TRANSPORT API
JAVA EDITION V3.0

VALUE ADDED COMPONENTS
DEVELOPERS GUIDE

Date of Issue: 30 April 2016
Document Version: 1.0
Document ID: ETAJ300UMVAC.160

THOMSON REUTERS

Table of Contents

Table of Figures

List of Tables

Chapter 1 Introduction

1.1 About this Manual

This document is authored by Transport API architects and programmers who encountered and resolved many of the issues the reader might face. Several of its authors have designed, developed, and maintained the Transport API product and other Thomson Reuters products which leverage it. As such, this document is concise and addresses realistic scenarios and use cases.

1.2 Audience

This manual provides information and examples that aid programmers using the Transport API Value Added Components. The level of material covered assumes that the reader is a user or a member of the programming staff involved in the design, coding, and test phases for applications which will use the Transport API or Transport API Value Added Components. It is assumed that the reader is familiar with the data types, operational characteristics, and user requirements of real-time data delivery networks, and has experience developing products using the Java programming language in a networked environment. Although the Transport API Value Added Components offer alternate entry points to Transport API functionality, it is recommended that users are familiar with general Transport API usage and interfaces.

1.3 Programming Language

Transport API Value Added Components are written to the Java language. All code samples in this document, value added component source, and example applications provided with the product are written in Java.

1.4 References

- *Transport API Java Developers Guide*
- *Transport API Java RDM Usage Guide*

1.5 Documentation Feedback

While we make every effort to ensure the documentation is accurate and up-to-date, if you notice any errors, or would like to see more details on a particular topic, you have the following options:

- Send us your comments via email at apidocumentation@thomsonreuters.com.
- Mark up the PDF using the Comment feature in Adobe Reader. After adding your comments, you can submit the entire PDF to Thomson Reuters by clicking **Send File** in the **File** menu. Use the apidocumentation@thomsonreuters.com address.

1.6 Acronyms and Abbreviations

ACRONYM	DEFINITION
ADH	Advanced Data Hub
ADS	Advanced Distribution Server
API	Application Programming Interface
DMM	Domain Message Model
OMM	Open Message Model
QoS	Quality of Service
RDM	Reuters Domain Model
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format
TREP	Thomson Reuters Enterprise Platform. Includes internal components of TREP, mainly referred to as the ADS and ADH
UML	Unified Modelling Language
Value Added	Transport Java Value Added Components

Table 1: Acronyms and Abbreviations

1.7 Document Conventions

1.7.1 Typographic

- Java classes, methods, and types are shown in `orange, Lucida Console` font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples (one or more lines of code) are show in Lucida Console font against an orange background. For example:

```
/* decode contents into the filter list object */
if ((retVal = filterList.decode(decIter)) >= CodecReturnCodes.SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    FilterEntry filterEntry = CodecFactory.createFilterEntry();
```

1.7.2 Document Structure

Chapters throughout this document typically follow the format:

- General Concepts
- Detailed Concepts
- Interface Definitions
- Example Code

Chapter 2 Product Description

2.1 What is the Transport API?

The Transport API is a low-level transport API that provides the most flexible development environment to the application developer. It is the foundation on which all Thomson Reuters OMM-based components are built. The Transport API allows applications to achieve the highest throughput and lowest latency available with any OMM API, but requires applications to perform all message encoding/decoding and manage all aspects of network connectivity. Transport API, Elektron API, and RFA make up the set of OMM API offerings.

