided. This will return the details of the single Catalyst Item specified by the `ObjectKeyFilter`.

When the properties or metadata for multiple Objects needs to be retrieved, the `osCltCmd_ListObjects` should be called with one or more of the following filters applied:

- A Catalyst Item Key Filter with exact match disabled.
- Tag List Filter
- Created Date Period Filter
- Modified Date Period Filter

The Catalyst Store shall respond with a list of all Objects that match all the filters applied. If there are more matching Objects to list than can be returned in the buffers provided, the `PartialData` flag shall be set to indicate that more Catalyst Item listing is available to be retrieved.

In order to retrieve the next batch of the Catalyst Item listings, the backup application should repeat the `osCltCmd_ListObjects` operation passing back the `ListHandle` returned in the previous response. The backup application should repeat this sequence until all the required Objects have been listed and the `PartialData` flag is set to false.

It is important to note that the ListHandle must be memset to zero whenever the first `osCltCmd_ListObjects` call is made to avoid getting incomplete list responses.

8.1.5.1 Code Example - List Item by Key

```
int objSDKPrv_GetObjectByKey(const char *pServerAddress, const char
*pStoreKey, const char *pObjectKey, OSCMN_sObjectType *pObject, char
*pMetaData, uint32_t metaDataSize)
```

```
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sKeyType storeKey;
      OSCMN_sBufferType objectMetaBuffer;
      OSCMN_sObjectListFilterType objectListFilter;
      OSCLT_sCompoundListHandleType listHandle;
      OSCLT_sObjectTeamingListConsistencyStatusType
            *pObjectTeamingListConsistencyStatusArray = NULL;

      bool includeMeta = false;
      bool partialData = false;
      uint32_t numObjectsReturned = 0;

      //<<Memset all structs>>
      //<<Set Store Key>>

      // Match the object using key
      objectListFilter.KeyFilter.ExactMatchOnly = true;
      strncpy(objectListFilter.KeyFilter.KeyFilter.String,
              pObjectKey,
              sizeof(objectListFilter.KeyFilter.KeyFilter.String));

      if (objectListFilter.KeyFilter.KeyFilter.String[
            sizeof(objectListFilter.KeyFilter.KeyFilter.String) - 1])
      {
            returnStatus = OSCLT_ERR_INTERNAL_ERROR;
            return returnStatus;
      }
      objectListFilter.KeyFilter.KeyFilter.StringSize =
            strlen(objectListFilter.KeyFilter.KeyFilter.String);
      // Get metadata details?
      if (pMetaData && metaDataSize)
      {
            includeMeta = true;
            objectMetaBuffer.BufferSize = metaDataSize;
            objectMetaBuffer.pBufferLocation = (uint8_t *) pMetaData;
      }
      //<<Open command session>>
      callStatus = osCltCmd_ListObjects (pCmdSessionHandle,
                                         &storeKey,
                                         &objectListFilter,
                                         includeMeta,
                                         false /* increasingOrder */,
                                         1,
                                  pObject,
                                  pObjectTeamingListConsistencyStatusArray,
                                  &objectMetaBuffer,
                                  &listHandle,
                                  &partialData,
                                  &numObjectsReturned);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
          printf("Error : Listing object '%s' from store '%s' failed with "
                 "error : %s.\n",  pObjectKey, pStoreKey,
                  objSDKPrv_ErrorNumToString(callStatus));
          //<<Close command session>>
          returnStatus = callStatus;
          return returnStatus;
```

```
        }
        if (numObjectsReturned != 1)
        {
            // numObjectsReturned is zero and partialData not set then
            // object not found, otherwise some other issue
            //<<Close command session>>
            returnStatus = (numObjectsReturned == 0 && !partialData)?
                OSCLT_ERR_OBJECTSTORE_KEY_NOT_FOUND :
                OSCLT_ERR_INTERNAL_ERROR;
            return returnStatus;
        }
        //<<Close command session>>
        return returnStatus;
}
```

8.1.5.2 Code Example - List Item by Tags

```
int objSDKPrv_GetObjectByTags(const char *pServerAddress, const char
*pStoreKey, const char *pObjectTags, OSCMN_sObjectType *pObject, char
*pMetaData, uint32_t metaDataSize)
{
    //<<As objSDKPrv_GetObjectByKey | Use TagListFilter instead
    // of KeyFilter>>
    objectListFilter.TagListFilter.AllMatchOnly = true;
    strncpy(objectListFilter.TagListFilter.TagListFilter.String,
            pObjectTags,
            sizeof(objectListFilter.TagListFilter.TagListFilter.String));

    if (objectListFilter.TagListFilter.TagListFilter.String
            [sizeof(objectListFilter.TagListFilter.TagListFilter.String)
                    - 1])
    {
        printf("Error : Unable to set Object Filter %s\n",
                objectListFilter.TagListFilter.TagListFilter.String);
        returnStatus = OSCLT_ERR_INTERNAL_ERROR;
        return returnStatus;
    }
    objectListFilter.TagListFilter.TagListFilter.StringSize =
            strlen(objectListFilter.TagListFilter.TagListFilter.String);
}
```

8.2 Section Summary and recommendated best practises

- Catalyst objects cannot to renamed after creation
- Catalyst integratoers are recommended to implement object metadata using the example JSON metadata encoding provided in the section example. This allows common support across all Catalyst implementations.
- Add “Complete” / “Incomplete” tags to backup items where possible.

8.2.1.1 List all Items with Metadata

```
int objSDKPrv_ListAllObjectsFromStore(const char *pServerAddress, const
char *pStoreKey, char *pMetaData, uint32_t metaDataSize)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sServerPropertiesType serverProp;
      OSCMN_sKeyType storeKey;
      OSCMN_sBufferType objectMetaBuffer;
      OSCMN_sObjectListFilterType objectListFilter;
      OSCLT_sCompoundListHandleType listHandle;
      OSCMN_sObjectType *pObjectList = 0;
      bool includeMeta = false;
      bool partialData = false;
      uint32_t i, numObjectsReturned = 0;
      OSCLT_sObjectTeamingListConsistencyStatusType
            *pObjectTeamingListConsistencyStatusArray = NULL;

      // <<Get server properties and allocate memory for list>>
      memset(&objectMetaBuffer,
           0,
           sizeof(objectMetaBuffer));
      // Empty filter (list all objects)
      memset(&objectListFilter,
           0,
           sizeof(objectListFilter));
      memset(&listHandle,
           0,
           sizeof(listHandle));

      // Get metadata details?
      if (pMetaData && metaDataSize)
      {
            includeMeta = true;
            objectMetaBuffer.BufferSize = metaDataSize;
            objectMetaBuffer.pBufferLocation = (uint8_t *) pMetaData;
      }

      // <<Set Store key>>
      printf("List all objects from store : %s\n", pStoreKey);
      // <<Open command session>>
      do
      {
            callStatus = osCltCmd_ListObjects(pCmdSessionHandle,
                                 &storeKey,
                                 &objectListFilter,
                                 includeMeta,
                                 false /* increasingOrder */,
                                 serverProp.ServerCapabilities.
                                           MaximumObjectsPerListIteration,
                                 pObjectList,
                                 pObjectTeamingListConsistencyStatusArray,
                                 &objectMetaBuffer,
                                 &listHandle,
                                 &partialData,
                                 &numObjectsReturned);
            if (callStatus != OSCLT_ERR_SUCCESS)
            {
                  printf("Error : Listing objects from store '%s' failed"
                         "with error : %s.\n", pStoreKey,
                     objSDKPrv_ErrorNumToString(callStatus));
                   // <<Close command session>>
                   returnStatus = callStatus;
                   return returnStatus;
             }
             for (i = 0; i < numObjectsReturned; i++)
             {
                   // Print Object Properties
                   printf("\tName     : %s\n",
                           pObjectList[i].Key.String);
                   printf("\tTags     : %s\n",
                           pObjectList[i].TagList.String);
                   printf("\tDataSize : %"PRIu64"\n",
                           pObjectList[i].DataSize);
                   printf("\tMetaSize : %u\n",
                           pObjectList[i].ObjectMeta.MetaSize);
                   printf("\tMetaData : %.*s\n",
                           pObjectList[i].ObjectMeta.MetaSize,
                           pObjectList[i].ObjectMeta.pMetaData);
                   printf("\n");
             }
       } while (partialData);// Loop until partial data set to true
       // <<Close command session>>
       // <<Free object list>>
       return returnStatus;
}
```

8.2.2 Modifying Catalyst Item Meta Data

The metadata of a Catalyst Item, including tag lists, may be updated at any time using the `osCltCmd_ModifyObjectMeta` operation. This operation is atomic and does not require a Catalyst Item be locked in order for it to be called. This means that the metadata can be updated whilst a Catalyst Item is open with a Catalyst Item Data Session and is being written to.

The Tag List section on page 77 recommends "Incomplete" and "Complete" tags is used to identify Objects. Calling `osCltCmd_ModifyObjectMeta` is the operation that should be used to change these tag lists.