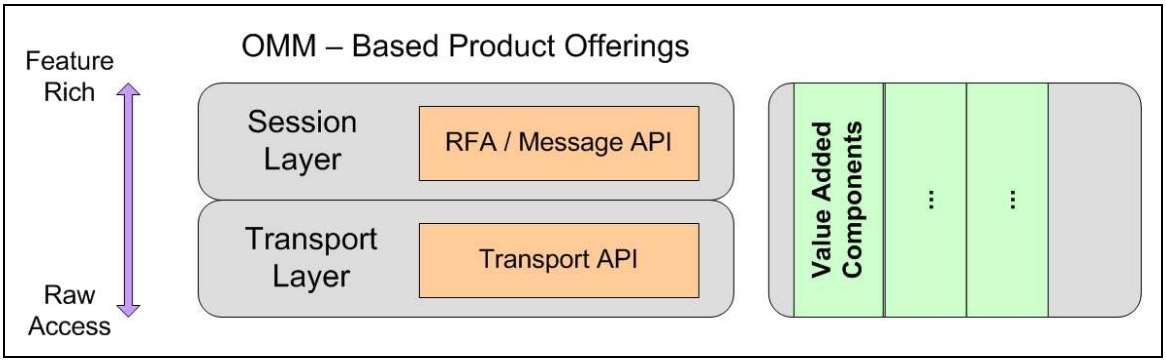

Figure 2: OMM APIs with Value Added Components

The Transport API Value Added Components provide alternate entry points for applications to leverage OMM-Based APIs with more ease and simplicity. These optional components help to offload much of the connection management code and perform encoding and decoding of some key OMM domain representations. Unlike older Domain-based APIs that lock the user into capabilities or ease of use into the highest layer of API, Value Added components are independently implemented for use with the Transport API and RFA in their native languages (Example: Transport API in C and Java, RFA in C++ and Java). These implementations are then shipped with the RFA and Transport API products respectively, as options for the application developer that may want these additional capabilities.

2.2 What are Transport API Value Added Components?

The Value Added Components simplify and compliment the use of the Transport API. These components (depicted in green in the figure below) are offered along side of the Transport API in order to maximize the user experience and allow for more intuitive and straight forward, rapid creation of Transport API applications. Applications can write directly to the Transport API interfaces or commingle some or all Value Added Components. The choice to leverage these components is up to the application developer; Value Added Component use is not required to use the Transport API. By using Transport API Value Added Components, it allows the application to choose and customize the balance between ultra high performance raw access and ease of use feature functionality. Value Added Components are written to the Transport API interfaces and are designed to work alongside the Transport API. Their interfaces have a similar look and feel to Transport API interfaces in order to provide simple migration and consistent use between all components and the Transport API.

All value added components are provided in two forms:

- Fully supported library and header files ready to build into new or existing Transport API applications. Examples and documentation are provided to show the full power and capability of the component.
- Buildable source code[1] to allow for customization and modification to suit specific user needs. This source code serves two purposes:
  - Clients may want to provide their own implementation of the component in a slightly different way than most clients. Rather than starting from scratch, clients can modify the component (as their own code) to jumpstart their development efforts. This code is then theirs to support and maintain.
  - Clients may want to aid in troubleshooting or suggest improvements to the component to benefit everyone.
  - Clients might want to build a new component that has similar behaviours to an existing component. Clients can leverage the code of one component to jump start their development efforts.

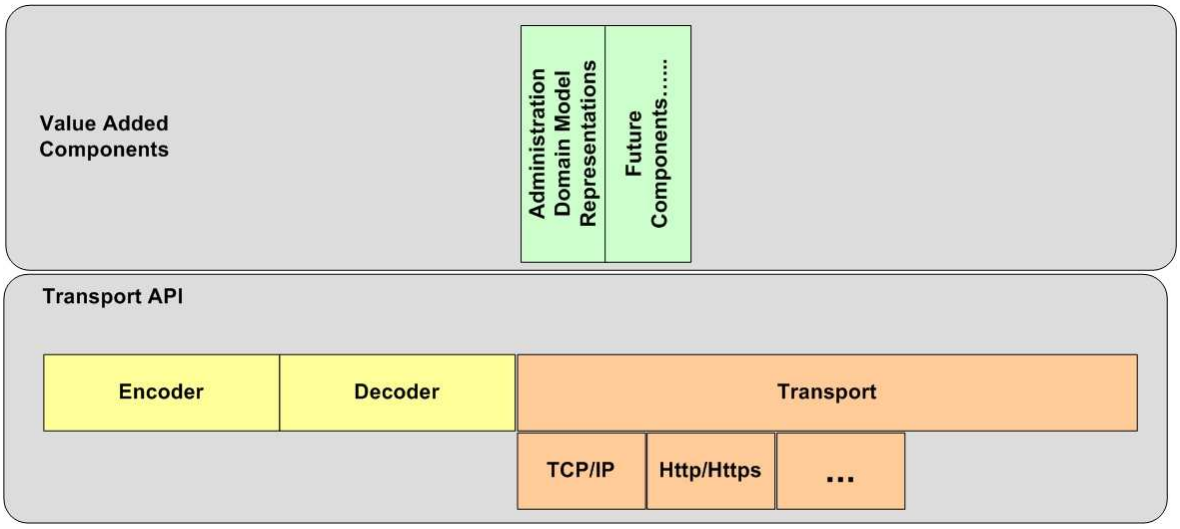

Figure 1: Transport API Value Added Components

[1] Use of the pre-built library and interfaces are fully supported by Thomson Reuters. The provided source code can help with user troubleshooting and debugging. Any modifications to the provided source are supported by the user, not Thomson Reuters.

2.3 Value Added Component Overview

This release of the Transport API includes the Value Added Components described in the following sections. Future Transport API releases may provide additional Value Added Components and domain model representations.

2.3.1 Transport API Reactor

The ***Transport API Reactor*** is a connection management and event processing component that can significantly reduce the amount of code an application must write to leverage OMM in their own applications and to connect to other OMM based devices. The Reactor can be used to create or enhance Consumer, Interactive Provider, and Non-Interactive Provider applications and manage Consumer and Non-Interactive Provider start-up processing, including user log in, source directory establishment, and dictionary download. The Reactor also allows for dispatching of events to user implemented callback methods. In addition, it handles flushing of user written content and manages network pings on the user's behalf. The connection recovery feature allows the Value Added Reactor to automatically recover from disconnects. Value Added domain representations are coupled with the Reactor, allowing domain specific callbacks to be presented with their respective domain representation for easier, more logical access to content. See Chapter 6: Reactor Detailed View for more information. This component depends on the Value Added Administration Domain Model Representation component, the Value Added Utilities, Transport API Java Transport Package, and Transport API Java Codec Package.

2.3.2 Administration Domain Model Representations

The ***Administration Domain Model Representations*** are RDM specific representations of the OMM administrative domain models. This Value Added Component contains classes and interfaces that represent the messages within the Login, Source Directory, and Dictionary domains. All classes follow the formatting and naming specified in the *Transport API Java RDM Usage Guide* document, so access to content is logical and specific to the content being represented. This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's class members to send or receive this content. This not only significantly reduces the amount of code an application needs to interact with OMM devices (i.e., TREP infrastructure), but also ensures that encoding/decoding for these domain models follow OMM specified formatting rules. Applications can use this Value Added Component directly to help with encoding, decoding, and representation of these domain models. When using the Transport API Reactor, this component is embedded to manage and present callbacks with a domain specific representation of content. For more information, refer to Chapter 7. This component depends on the Transport API Java Codec Package.

2.3.3 Value Added Utilities

The Value Added Utilities are a collection of common classes, mainly used by the Transport API Reactor. Included is a selectable bidirectional queue used to communicate events between the Reactor and Worker threads. Other Value Added Utilities include a simple queue along with iterable and concurrent versions of it.

2.3.4 Payload Cache

Applications can leverage Transport API's OMM payload cache feature. Using the payload cache, an application can maintain a local store of the OMM container data it consumes, publishes, or transforms. The cache maintains the latest values of OMM data entries: container values update to reflect the most recent refresh and update message payloads whenever the application receives them. The Transport API retrieves data from the cache entry in the form of an encoded OMM container. The payload cache is independent of other Value Added components, and only requires the Transport API Data Package and Transport API Message Package. Only library and API header files are available for the cache component.

2.4 OMM Consumer Watchlist

The Reactor features a per-channel watchlist that provides a wealth of functionality for OMM Consumer applications. The watchlist automatically performs various recovery behaviors for which developers would normally need to account. The watchlist supports consuming from TCP-based connections (ConnectionTypes.SOCKET).

For details on configuring the Reactor to enable the consumer watchlist, refer to Section 6.3.1.1.

2.4.1 Data Stream Aggregation and Recovery

The watchlist automatically recovers data streams in response to failure conditions, such as disconnects and unavailable services, so that applications do not need special handling for these conditions. As conditions are resolved, the watchlist will re-request items on the application's behalf. Applications can also use this method to request data before a connection is fully established.