8.2.2.1 Code Example – Modify Item Meta Data

```
int objSDKPrv_ModifyObjectMeta(const char *pServerAddress, const char
*pStoreKey, const char *pObjectKey, const char *pObjectTags, char
*pMetaData, uint32_t metaDataSize)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sKeyType storeKey;
      OSCMN_sKeyType objKey;
      OSCMN_sTagListType objectTagList;
      OSCMN_sObjectMetaType objectMetaBuffer;

      //<<Memset all structs>>
      //<<Set Store Key>>
      //<<Set Object Key>>

      if (pObjectTags)
      {
            strncpy(objectTagList.String,
                    pObjectTags,
                    sizeof(objectTagList.String));
            if (objectTagList.String[sizeof(objectTagList.String) - 1])
            {
                  printf("Error : Unable to set Object Tags %s\n",
                         objectTagList.String);
                  returnStatus = OSCLT_ERR_INTERNAL_ERROR;
                  return returnStatus;
            }
            objectTagList.StringSize = strlen(objectTagList.String);
      }
      // Metadata provided?
      if (pMetaData && metaDataSize)
      {
            objectMetaBuffer.MetaSize = metaDataSize;
            objectMetaBuffer.pMetaData = (uint8_t *) pMetaData;
      }
      //<<Open command session>>
      callStatus = osCltCmd_ModifyObjectMeta(pCmdSessionHandle,
                                             &storeKey,
                                             &objKey,
                                             &objectTagList,
                                             &objectMetaBuffer);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Modifying object '%s' in store '%s' failed "
                   "with error : %s.\n", pObjectKey, pStoreKey,
                    objSDKPrv_ErrorNumToString(callStatus));
            //<<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      printf("Modified object '%s' in store '%s' with new meta/tags.\n",
             pObjectKey, pStoreKey);
      //<<Close command session>>
      return returnStatus;
}
```

8.2.3 Deleting a Catalyst Item

When a backup expires the Catalyst Item containing the data should be deleted. A backup application may choose to batch multiple Catalyst Item deletion jobs and run them at predefined times. It is recommended where possible that delete operations are performed after backup jobs at least once per day. This ensures that matching data in the dedupe store is first referenced before being dereferenced by the delete and the disk space is freed up regularly.

If the client does not have access to that store the request will fail with error OSCLT_ERR_PERMISSION_DENIED.

8.2.3.1 Code Example – Delete an Item

```
int objSDKPrv_DeleteObject(const char *pServerAddress, const char
*pStoreKey, const char *pObjectKey)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sKeyType storeKey;
      OSCMN_sKeyType objKey;

      //<<Memset all structs>>
      //<<Set Store Key>>
      //<<Set Object Key>>

      //<<Open command session>>
      callStatus = osCltCmd_DeleteObject(pCmdSessionHandle,
                                         &storeKey,
                                         &objKey);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            if (callStatus != OSCLT_ERR_OBJECT_KEY_NOT_FOUND)
            {
                  printf("Error : Deleting object '%s' from store '%s'"
                         "failed with error : %s.\n", pObjectKey, pStoreKey,
                          objSDKPrv_ErrorNumToString(callStatus));
            }
            else
            {
                  printf("Error : Object '%s' does not exist in store"
                         " '%s'.\n", pObjectKey, pStoreKey);
            }
            //<<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      printf("Deleted object '%s' from store '%s'.\n", pObjectKey,
              pStoreKey);
      //<<Close command session>>
      return returnStatus;
}
```

8.2.4 Federated Catalyst Items

A Federated Catalyst Item created in a Federated Catalyst Store has the exact same properties and behaviours as a regular Catalyst Item in a regular Catalyst Store. Federated Catalyst Items should be interacted with using Federated Command Sessions and Federated Catalyst Item Data Sessions. For more information see the Federated Command Sessions section on page 25 and Federated Catalyst Item Data Sessions section on page 36.

8.3 Writing to Catalyst Items

8.3.1 Writing to a Catalyst Item

When writing to a Catalyst Item the typical sequence will be as follows:

```
osCltData_OpenObjectDataSession
osCltData_SeekToWriteBytes
osCltData_WriteBytes
osCltData_WriteBytes
osCltData_Flush
…
osCltData_WriteBytes
```

```
   osCltData_Flush
   osCltData_CloseObjectDataSession
```

Before opening the Catalyst Item data session to the Catalyst Item, the backup application should define the most appropriate values for the following settings:

- Bandwidth Mode
- Data Buffer Size (Recommend to use default)
- Payload Checksums
  - If the Catalyst Server address starts "COFC-" to transport over Fibre Channel Payload Checksums should be disabled.
  - If the backup will be performed over LAN checksums should be disabled
  - If the backup will be performed over WAN checksums should be enabled
- Compression
  - If the Catalyst Server address starts "COFC-" to transport over Fibre Channel Payload Compression should be disabled.
  - If the backup will be performed over LAN compression should be disabled
  - If the backup will be performed over WAN compression should be enabled

The bandwidth mode used should normally be specifiable by the backup application user. In cases where this is not possible the backup application should default to using the preferred mode as reported in `OSCMN_sWritePolicyType` when calling `osCltData_OpenObjectDataSession`.

Payload checksums are recommended when writing data to a Catalyst Store that resides on a different network segment, for example in a remote data centre. When writing data to a Catalyst Store that is accessible locally, payload checksums are not required and may be disabled to improve throughput performance, though this is not recommended.

Performing `osCltData_Flush` at frequent intervals during the backup allows a backup application to define checkpoints at which data will be confirmed to have been committed to disk. Knowing these checkpoints a backup application could restart a failed backup from a known point. It is recommended that the frequency of the flush occur no more often than each 10GB of data written, whichever occurs first. If `osCltData_Flush` is not called a default implicit flush frequency of 10GB will be used (1GB for server releases older than 3.16).

Once the backup application has selected the most appropriate values for the data session it should first check if the Catalyst Item already exists and if it does not it should create it with the tag of "Incomplete" and begin the backup. If the Catalyst Item does already exist then the backup application should check that the Items Storage Mode matches the intended write mode, it should also check if there are enough free sessions and lock resources available to the Item by getting the Catalyst Server Properties (page 54).

In the event that a backup application need to restart a backup from a known checkpoint the backup application should to open the session to the Catalyst Item and seek to write bytes at the offset of the known checkpoint and with the truncate flag set.

Once the backup is complete the Catalyst Item tag list and metadata should be updated using `osCltCmd_ModifyObjectMeta`. The "Incomplete" tag should be replaced with a tag of "Complete" and optionally the metadata content may be updated.

Note that the StoreOnce data immutability (WORM) mechanism uses the "Incomplete"/"Complete" tags in the object to know if it is immutable (only supported from server version 3.16). Integrators are encouraged to implement the Incomplete / Complete functionality when performing backups.

8.3.1.1 Code Example – Write Data to an Item

```
int objSDKPrv_WriteData(OSCLT_sSessionHandleType *pDataSessionHandle, char
*pBuffer, uint32_t bufferSize)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCMN_sBufferType suppliedData;

      memset(&suppliedData,
             0,
             sizeof(suppliedData));
      suppliedData.BufferSize = bufferSize;
      suppliedData.pBufferLocation = (uint8_t *) pBuffer;

      callStatus = osCltData_WriteBytes(pDataSessionHandle,
                                        &suppliedData);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Unable to write data of '%u' bytes in object,"
                   "error : %s.\n", suppliedData.BufferSize,
                    objSDKPrv_ErrorNumToString(callStatus));
            returnStatus = callStatus;
            return returnStatus;
      }
      return returnStatus;
}

int objSDKPrv_WriteGeneratedDataToObject(const char *pServerAddress, const
char *pStoreKey, const char *pObjectKey, uint64_t generatedDataSize)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType dataSessionHandle;

      int i;
      char *pBuffer = 0;
      uint64_t buffSize, bytesYetToWrite, bytesToWrite;

      OSCMN_IdType dataJobID;
      OSCMN_sObjectType object;

      buffSize = 2 * MB;
      pBuffer = (char *) malloc(buffSize);
      if (pBuffer == 0)
      {
            printf("Error : Unable to allocate memory for write "
                    "buffer.\n");
            returnStatus = OSCLT_ERR_INTERNAL_ERROR;
            return returnStatus;
      }
      // <<Open Object for Write>>
      printf("Write '%"PRIu64"' bytes of random data into source object
             "'%s' (store : %s) for copy.\n", generatedDataSize,
              pObjectKey, pStoreKey);

      i = 0;
      bytesYetToWrite = generatedDataSize;

      while(bytesYetToWrite)
      {
            bytesToWrite = (buffSize < bytesYetToWrite)?
                            buffSize : bytesYetToWrite;

            // <<Get data to write in pBuffer of side bytesToWrite>>
            // Write Data
            callStatus = objSDKPrv_WriteData(&dataSessionHandle, pBuffer,
                                             bytesToWrite);
            if (callStatus != OSCLT_ERR_SUCCESS)
            {
                  printf("Error : Unable to write random data to '%s'.\n",
                          pObjectKey);
                  // <<Close object and free buffer>>
                  return returnStatus;
            }

            bytesYetToWrite -= bytesToWrite;
            if ((i % 5) == 0)
            {
                  printf(".");
                  (void) fflush(stdout);
            }
            i++;
      }
      // <<Close object and free buffer>>
      return returnStatus;
}
```

8.3 Reading from Catalyst Items

8.3.2 Reading from a Catalyst Item

When reading from a Catalyst Item the typical sequence will be as follows:

```
osCltData_OpenObjectDataSession
osCltData_SeekToReadBytes
osCltData_ReadBytes
osCltData_ReadBytes
…
osCltData_ReadBytes
osCltData_CloseObjectDataSession
```

Before opening the Catalyst Item data session to the Catalyst Item, the backup application should define the most appropriate values for the following settings:

- Data Buffer Size (Recommend default)

- Payload Checksums
  - If the Catalyst server address starts “COFC-“ to transport over Fibre Channel Payload Checksums should be disabled.
  - If the backup will be performed over LAN checksums should be disabled
  - If the backup will be performed over WAN checksums should be enabled
- Compression
  - If the Catalyst server address starts “COFC-“ to transport over Fibre Channel Payload Compression should be disabled.
  - If the backup will be performed over LAN compression should be disabled
  - If the backup will be performed over WAN compression should be enabled

In normal operation the data buffer size that is specified should be `OSCLT_HIGH_BW_DATA_BUFFER_DEFAULT_SIZE`. However when reading data over a WAN connection it is recommended to increase the data buffer size to `MaximumLowBandwidthDataBufferSize` as returned by `osCltCmd_GetServerProperties`.

Payload checksums are recommended when reading data from a Catalyst Store that resides on a different network segment, for example in a remote data centre. When reading data from a Catalyst Store that is accessible locally, payload checksums are not required and may be disabled to improve throughput performance, though this is not recommended.

Read operations perform most efficiently when requested sequentially as so the backup application should aim to read from Objects sequentially starting at a defined offset. Backup applications that repeatedly seek and read small amounts of data will find that throughput performance is sub-optimal.

8.3.2.1 Code Example – Read Data from an Item

```
static int objSDKPrv_ReadObjectToFile(const char *pServerAddress, const
char *pStoreKey, const char *pObjectKey)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      int i;
      char *pRestoredData = 0;
      OSCMN_sObjectType object;
      OSCMN_IdType readDataJobID;
      OSCLT_sSessionHandleType dataSessionHandle;
      char *pFileName = pObjectKey;

      uint64_t buffSize, bytesYetToRead, bytesToRead;

      buffSize = 2 * MB;
      pRestoredData = (char *) malloc(buffSize);
      if (pRestoredData == 0)
      {
            printf("Error : Unable to allocate read buffer.\n");
            returnStatus = OSCLT_ERR_INTERNAL_ERROR;
            return returnStatus;
```

```c
    }

    // <<List object using Key>>

    printf("Open Object for Read.\n");
    // Open Object for Read
    callStatus = objSDKPrv_OpenObject(pServerAddress,
                                      pStoreKey,
                                      pObjectKey,
                                      false,
                                      0,
                                      &dataSessionHandle,
                                      &readDataJobID);
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
          printf("Error : Unable to open object '%s' for read.\n",
                  pObjectKey);
          returnStatus = callStatus;
          // <<Free restored data buffer>>
          return returnStatus;
    }

    // Open file
    FILE *file = fopen(pFileName, "w+");

    i = 0;
    bytesYetToRead= object.DataSize;

    while(bytesYetToRead)
    {
          // Get size of data to read
          bytesToRead = (buffSize < bytesYetToRead) ?
                         buffSize : bytesYetToRead;
          // Read data
          callStatus = objSDKPrv_ReadData(&dataSessionHandle,
                                          (uint8_t *) pRestoredData,
                                          &bytesToRead);
          if (callStatus != OSCLT_ERR_SUCCESS)
          {
                printf("Error : Unable to read from '%s'.\n",
                        pObjectKey);
                // <<Close object and close file>>
                // <<Free restore data buffer>>
                returnStatus = callStatus;
          return returnStatus;
          }

          // Write data to file
          callStatus = fprintf(file, pRestoredData);
          if (callStatus < 1)
          {
                printf("Error : Unable to write data (of size : %lu) "
                       "to file %s.\n",
                        sizeof(pRestoredData), pFileName);
                returnStatus = OSCLT_ERR_INTERNAL_ERROR;
                // <<Close object and close file>>
                // <<Free restore data buffer>>
                return returnStatus;
          }
```

```
            bytesYetToRead -= bytesToRead;
            if ((i % 5) == 0)
            {
                  printf(".");
                  (void) fflush(stdout);
            }
            i++;
      }
      printf("\n");
      // <<Close object and close file>>
      // <<Free restore buffer>>
      return returnStatus;
}
```

8.4 Cloning Data from a Catalyst Object

As well as writing new data via the backup application to a Catalyst object it is possible to clone data from previous Catalyst objects (referred to as parents) into a new Catalyst object (referred to a child). Cloning from previous objects allows a backup application to implement advanced functionality such as synthetic backups without needing to send all backup data from a client to StoreOnce when little data has changed since the previous backup. A typical workflow might be as follows where new and old data are considered:

```
osCltData_OpenObjectDataSession
osCltData_SeekToWriteBytes
osCltData_WriteBytes
osCltData_WriteBytes
osCltData_OpenParent
osCltData_SeekToCloneExtents
osCltData_CloneExtent
osCltData_CloneExtent
osCltData_SeekToWriteBytes
osCltData_WriteBytes
…
osCltData_Flush
osCltData_CloseObjectDataSession
```

Parent objects will be implicitly closed by the Catalyst server when the child data session is closed. However, it is best practise for to explicitly close parents, by calling osCltData_CloseParent.

Integrators are advised that there is overhead associated with switching between write and clone operations so switching modes should be limited, where possible. It is advised to supply large clone sizes where possible.

It is possible to open multiple parents to be used for cloning within a single data session. The maximum number of parents supported can be queried by merging the `MaximumOpenParentsPerCloneSession` client and server capability.

There may be cases where ISV's wish to check the integrity of a parent object prior to cloning from it. Where required the function osCltData_SeekToReadParentExtents and osCltData_ReadParentExtent can be used for such purposes.

8.5 Getting Data Transfer Information

Each Catalyst Item Data Session stores detailed statistics that may be queried by the backup application. This information includes for example transfer efficiency statistics that could be displayed to the user following a low bandwidth backup.

In order to retrieve the data job statistics, the backup application should use the osCltCmd_ListObjectDataJobs operation. It is recommended that the backup application calls this after each backup or restore, passing the JobID of each Catalyst Item Data Session that was used. The returned list of data jobs can then be parsed to provide the user with a consolidated efficiency metric.

If required a backup application may also request a list of data jobs using one or more of the following filters (see OSCMN_sObjectDataJobListFilterType definition for a full detailed list of available filters):

- Catalyst Item Key Filter
- Started Date Period Filter
- Stopped Date Period Filter
- Client Identifier Filter
- Client IP Address Filter
- Status Filter

The Catalyst Store shall respond with a list of all data job statistics that match all the filters applied. If there are more matching data job statistics to list than can be returned in the buffers provided, the PartialData flag shall be set to indicate that more data job listing is available to be retrieved.

In order to retrieve the next batch of the data job statistics, the backup application should repeat the osCltCmd_ListObjectDataJobs operation passing back the ListHandle returned in the previous response. The backup application should repeat this sequence until all the required data job statistics have been listed and the PartialData flag is set to false.

It is important to note that the ListHandle must be memset to zero whenever the first osCltCmd_ListObjectDataJobs call is made to avoid getting incomplete list responses.

The properties returned for a Data Job are:

Table 13 Dat Job Properties Supported Protocol

Property (OSCMN_sObjectDataJobType)	**Protocol v4**	**Protocol v5**	**Protocol v6**	**Protocol v7**	**Protocol v8**	**Protocol v9**
DataJobId	✓	✓	✓	✓	✓	✓
ObjectKey	✓	✓	✓	✓	✓	✓
JobReference		✓	✓	✓	✓	✓
DebugInfo	✓	✓	✓	✓	✓	✓
JobReceivedFromCredentials	✓	✓	✓	✓	✓	✓
JobReceivedFromAddress	✓	✓	✓	✓	✓	✓
JobStartedDateUTC	✓	✓	✓	✓	✓	✓
JobStoppedDateUTC	✓	✓	✓	✓	✓	✓
JobStatus	✓	✓	✓	✓	✓	✓
JobStatusCode	✓	✓	✓	✓	✓	✓
JobObjectDataSize	✓	✓	✓	✓	✓	✓
JobObjectDataStored		✓	✓	✓	✓	✓
JobStatistics	✓	✓	✓	✓	✓	✓
ClientPipelineStatistics	✓	✓	✓	✓	✓	✓
JobCancellationReason	✓	✓	✓	✓	✓	✓
PayloadChecksumsDisabled	✓	✓	✓	✓	✓	✓
BodyAndPayloadCompressionEnabled	✓	✓	✓	✓	✓	✓
DataJobTeaming			✓	✓	✓	✓
TransportProtocol						✓

8.5.1.1 Code Example – Get Job Status String

The Job Status should be translated to a user friendly string.

```
const char * objSDKPrv_GetJobStatusString(OSCMN_eJobStatusType jobStatus)
{
      switch (jobStatus)
      {
      case OSCMN_JOB_STATUS_PENDING :
            return "Pending";
      case OSCMN_JOB_STATUS_COMPLETED :
            return "Completed";
      case OSCMN_JOB_STATUS_RUNNING :
            return "Running";
      case OSCMN_JOB_STATUS_PAUSED :
            return "Paused";
      case OSCMN_JOB_STATUS_CANCELLED :
            return "Cancelled";
      case OSCMN_JOB_STATUS_INTERRUPTED :
            return "Interrupted";
      case OSCMN_JOB_STATUS_FROZEN :
            return "Frozen";
      default :
            return "Unknown";
      }
}
```