To recover from disconnects using a watchlist, enable the Reactor's connection recovery. Options to reconnect disconnected channels are detailed in Section 6.3.2.2.

For efficient bandwidth usage, the watchlist also combines multiple requests for the same item into a single stream and forwards response messages to each requested stream as appropriate.

2.4.2 Additional Features

The watchlist provides additional features for convenience:

- Group and Service Status Fanout: The Reactor maintains a directory stream to receive service updates. As group status messages or service status messages are received, the Reactor forwards the status to all affected streams via StatusMsgs.
- QoS Range Matching: The Reactor will accept and aggregate item requests that specify a range of Qos, or requests that do not specify a Qos. After comparing these requests with the QoS from the providing service, the watchlist uses the best matching QoS.
- Support for Enhanced Symbol List Behaviors: The Reactor supports data streams when requesting a Symbol List item. For details on requesting Symbol list data streams, refer to the *Transport API C Edition RDM Usage Guide*.
- Support for Batch Requests: The Reactor will accept batch requests regardless of whether the connected provider supports them.

2.4.3 Usage Notes

Applications should note the following when enabling the watchlist:

- The application must use the ReactorChannel.submit(Msg) and ReactorChannel.submit(MsgBase) methods to send messages. It cannot use ReactorChannel.submit(TransportBuffer).
- Only one login stream should be opened per ReactorChannel.
- To prevent unnecessary bandwidth use, the watchlist will not recover a dictionary request after a complete refresh is received.
- As private streams are intended for content delivery between two specific points, the watchlist does not aggregate nor recover them.
- The ConsumerRole.dictionaryDownloadMode option is not supported when the watchlist is enabled.

Chapter 3 Building an OMM Consumer

3.1 Overview

This chapter provides an overview of how to create an OMM Consumer application using the Transport API Reactor and Administration Domain Model Representation Value Added Components. The Value Added Components simplify the work done by an OMM Consumer application when establishing a connection to other OMM Interactive Provider applications, including the Enterprise Platform, Data Feed Direct, and Elektron. After the Reactor indicates that the connection is ready, an OMM Consumer can then consume (i.e., send data requests and receive responses) and publish data (i.e., post data).

The general process can be summarized by the following steps.

- Leverage Existing or Create New `Reactor`
- Implement Callbacks and Populate Role
- Establish connection using `Reactor.connect`
- Issue Requests and/or Post information
- Log out and shut down

The Value Add **Consumer** example application, included with the Transport API product, provides one implementation of an OMM Consumer application that uses the Transport API Value Added Components. The application is written with simplicity in mind and demonstrates usage of the Transport API and Transport API Value Added Components. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

3.2 Leverage Existing or Create New Reactor

The `Reactor` can manage one or multiple `ReactorChannel` objects. This allows the application to choose to associate OMM Consumer connections with an existing `Reactor`, having it manage more than one connection, or to create a new `Reactor` to use with the connection.

If the application is creating a new `Reactor`, the `ReactorFactory.createReactor` method is used. This will create any necessary memory and threads that the Reactor uses to manage `ReactorChannel`s and their content flow. If the application is using an existing `Reactor`, there is nothing additional to do.

Detailed information about the `Reactor` and its creation are available in Section 6.2.1.

3.3 Implement Callbacks and Populate Role

Before creating the OMM Consumer connection, the application needs to specify callback methods to use for all inbound content. The callback methods are specified on a per `ReactorChannel` basis so each channel can have its own unique callback methods or existing callback methods can be specified and shared across multiple `ReactorChannel`s.

Several of the callback methods are required for use with a `Reactor`. The application must have an

- `ReactorChannelEventCallback`, which returns information about the `ReactorChannel` and its state (e.g., connection up)
- `DefaultMsgCallback`, which processes all data not handled by other optional callbacks.

In addition to the required callbacks, an OMM Consumer can specify several administrative domain specific callback methods. The available domain specific callbacks are

- `RDMLoginMsgCallback`, which processes all data for the RDM Login domain.
- `RDMDirectoryMsgCallback`, which processes all data for the RDM Source Directory domain.
- `RDMDictionaryMsgCallback`, which processes all data for the RDM Dictionary domain.

The `ConsumerRole` object should be populated with all callback information for the `ReactorChannel`.