8.5.1.2 Code Example – List all Data Jobs from a Store

```
int objSDKPrv_ListAllDataJobsFromStore(const char *pServerAddress, const
char *pStoreKey)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sServerPropertiesType serverProp;
      OSCMN_sKeyType storeKey;
      OSCLT_sCompoundListHandleType listHandle;
      OSCMN_sObjectDataJobListFilterType objectDataJobListFilter;
      OSCMN_sObjectDataJobType *pJobList = 0;
      OSCLT_sObjectDataJobTeamingListConsistencyStatusType
                      *pObjDataJobTeamingListConsistencyStatusArray = NULL;
      bool partialData = false;
      uint32_t i, numObjectDataJobsReturned = 0;

      // <<Memset all structs | Setting Filter to list all jobs>>
      // <<Get sever properties>>

      // Allocate space for list jobs
      if ((pJobList = (OSCMN_sObjectDataJobType *) calloc(1,
            sizeof(OSCMN_sObjectDataJobType) *
                    serverProp.ServerCapabilities.
                            MaximumJobsPerListIteration)) == NULL)
      {
            printf("Error : Unable to allocate space for '%"PRIu64"'"
                   "data jobs.\n",
                    serverProp.ServerCapabilities.
                                            MaximumJobsPerListIteration);
            returnStatus = callStatus;
            return returnStatus;
      }
      // <<Set Store Key>>
      printf("List all data jobs from store : %s\n", pStoreKey);
```

```
       // <<Open command session>>
       do
       {
             callStatus = osCltCmd_ListObjectDataJobs (pCmdSessionHandle,
                                  &storeKey,
                                  &objectDataJobListFilter,
                                  false /* increasingOrder */,
                                  serverProp.ServerCapabilities.
                                                 MaximumJobsPerListIteration,
                                  pJobList,
                                  pObjDataJobTeamingListConsistencyStatusArray,
                                  &listHandle,
                                  &partialData,
                                  &numObjectDataJobsReturned);
             if (callStatus != OSCLT_ERR_SUCCESS)
             {
                   printf("Error : Listing jobs from store '%s' failed"
                          "with error : %s.\n",
                           pStoreKey,
                           objSDKPrv_ErrorNumToString(callStatus));
                   // <<Close command session & free list>>
                   returnStatus = callStatus;
                   return returnStatus;
             }
             for (i = 0; i < numObjectDataJobsReturned; i++)
             {
                   // Print Object Properties
                   printf("\t\tID:%"PRIu64", Object : %s, Status : %s.",
                           pJobList[i].DataJobId,
                           pJobList[i].ObjectKey.String,
                           objSDKPrv_GetCopyJobStatusString(
                                                 pJobList[i].JobStatus));
             }
       } while (partialData);
       // <<Close command session>>
       // <<Free list>>
       return returnStatus;
}
```

8.6 Copying a Catalyst Item

A Catalyst Item may be copied between Catalyst Stores in order to allow backup application to keep additional copies of backup data in multiple locations. These copies are performed asynchronously by the Catalyst Server in a bandwidth efficient way ensuring that only unique data is transmitted between locations.

A typical sequence would be:

- Create an empty Catalyst Item in the destination Catalyst Store using `osCltCmd_CreateObject`.
- Queue the asynchronous copy using `osCltCmd_QueueObjectCopyJob`.
- Monitor copy progress using osCltCmd_ListOriginObjectCopyJobs.

Copy jobs can be performed over Fibre Channel or Ethernet.

See Catalyst Item Copy Job Process () on page 103 for the process flow of a Catalyst Item Copy Job.

8.6.1 Queuing a Catalyst Item Copy Job

A backup application requests a Catalyst Item is copied using the `osCltCmd_QueueObjectCopyJob` operation. This will return an `OriginObjectCopyJobId` and add the copy request to a queue to be processed when resources permit. The backup application should then poll the Catalyst Server using the `osCltCmd_ListOriginObjectCopyJobs` operation, passing back an array of `OriginObjectCopyJobId`'s to track. The backup application can then collate the progress of the copy jobs and report back to the user as appropriate.

8.6.1.1 Code Example – Queue a Copy Job

```
int objSDKPrv_QueueCopy(const char *pSrcServerAddr, const char
*pSrcStoreKey, const char *pSrcObjectKey, uint64_t srcExtOffset, uint64_t
srcExtLength, const char *pDesServerAddr, const char *pDesStoreKey, const
char *pDesObjectKey, uint64_t desOffset, OSCMN_IdType *pCopyJobID)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      const char *pServerAddress = 0;
      OSCMN_sKeyType srcStoreKey;
      OSCMN_sKeyType srcObjectKey;
      OSCMN_sKeyType desStoreKey;
      OSCMN_sKeyType desObjectKey;
      OSCMN_sAddressType desAddress;
      OSCMN_sExtentType srcObjExtent;
      OSCMN_sStringType pJobRef;

      // <<Memset all structs>>

      // <<Set Source Catalyst Server Address, Item Key,
      // Extent Offset, Extent Length>>
      // <<Set Destination Catalyst Server Address, Store Key, Item Key>>

      memset(&pJobRef,
             0,
             sizeof(pJobRef));

      strncpy(pJobRef.String,
              OBJSDK_JOB_REF,
              sizeof(pJobRef.String));

      if (pJobRef.String[sizeof(pJobRef.String) - 1])
      {
            printf("Error : Unable to set Job Reference %s\n",
                    pJobRef.String);
            returnStatus = OSCLT_ERR_INTERNAL_ERROR;
            return returnStatus;
      }
      pJobRef.StringSize = strlen(pJobRef.String);
      // <<Open command session>>
      callStatus = osCltCmd_QueueObjectCopyJob(pCmdSessionHandle,
                                                &srcStoreKey,
                                                &srcObjectKey,
                                                &srcObjExtent,
                                                &desAddress,
                                                OSCLT_DEFAULT_COMMAND_PORT,
                                                OSCLT_DEFAULT_DATA_PORT,
                                                &desStoreKey,
                                                    &desObjectKey,
                                                    desOffset,
                                                    &pJobRef,
                                                    pCopyJobID);

      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Queuing copy job from object '%s' (store : %s)"
                   "to object '%s' (store : %s) failed with error : %s.\n",
                    pSrcObjectKey, pSrcStoreKey, pDesObjectKey,
                    pDesStoreKey, objSDKPrv_ErrorNumToString(callStatus));
            // <<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
    // <<Close command session>>
    return returnStatus;
}
```

8.6.2 Catalyst Item Copy Job Records

When polling the Catalyst Store for copy job status information using the `osCltCmd_ListOriginObjectCopyJobs` operation, it will return a `SuggestedSecondsWaitBeforeNextPoll` value. This value is calculated by the Catalyst Server and it represents the next time at which any of the queried copy jobs are expected to have changed state or updated progress. It is recommended that a backup application waits the suggested period before polling for status information again. This will avoid putting unnecessary load on the Catalyst Server when no change of state or progress is expected to have occurred.

Copy Job's will be on one of the following states:

Table 14 - Possible Copy Job Status

Copy Job Status	Description
Pending	The copy job is queued but has not yet started.
Running	The copy job is currently running.
Paused	The copy job was started but is no longer running. It will back to a running state when resources allow.
Cancelled	The copy job was cancelled either because it was cancelled by the user or because it was an invalid request.
Completed	The copy job completed successfully.
Interrupted	The copy job was interrupted (by a power failure). This is a transitional state and will change to Paused/Cancelled once the Catalyst Store comes online.

All copy jobs will eventually move to a Cancelled or Completed state. If a copy job is in a Cancelled state the backup application should check the `LastJobPauseCancelReason` value returned to identify the cause of the cancellation and report back to the user an appropriate error message.

8.6.2.1 Code Example – Get Copy Job Record from ID

```
int objSDKPrv_GetCopyJobByID(const char *pServerAddress, const char
*pStoreKey, bool storeIsOrigin, OSCMN_IdType copyJobID,
OSCMN_sObjectCopyJobType *pCopyJob, uint32_t *pSuggestedWaitForNextPoll)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
```

```
    OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
    OSCMN_sKeyType storeKey;
    OSCMN_sObjectCopyJobListFilterType objCopyJobListFilter;
    OSCMN_sListHandleType listHandle;
    uint32_t numJobsReturned;
    bool partialData;

    // <<Memset all structs>>
    // <<Set Store Key>>
    // <<Open command session>>

    if (storeIsOrigin)
    {
          // List at origin
          objCopyJobListFilter.OriginObjectCopyJobIdFilterArray[0] =
                                                                copyJobID;
          callStatus = osCltCmd_ListOriginObjectCopyJobs(
                                                pCmdSessionHandle,
                                                &storeKey,
                                                &objCopyJobListFilter,
                                                false /* increasingOrder */,
                                                1,
                                                pCopyJob,
                                                &listHandle,
                                                &partialData,
                                                &numJobsReturned,
                                                pSuggestedWaitForNextPoll);
    }
    else
    {
          // List at Destination
          objCopyJobListFilter.DestinationObjectCopyJobIdFilterArray[0] =
                                                                copyJobID;
          // NOTE : List at destination don't suggest next poll time
          callStatus = osCltCmd_ListDestinationObjectCopyJobs(
                                                pCmdSessionHandle,
                                                &storeKey,
                                                &objCopyJobListFilter,
                                                false /* increasingOrder */,
                                                1 ,
                                                pCopyJob,
                                                &listHandle,
                                                &partialData,
                                                &numJobsReturned);
    }
    if (callStatus != OSCLT_ERR_SUCCESS)
    {
          printf("Error : Listing copy job from store '%s' failed"
                 "with error : %s.\n", pStoreKey,
              objSDKPrv_ErrorNumToString(callStatus));
          // <<Close command session>>
          returnStatus = callStatus;
          return returnStatus;
    }
    if (partialData)
    {
          printf("Error : Listing copy job using ID from store '%s'"
                 "returned with partialData set to true.\n", pStoreKey);
          // <<Close command session>>
          returnStatus = OSCLT_ERR_INTERNAL_ERROR;
```

```
                return returnStatus;
          }
          if (numJobsReturned != 1)
          {
                printf("Error : Listing copy job using ID from store '%s'"
                       "returned '%u' jobs.\n", pStoreKey, numJobsReturned);
                // <<Close command session>>
                returnStatus = OSCLT_ERR_INTERNAL_ERROR;
                return returnStatus;
          }
        // <<Close command session>>
        return returnStatus;
}
```

8.6.3 Cancelling Catalyst Item Copy Jobs

If the copy job is not in a Completed state the backup application may cancel a copy job by issuing `osCltCmd_CancelOriginObjectCopyJob` to the Catalyst Store from which the copy is being sent. And if the copy job had started the backup application should also issue `ocCltCmd_CancelDestinationObjectCopyJob` to the Catalyst Store to which the copy was being sent. This will ensure that the Inbound Copy Job record on the destination Catalyst Store is appropriately set to a cancelled state.

Once the backup application has issued the request to cancel the copy job, it should continue to poll the copy job waiting for the state to change to cancelled. Depending on the link speed, it may take several minutes to change state. Once the copy job is in a cancelled state, the Catalyst Item lock will be released and the backup application should delete the destination Catalyst Item. Failure to delete the cancelled destination Catalyst Item will result in orphaned Objects which a user will manually have to remove.

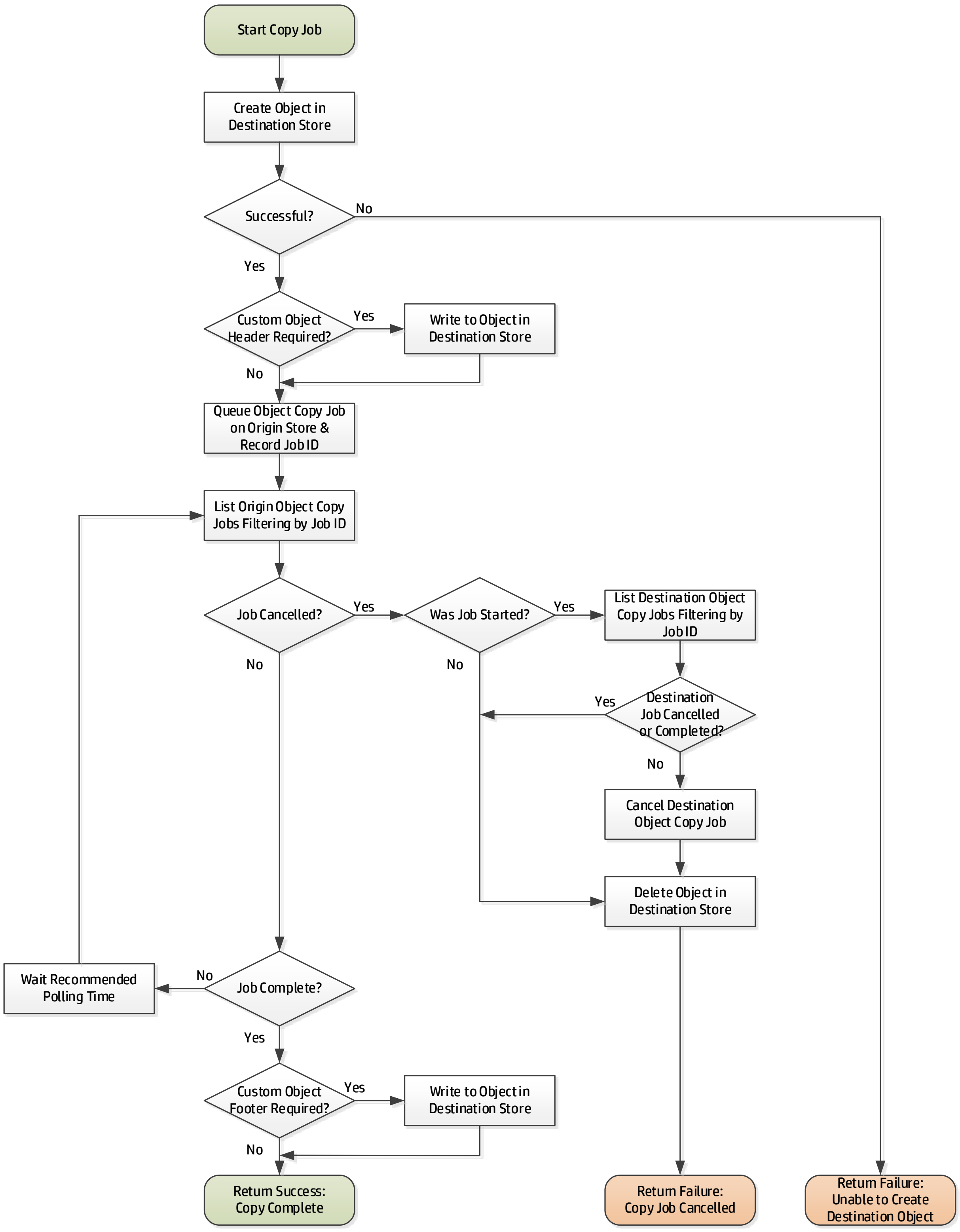
If required a backup application may choose to only copy a specific section of a Catalyst Item, for example, copy the user data region of a Catalyst Item but skip the header information. In these instances the backup application should still create the destination Catalyst Item but at this stage may also write new header information to it. Then when calling `osCltCmd_QueueObjectCopyJob` the backup application should provide an offset for the source and destination Objects that defines the start of the region to be copied.

8.6.3.1 Code example – Cancelling a Copy Job

```
int objSDKPrv_CancelCopy(const char *pServerAddress, const char *pStoreKey,
bool storeIsOrigin, OSCMN_IdType copyJobID)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      OSCMN_sKeyType storeKey;
      OSCMN_sObjectCopyJobType copyJob;
      uint32_t suggestedWaitForNextPoll;

      // <<Memset all structs>>
      // <<Set Store Key>>
      // <<Open command session>>
      // <<Get Copy Job Details | See 'Catalyst Items Copy Jobs Record'>>

      if (copyJob.JobStatus == OSCMN_JOB_STATUS_COMPLETED ||
            copyJob.JobStatus == OSCMN_JOB_STATUS_CANCELLED)
      {
            printf("Error : Job can't be cancelled, job is"
                   "in '%s' state.\n", objSDKPrv_GetCopyJobStatusString(
                   copyJob.JobStatus));
            // <<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      // Now, Cancel the job
      if (storeIsOrigin)
      {
            // Cancel at Origin
            callStatus = osCltCmd_CancelOriginObjectCopyJob(
                                                         pCmdSessionHandle,
                                                         &storeKey,
                                                         copyJobID);
      }
      else
      {
            // Cancel at Destination
            callStatus = osCltCmd_CancelDestinationObjectCopyJob(
                                     pCmdSessionHandle,
                                     &storeKey,
                                     copyJob.DestinationObjectCopyJobId);
      }
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Failed to cancel copy job at %s store '%s', "
                   "error : %s.", storeIsOrigin? "origin" : "destination\n",
                    pStoreKey, objSDKPrv_ErrorNumToString(callStatus));
            // <<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      // <<Close command session>>
      return returnStatus;
}
```

8.6.4 Catalyst Item Copy Job Process

Shows the process of starting a Catalyst Item Copy Job and the possible states it can end it.


Start Copy Job
Create Object in Destination Store
Successful?
No
Yes
Custom Object Header Required?
Yes
Write to Object in Destination Store
No
Queue Object Copy Job on Origin Store & Record Job ID
List Origin Object Copy Jobs Filtering by Job ID
Job Cancelled?
Yes
Was Job Started?
Yes
List Destination Object Copy Jobs Filtering by Job ID
No
No
Yes
Destination Job Cancelled or Completed?
No
Cancel Destination Object Copy Job
Delete Object in Destination Store
Wait Recommended Polling Time
No
Job Complete?
Yes
Custom Object Footer Required?
Yes
Write to Object in Destination Store
No
Return Success: Copy Complete
Return Failure: Copy Job Cancelled
Return Failure: Unable to Create Destination Object
Figure 21 - Catalyst Item Copy Operation Flow Diagram

8.6.5 Copy Job Records

Every Catalyst Copy operation will create a copy job record. A copy job record details the meaningful information about a Catalyst copy session. Their purpose is to allow Catalyst users the ability to review information about their Catalyst copy session. An example use would be for a Catalyst object which is copied from source to target. Both the source and target Catalyst objects will have copy job records created. The source will create an “Outbound Copy Job” record and the target an “Inbound Copy Job” record. The information recorded is similar to the above Data Job Records, including: object name, client IP address from which the session was written, the amount of data written, dedupe ratio; among others.

Copy jobs records have an expiration period. This expiration period is defined at a per Catalyst store level. The client can set the retention period via `OSCMN_sJobRetentionPolicyType`. It should be set to the default `OSCMN_JOB_RETENTION_POLICY_DEFAULT_DAYS` (90 days) unless the user selects otherwise. The Retention periods must be between 1 and `MaximumJobRetentionPeriod` in Server Capabilities.

The Catalyst Server stores the following details for Copy Jobs that are in progress or have ended:

Table 15 - Copy Job Properties Supported Protocol

Property (OSCMN_sObjectCopyJobType)	Protocol v4	Protocol v5	Protocol v6	Protocol v7	Protocol v8	Protocol v9
`OriginObjectCopyJobId`	✓	✓	✓	✓	✓	✓
`DestinationObjectCopyJobId`	✓	✓	✓	✓	✓	✓
`OriginServerAddress`	✓	✓	✓	✓	✓	✓
`OriginServerSerialNumber`	✓	✓	✓	✓	✓	✓
`OriginObjectStoreKey`	✓	✓	✓	✓	✓	✓
`OriginObjectKey`	✓	✓	✓	✓	✓	✓
`OriginObjectDataSize`	✓	✓	✓	✓	✓	✓
`OriginObjectLastModifiedDateUTC`	✓	✓	✓	✓	✓	✓
`OriginObjectStorageMode`	✓	✓	✓	✓	✓	✓
`OriginObjectExtent`	✓	✓	✓	✓	✓	✓
`DestinationServerAddress`	✓	✓	✓	✓	✓	✓
`DestinationServerSerialNumber`	✓	✓	✓	✓	✓	✓
`DestinationServerCommandPort`	✓	✓	✓	✓	✓	✓
`DestinationServerDataPort`	✓	✓	✓	✓	✓	✓
`DestinationCredentials`						✓
`DestinationObjectStoreKey`	✓	✓	✓	✓	✓	✓
`DestinationObjectKey`	✓	✓	✓	✓	✓	✓
`DestinationObjectOffset`	✓	✓	✓	✓	✓	✓
`JobReference`		✓	✓	✓	✓	✓
`JobQueuedFromCredentials`	✓	✓	✓	✓	✓	✓
`JobQueuedFromAddress`	✓	✓	✓	✓	✓	✓
`JobQueuedDateUTC`	✓	✓	✓	✓	✓	✓
`JobStartedDateUTC`	✓	✓	✓	✓	✓	✓
`JobStoppedDateUTC`	✓	✓	✓	✓	✓	✓
`JobStatus`	✓	✓	✓	✓	✓	✓
`JobStatusCode`	✓	✓	✓	✓	✓	✓
`JobObjectDataSize`	✓	✓	✓	✓	✓	✓
`JobBytesCopied`	✓	✓	✓	✓	✓	✓

`JobPercentageComplete`	✓	✓	✓	✓	✓	✓
`JobStatistics`	✓	✓	✓	✓	✓	✓
`OriginServerTransportStatistics`		✓	✓	✓	✓	✓
`LastJobPauseCancelReason`		✓	✓	✓	✓	✓
`LastJobFailedRunReason`	✓	✓	✓	✓	✓	✓
`NumberOfFailedRunAttemptsSinceLastRun`	✓	✓	✓	✓	✓	✓
`LastFailedRunAttemptDateUTC`	✓	✓	✓	✓	✓	✓
`NextRunAttemptDateUTC`	✓	✓	✓	✓	✓	✓
`IsMarkedForCancellation`	✓	✓	✓	✓	✓	✓
`EstimatedCompletionDateUTC`	✓	✓	✓	✓	✓	✓
`BodyAndPayloadCompressionEnabled`		✓	✓	✓	✓	✓
`TeamedOriginObjectDataJobUUID`		✓	✓	✓	✓	✓
`TeamedDestinationObjectDataJobUUID`		✓	✓	✓	✓	✓
`TransportProtocol`						✓
`OriginWIPOffset`						✓
`DestinationWIPOffset`						✓

* Protocol v5 will return Federated (Teaming) values in structures but they will always be false.

Section 9: StoreOnce Catalyst Authentication

9.1 Client Authentication

9.1.1 Client Access Permissions

There are two factors that affect access permissions:

- Vendor Licensing
- Client Permissions

If Client Access Permission Checking is not enabled on the Catalyst Server, then all clients are allowed to access all Catalyst Stores, create new Catalyst Stores, modify server properties, and manage client permissions. If the Catalyst Server administrator does not want clients to have this level of access, they should enable client permissions and lock down the permissions for different clients (based on Client Identifier) as desired.

When permissions checking is first enabled a guest Client Access Permission record is created and given permission to create new Catalyst Stores, modify server properties, and manage client permissions, so if the Catalyst Server administrator does not want all clients to inherit these permissions, they should remove them from guest. This enables the server to be managed by a client if desired, and the client can then lock down permissions appropriately. If the server administrator does not want clients to be able to manage the server, they should enable permissions via the server's management interface and lock down permissions from there.

When establishing a command or data session a caller must supply credentials. The credentials consist of three strings; a vendor string, an identifier string and a password.

9.1.1.1 Vendor String

The vendor string is a fixed string that identifies the application to the Catalyst Server. The vendor string will be provided by Hewlett-Packard Enterprise and should not be modified. Failure to pass the vendor string to the Catalyst Server may limit or prevent access to Catalyst Stores.

When a command session has been established with the server it is recommended to check whether the vendor is licensed by getting the server properties and checking the returned Booleans for `IsClientLicensed`, `IsClientTeamedLicensed` and `IsClientCoFCLicensed`. See Getting Server Properties on page 54.

9.1.1.2 Identifier String

The identifier string is a text string used to identify one or more Catalyst Clients. Using these identifier strings it is possible within the Catalyst Server GUI to control individual access to each of the Catalyst Stores. Within the Catalyst Server GUI this is called Client Permissions.

By default, client permissions are not enabled and therefore all Catalyst Stores are considered to be accessible by guests. A Catalyst Client passing an empty string as an Identifier is also considered to be a guest and will therefore have full access to the Catalyst Stores.

Once client permissions are enabled, only Catalyst Stores for which the Catalyst Client is listed as having access and Catalyst Stores marked for guest access will be accessible.

The calling application should provide a capability to their users to allow them to specify which Identifier String to use for connections. The user should create a new client identifier string within the Catalyst Server GUI and then supply the same identifier string to the backup application.

9.1.1.3 Password String

The password string is a text string used to authenticate a Catalyst Client. The password can be any UTF8 non control character and has a maximum length of 256 characters.

9.1.1.4 Complete Code Example – Authenticating the Client

```
#define OBJSDK_DEFAULT_VENDOR_NAME "HPD2DTools"
#define OBJSDK_DEFAULT_IDENTIFIER "SDKExampleCode"

OSCMN_sCredentialsType cred;
memset(&cred, 0, sizeof(cred));

strncpy(cred.VendorString, OBJSDK_DEFAULT_VENDOR_NAME,
        sizeof(cred.VendorString));
cred.VendorString[sizeof(cred.VendorString) - 1] = 0;
cred.VendorStringSize = strlen(cred.VendorString);

strncpy(cred.IdentifierString, OBJSDK_DEFAULT_IDENTIFIER,
        sizeof(cred.IdentifierString));
cred.IdentifierString[sizeof(cred.IdentifierString) - 1] = 0;
cred.IdentifierStringSize = strlen(cred.IdentifierString);
```

9.1.2 Managing Access Permission

Client access permissions are most commonly managed through the web management interface of the StoreOnce appliance. However a StoreOnce appliance, which supports StoreOnce Catalyst v5 or above, may be configured so that client access permissions may be remotely managed using software that integrates StoreOnce Catalyst. Software wishing to provide this remote management of client access permissions will need to support the following functions:

- osCltCmd_SetClientPermissionsChecking
- osCltCmd_ListClientPermissions
- osCltCmd_CreateClientPermissions
- osCltCmd_ModifyClientPermissions
- osCltCmd_ModifyClientPassword
- osCltCmd_DeleteClientPermissions
- osCltCmd_ListObjectStoreClientPermissions
- osCltCmd_AddObjectStoreClientPermission
- osCltCmd_RemoveObjectStoreClientPermission

Support of these functions with a backup application is considered advanced integration and is not undertaken by the majority of integrators.

9.1.2.1 Checking Remote Management Capability

A backup application can check whether it has been granted permission to remotely manage client access permissions by checking the `CanClientManageClientPermissions` Boolean returned by `osCltCmd_GetServerProperties`.

If `CanClientManageClientPermissions` is true, then the software is able to use the commands listed above, otherwise the functions will return permission denied when called.

9.1.2.2 Enabling/Disabling Client Permissions Checking

To determine whether client permissions checking is enable, a backup application should check the `IsClientPermissionsCheckingEnabled` Boolean returned by `osCltCmd_GetServerProperties`.

To enable or disable client permissions checking the `osCltCmd_SetClientPermissionsChecking` function should be called with the `IsClientPermissionsCheckingEnabled` input set to True to enable client permission checking or False to disable client permissions checking.

9.1.2.2.1 Code Example – Enabling or Disabling Client Permission Checking

```
int objSDKPrv_SetClientPermissionChecking(const char *pServerAddress, bool
clientPermissionChecking)
{
	int returnStatus = OSCLT_ERR_SUCCESS;
	int callStatus = OSCLT_ERR_SUCCESS;

	OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
	//<<Open command session>>
	callStatus = osCltCmd_SetClientPermissionsChecking(
	                                      pCmdSessionHandle,
	                                      clientPermissionChecking);
	if (callStatus != OSCLT_ERR_SUCCESS)
	{
		printf("Error : Failed to set client permission : %s.\n",
		       objSDKPrv_ErrorNumToString(callStatus));
		//<<Close command session>>
		returnStatus = callStatus;
		return returnStatus;
	}
	//<<Close command session>>
	return returnStatus;
}
```

9.1.2.3 List Client Permissions Records

To allow customers to manage client permissions it is necessary to list all client permission records configured on the Catalyst Server. The backup application must have permission to manage client permissions.

Listing all client permissions can only be done using a non-federated command session. Depending on the number of client permission records it might not be possible to return the entire list in one operation, if the full list cannot be returned Partial Data will be returned as true. The called can then call again with the List Handle set to the value returned in the previous call to continue the returned list from where the previous call reached.

9.1.2.3.1 Code Example – Listing Client Permission Records on Server

```
int objSDKPrv_ListClientPermissions(const char *pServerAddress)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;

    OSCLT_sSessionHandleType *pCmdSessionHandle = 0;
    bool partialData = true;
    uint32_t i, numClientPermissionsReturned = 0;
    OSCMN_sListHandleType listHandle;
    int maxClientPermissions = 10;
    OSCMN_sClientPermissionsType *pClientPermissions = NULL;

    memset(&listHandle,
           0,
           sizeof(listHandle));

    //<<Calloc pClientPermissions>>
    printf("List all permissions from server : %s\n", pServerAddress);
    do
    {
        //<<Open command session>>
        callStatus = osCltCmd_ListClientPermissions(pCmdSessionHandle,
                                                    maxClientPermissions,
                                                    pClientPermissions,
                                                    &listHandle,
                                                    &partialData,
                                                    &numClientPermissionsReturned);
        if (callStatus != OSCLT_ERR_SUCCESS)
        {
            printf("Error : Listing permissions from server failed"
                   "with error : %s.\n",
                   objSDKPrv_ErrorNumToString(callStatus));
            // <<Close command session>>
            // <<Free pClientPermissions>>
            returnStatus = callStatus;
            return returnStatus;
        }
        // <<Close command session>>
        for (i = 0; i < numClientPermissionsReturned; i++)
        {
            // Print permissions list
            printf("Identifier \t\t\t: %s\n",
                   pClientPermissions[i].ClientIdentifier.StringSize
                           == 0 ? "GUEST" :
                           pClientPermissions[i].ClientIdentifier.String);
            printf("Description \t\t\t: %s\n",
                   pClientPermissions[i].Description.String);
            printf("Create ObjStores \t\t: %s\n",
                   pClientPermissions[i].CanCreateObjectStores ?
                   "Yes" : "No");
            printf("Set Server Properties \t\t: %s\n",
                   pClientPermissions[i].CanSetServerProperties ?
                   "Yes" : "No");
            printf("Manage Permissions \t\t: %s\n",
                   pClientPermissions[i].CanManageClientPermissions ?
                   "Yes" : "No");
            printf("\n");
        }
        // Loop until partial data set to true
    } while (partialData);
    //<<Free pClientPermissions>>
    return returnStatus;
}
```

9.1.2.4 List Catalyst Store Access Permissions

The backup application may allow customers to view the client permissions that currently have access to a specific Catalyst Store. The backup application must have permission to manage client permissions.

9.1.2.4.1 Code Example – Listing Client Store Permissions on Server

```
int objSDKPrv_ListClientStorePermissions(const char *pServerAddress, const
char *pStoreKey)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;
      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      bool partialData = true;
      uint32_t i, numClientPermissionsReturned = 0;
      OSCMN_sListHandleType listHandle;
      int maxClientPermissions = 10;
      OSCMN_sClientPermissionsType *pClientPermissions = NULL;
      OSCMN_sKeyType storeKey;
      OSCMN_sStringType *pClientIdentifiers = NULL;

      // <<Memset list handle and Store Key>>
      // <<Malloc pClientPermissions and pClientIdentifiers>>
      // <<Set Store Key>>
      printf("List all permissions for store : %s\n", pStoreKey);
      do
      {
            // <<Open command session>>
            pCmdSessionHandle = &cmdSessionHandle;
            callStatus = osCltCmd_ListObjectStoreClientPermissions(
                                                pCmdSessionHandle,
                                                &storeKey,
                                                maxClientPermissions,
                                                pClientIdentifiers,
                                                &listHandle,
                                                &partialData,
                                                &numClientPermissionsReturned);
            if (callStatus != OSCLT_ERR_SUCCESS)
            {
                  printf("Error : Listing permissions for store '%s'"
                         "failed with error : %s.\n", pStoreKey,
                          objSDKPrv_ErrorNumToString(callStatus));
                  // <<Close command session>>
                  // <<Free memory allocations>>
                  returnStatus = callStatus;
                  return returnStatus;
            }
            // <<Close command session>>
            for (i = 0; i < numClientPermissionsReturned; i++)
            {
                  // Print permissions list
                  printf("Identifier \t\t\t: %s\n",
                      pClientPermissions[i].ClientIdentifier.StringSize
```

```
                                  == 0 ? "GUEST" :
                                     pClientPermissions[i].ClientIdentifier.String);
                        printf("Description \t\t\t: %s\n",
                                   pClientPermissions[i].Description.String);
                        printf("Create ObjStores \t\t: %s\n",
                                   pClientPermissions[i].CanCreateObjectStores ?
                                     "Yes" : "No");
                        printf("Set Server Properties \t\t: %s\n",
                                  pClientPermissions[i].CanSetServerProperties ?
                                      "Yes" : "No");
                        printf("Manage Permissions \t\t: %s\n",
                                  pClientPermissions[i].CanManageClientPermissions ?
                                      "Yes" : "No");
                        printf("\n");
                  }
                  // Loop until partial data set to true
            } while (partialData);
            // <<Free memory allocations>>
            return returnStatus;
}
```

9.1.2.5 Creating a Client Permissions Record

The backup application may allow customers to create client permissions on the Catalyst Server. The backup application must have permission to manage client permissions. Creating a client permission record can only be done using a non-federated command session.

9.1.2.5.1 Code Example – Creating a new Client Permission Record

```
int objSDKPrv_CreateClientPermissions(const char *pServerAddress,
OSCMN_sClientPermissionsType clientPermissions, OSCMN_sStringType
clientPassword)
{
    int returnStatus = OSCLT_ERR_SUCCESS;
    int callStatus = OSCLT_ERR_SUCCESS;

    OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;

    printf("Creating permissions for %s using password : \"%s\"\n",
            clientPermissions.ClientIdentifier.String,
            clientPassword.String);

    //<<Open command session>>

    callStatus = osCltCmd_CreateClientPermissions(pCmdSessionHandle,
                                                  &clientPermissions,
                                                  &clientPassword);
    if (callStatus != OSCLT_ERR_SUCCESS && callStatus !=
                                     OSCLT_ERR_DUPLICATE_CLIENT_IDENTIFIER)
    {
        printf("Error : Creating permissions on server '%s'"
               "failed with error : %s.\n", pServerAddress,
                objSDKPrv_ErrorNumToString(callStatus));
        //<<Close command session>>
        returnStatus = callStatus;
        return returnStatus;
    }
    if (callStatus == OSCLT_ERR_DUPLICATE_CLIENT_IDENTIFIER)
    {
        printf("%s already exists\n",
                clientPermissions.ClientIdentifier.String);
    }
    if (callStatus == OSCLT_ERR_SUCCESS)
    {
        printf("Permissions successfully created for %s\n",
               clientPermissions.ClientIdentifier.String);

    }
    //<<Close command session>>
    return returnStatus;
}
```

9.1.2.6 Modifying a Client Permissions Record

The backup application may allow customers to modify previously configured client permissions on the Catalyst Server. The backup application must have permission to manage client permissions. Modifying a client permission record can only be done using a non-federated command session.

9.1.2.6.1 Code Example – Modify an existing Client Permission Record

```
int objSDKPrv_ModifyClientPermissions(const char *pServerAddress,
OSCMN_sClientPermissionsType clientPermissions)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;

      printf("Changing permissions for %s\n",
             clientPermissions.ClientIdentifier.String);

      //<<Open command session>>

      callStatus = osCltCmd_ModifyClientPermissions(pCmdSessionHandle,
                                                    &clientPermissions);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Changing permissions on server '%s'"
                   "failed with error : %s.\n", pServerAddress,
                    objSDKPrv_ErrorNumToString(callStatus));
            //<<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      printf("Permissions successfully changed\n\n");
      //<<Close command session>>
      return returnStatus;
}
```

9.1.2.7 Modifying a Client Password

The backup application may allow customers to modify the password of a previously configured client permission on the Catalyst Server. The backup application must have permission to manage client permissions. Modifying the password can only be done using a non-federated command session.

9.1.2.7.1 Code Example – Modifying a Client Permission Records Password

```
int objSDKPrv_ModifyClientPassword(const char *pServerAddress,
OSCMN_sStringType clientIdentifier, OSCMN_sStringType clientPassword)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      //<<Open command session>>
      callStatus = osCltCmd_ModifyClientPassword(pCmdSessionHandle,
                                                 &clientIdentifier,
                                                 &clientPassword);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Changing password on server '%s' failed"
                   "with error : %s.\n",  pServerAddress,
                    objSDKPrv_ErrorNumToString(callStatus));
            //<<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      printf("Password successfully changed\n\n");
      //<<Close command session>>
      return returnStatus;
}
```

9.1.2.8 Delete a Client Permissions Record

The backup application may allow customers to delete configured client permission records on the Catalyst server. The backup application must have permission to manage client permissions. Deleting a client permission record can only be done using a non-federated command session.

9.1.2.8.1 Code Example – Delete a Client Permission Record

```
int objSDKPrv_DeleteClientPermissions(const char *pServerAddress,
OSCMN_sStringType clientIdentifier)
{
      int returnStatus = OSCLT_ERR_SUCCESS;
      int callStatus = OSCLT_ERR_SUCCESS;

      OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
      //<<Open command session>>
      callStatus = osCltCmd_DeleteClientPermissions(pCmdSessionHandle,
                                                   &clientIdentifier);
      if (callStatus != OSCLT_ERR_SUCCESS)
      {
            printf("Error : Deleting %s's permissions failed"
                   "with error : %s.\n", pServerAddress,
                    objSDKPrv_ErrorNumToString(callStatus));
            //<<Close command session>>
            returnStatus = callStatus;
            return returnStatus;
      }
      printf("Successfully deleted %s\n\n", clientIdentifier.String);
      //<<Close command session>>
      return returnStatus;
}
```

9.1.2.9 Enabling Catalyst Store Access

The backup application may allow customers to give a client access to a specific Catalyst Store. The backup application must have permission to manage client permissions. Assigning permissions can only be done using a non-federated command session.

9.1.2.9.1 Code Example – Enabling a Client Permission Records access to a Store

```
int objSDKPrv_AddObjectStorePermissions(const char *pServerAddress, const
char *pObjectStoreKey, OSCMN_sStringType clientIdentifier)
{
	int returnStatus = OSCLT_ERR_SUCCESS;
	int callStatus = OSCLT_ERR_SUCCESS;

	OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
	OSCMN_sKeyType objStoreKey;

	//<<Set ObjectStore key>>

	printf("Adding %s permissions to store %s\n",
		clientIdentifier.String, objStoreKey.String);

	//<<Open command session>>

	callStatus = osCltCmd_AddObjectStoreClientPermission(
									pCmdSessionHandle,
									&objStoreKey,
									&clientIdentifier);
	if (callStatus != OSCLT_ERR_SUCCESS)
	{
		printf("Error : Adding %s's permissions failed with"
		       "error : %s.\n", pServerAddress,
			objSDKPrv_ErrorNumToString(callStatus));
		//<<Close command session>>
		returnStatus = callStatus;
		return returnStatus;
	}
	printf("Successfully added %s to %s\n\n", clientIdentifier.String,
		objStoreKey.String);
	//<<Close command session>>
	return returnStatus;
}
```

9.1.2.10 Disabling Catalyst Store Access

The backup application may allow customers to disable a clients access to a specific Catalyst Store. The backup application must have permission to manage client permissions. Deleting permissions can only be done using a non-federated command session.

9.1.2.10.1 Code Example – Disabling a Client Permission Records access to a Store

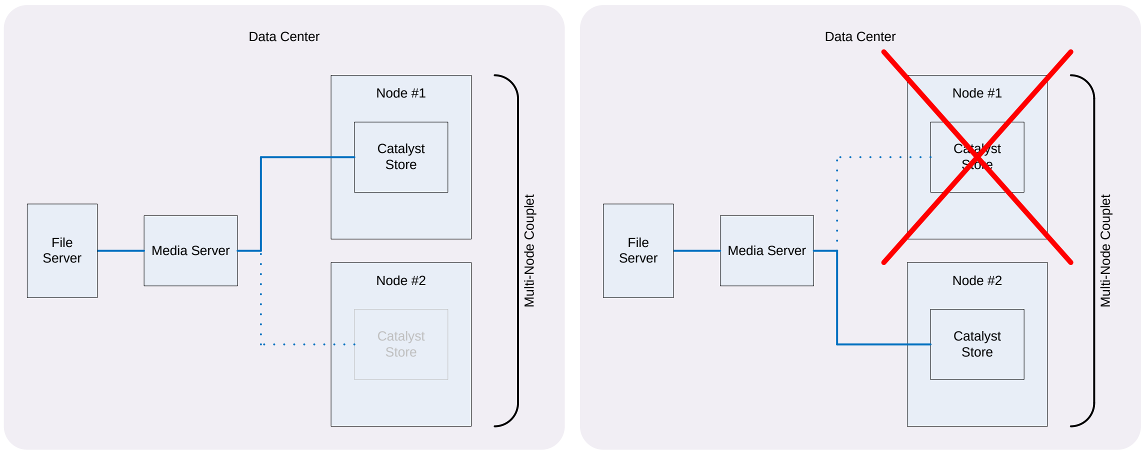
```
int objSDKPrv_RemoveObjectStorePermissions(const char *pServerAddress,
const char *pObjectStoreKey, OSCMN_sStringType clientIdentifier)
{
	int returnStatus = OSCLT_ERR_SUCCESS;
	int callStatus = OSCLT_ERR_SUCCESS;

	OSCLT_sSessionHandleType cmdSessionHandle, *pCmdSessionHandle = 0;
	OSCMN_sKeyType objStoreKey;
	//<<Set ObjectStore key>>
	printf("Remove %s permissions from store %s\n",
		clientIdentifier.String, objStoreKey.String);
	//<<Open command session>>
	callStatus = osCltCmd_RemoveObjectStoreClientPermission(
									pCmdSessionHandle,
									&objStoreKey,
									&clientIdentifier);
	if (callStatus != OSCLT_ERR_SUCCESS)
	{
		printf("Error : Removing %s's permissions failed with"
                   "error : %s.\n", pSeverAddress, .
                    objSDKPrv_ErrorNumToString(callStatus));
             //<<Close command session>>
             returnStatus = callStatus;
             return returnStatus;
       }
       printf("Successfully removed %s from %s\n\n",
              clientIdentifier.String, objStoreKey.String);
       //<<Close command session>>
       return returnStatus;
}
```

Section 10: StoreOnce Autonomic Failover

10.1 Multi-Node StoreOnce Appliance Failover

HPE StoreOnce multi-node appliances have a failover feature which ensures high availability of Catalyst Stores within the cluster. Failover is provided at a node level, with each node being assigned to a couplet. Catalyst Stores hosted on a node are part of a Service Set (collection of services), when failover occurs the Service Set will move from its primary node to the secondary node within the couplet.

From a backup applications point of view, a failover over event will be seen as the Catalyst Stores going offline and then coming back online again a few minutes later. The Catalyst Server address and Catalyst Store name will remain the same and so no reconfiguration is required within the backup application.



As failover may occur during a backup, it is recommended that at minimum backup applications should implement automated retries of failed jobs with retry logic that will wait a period of time before retrying a failed job. For write operations this should include cleaning up any partially written Catalyst Items.

Advanced failover integration can be achieved by creating regular checkpoints during write operations. Following a failover event a backup application would then be able to restart the write operation from the most recent known good checkpoint. See Writing to a Catalyst Item on page 88 for more information on how to reopen Catalyst Items for writing.

Section 11: Compiling with StoreOnce Catalyst

11.1 Catalyst Client Library

11.1.1 Supported Platforms

Table 16 shows the platforms that are supported and compiler versions used to build the Catalyst Client Library.

Table 16 - Supported Platforms and Compilers

Supported Operating System	Build Platform	Compiler	Catalyst Client Library
Windows Server 2008 (x64) Windows Server 2008 R2 (x64) Windows Server 2012 (x64) Windows Server 2012 R2 (x64) Windows Server 2016 (x64)	Windows Server 2008 R2 (x64)	VS2008	libobjstoreclient.lib cofclib.lib libjansson.lib lzo2.lib
SuSE Linux Enterprise Server 11 (x64) SuSE Linux Enterprise Server 12 (x64)	RedHat Enterprise Linux 5 (x64)	GCC 4.1.2	libobjstoreclient.a libjansson.a liblzo2.a libhpcofcp.a
HP-UX 11.23 (IA-64) HP-UX 11.31 (IA-64)	HP-UX (IA64) 11.31		libobjstoreclient.a libjansson.a liblzo2.a libhpcofcp.a
Solaris 10 (x86)	Solaris 10 (x86)	Solaris Studio 12.3 (Sun C 5.12)	libobjstoreclient.a libjansson.a liblzo2.a libhpcofcp.a
Solaris 10 (SPARC-64) Solaris 11 (SPARC-64)	Solaris 10 (SPARC-64)	Solaris Studio 12.3 (Sun C 5.12)	libobjstoreclient.a libjansson.a liblzo2.a libhpcofcp.a
AIX 6.1 (PowerPC) AIX 7.1 (PowerPC)	AIX 6.1 (PowerPC)	GCC 4.8.1	libobjstoreclient.a libjansson.a liblzo2.a libhpcofcp.a

Section 12: StoreOnce Catalyst Server and Client Version Release Notes

12.1 StoreOnce Release

This section will detail the client and server Catalyst features which were added per major StoreOnce release. It details from StoreOnce 3.15 onwards.

12.1.1 StoreOnce 3.15

Introduces v8 Catalyst protocol. The maximum supported protocol version is v8.

12.1.1.1 Multi-reader Functionality

It is now possible to have multiple read concurrent data sessions open to a single Catalyst object. Prior to this release, a request to open a subsequent data session would have resulted in `OSCLT_ERR_OBJECT_LOCK_FAILED`. Multiple sessions can only be opened when only read operations have been performed on an object. Once a clone or write operation is performed the object will be exclusively locked to that single data session.

Multi-reader functionality is provided to support the use case where multiple copies or multiple restores need to be performed from an object.

Applications can query whether a Catalyst store supports multiple readers by querying the `SupportMultipleObjectOpeners` in `OSCMN_sObjectStoreCapabilitiesType` which is returned by a call to `osCltCmd_ListObjectStores`.

This is server functionality and does not require an updated Catalyst client

12.1.1.2 Catalyst over Fibre Channel multiple devices on Linux

Prior to the StoreOnce 3.15 version of the Catalyst client a Linux based client ignored the Catalyst over Fibre Channel setting of “number of devices per login” in the StoreOnce GUI. With this new client version the setting also now applies to Linux, as well as Windows, HP-UX, AIX and Solaris.

12.1.1.3 Quality Fixes / Performance Improvements

A number of small quality improvements have been made, especially in the area of Catalyst over Fibre Channel.

Improves random read performance on the server.
Improves Catalyst clone performance on the server.

12.1.2 StoreOnce 3.16

The maximum supported protocol version is v9.

12.1.2.1 Catalyst Copy over Fibre Channel

Prior to StoreOnce 3.16 Catalyst copy was supported over Ethernet only. An attempt to initiate a Catalyst copy over FC would previously have resulted in an `OSCLT_ERR_INVALID_ADDRESS` error being returned to the Catalyst client. As of 3.16 it is possible to perform Catalyst copies over Fibre Channel between StoreOnce appliances which are running StoreOnce 3.16 and higher, and support CoFC. The primary use cases remain the same as Catalyst Copy over Ethernet - this implementation is also transparent to backup applications, who simply now provide a COFC- address to `osCltCmd_QueueObjectCopyJob` inplace of an IP address / FQDN.

12.1.2.2 Write In Place

StoreOnce 3.16 adds support for Catalyst Copy of Write In Place (WIP) data. This requires both a server and client update.

12.1.2.3Copy Job Destination Credentials

The v9 Catalyst client now supports different credentials being provided to a copy job for the origin and destination servers. `OSCMN_sObjectCopyJobType` now accepts a `DestinationCredentials` field. To determine whether the client are servers support destination credentials the `SupportObjectCopyDestinationCredentials` property of `OSCLT_sClientCapabilitiesType` and `OSCLT_sServerCapabilitiesType` should be merged.

12.1.2.4Catalyst over Fibre Channel

StoreOnce 3.16 increases the number of supported Catalyst over Fibre Channel devices per FC client login to 256.

12.1.2.5Bug / Quality Fixes

A number of small quality improvements have been made, especially in the area of Catalyst over Fibre Channel.

Improves Catalyst clone performance on the server.