The `ConsumerRole` allows the application to provide login, directory and dictionary request information. This can be initialized with default information. The callback methods are specified on the `ConsumerRole` object or with specific information according to the application and user. The `Reactor` will use this information when starting up the `ReactorChannel`.

Detailed information about the `ConsumerRole` is in Section 6.3.1. Information about the various callback methods and their specifications are available in Section 6.4.6.

3.4 Establish Connection using Reactor.connect

Once the `ConsumerRole` is populated, the application can use `Reactor.connect` to create a new outbound connection. `Reactor.connect` will create an OMM Consumer type connection using the provided configuration and role information.

Once the underlying connection is established a channel event will be returned to the application's `ReactorChannelEventCallback`; this will provide the `ReactorChannel` and to indicate the current connection state. At this point, the application can begin using the `ReactorChannel.dispatch` method to dispatch directly on this `ReactorChannel`, or use `Reactor.dispatchAll` to dispatch across all channels associated with the `Reactor`.

The `Reactor` will use the login, directory, and dictionary information specified on the `ConsumerRole` to perform all channel initialization for the user. After the user is logged in, has received a source directory response, and downloaded field dictionaries, a channel event is returned to inform the application that the connection is ready.

The `Reactor.connect` method is described in Section 6.3.2.1. Dispatching is described in Section 6.4.

3.5 Issue Requests and/or Post Information

After the `ReactorChannel` is established, it can be used to request additional content. When issuing the request, the consuming application can use the `serviceId` of the desired service, along with the stream's identifying information. Requests can be sent for any domain using the formats defined in that domain model specification. Domains provided by Thomson Reuters are defined in the *Transport API Java RDM Usage Guide*. This content will be returned to the application via the `DefaultMsgCallback`.

At this point, an OMM Consumer application can also post information to capable provider applications. All content requested, received, or posted is encoded and decoded using the Transport API Codec Package described in the *Transport API Java Developers Guide*.

3.6 Log Out and Shut Down

When the Consumer application is done retrieving or posting content, it can close the `ReactorChannel` by calling `ReactorChannel.close`. This will close all item streams and log out the user. Prior to closing the `ReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `Reactor`, the `Reactor.shutdown` method can be used to shutdown and cleanup any `Reactor` resources.

Closing a `ReactorChannel` is described in Section 6.3.3. Shutting down a `Reactor` is described in Section 6.2.2.

3.7 Additional Consumer Details

The following locations provide specific details about using OMM Consumers, the Transport API, and Transport API Value Added Components:

- The Value Add **Consumer** application demonstrates one way of implementing of an OMM Consumer application that uses the Transport API Value Added Components. The application's source code and Javadoc contain additional information about specific implementation and behaviors.
- Chapter 6 provides a detailed look at the Transport API Reactor.
- Chapter 7 provides more information about the Administration Domain Model Representations.
- The *Transport API Java Developers Guide* provides specific Transport API encoder/decoder and transport usage information.
- The *Transport API Java RDM Usage Guide* provides specific information about the DMMs used by this application type.

Chapter 4 Building an OMM Interactive Provider

4.1 Overview

This chapter provides a high-level description of how to create an OMM Interactive Provider application using the Transport API Reactor and Administration Domain Model Representation Value Added Components. An OMM Interactive Provider application opens a listening socket on a well-known port allowing OMM Consumer applications to connect. The Transport API Value Added Components simplify the work done by an OMM Interactive Provider application when accepting connections and handling requests from OMM Consumers.

The following steps summarize this process:

- Leverage Existing or Create New `Reactor`
- Create a `Server`
- Implement Callbacks and Populate Role
- Associate incoming connections using `Reactor.accept`
- Perform Login Process
- Provide Source Directory Information
- Provide Necessary Dictionaries
- Handle Requests and Post messages
- Disconnect Consumers and shut down

Included with the Transport API package, the Value Add **Provider** example application provides one way of implementing an OMM Interactive Provider application that uses the Transport API Value Added Components. The application is written with simplicity in mind and demonstrates the use of the Transport API and Transport API Value Added Components. Portions of the functionality are abstracted for easy reuse, though you might need to customize it to achieve your own unique performance and functionality goals.

4.2 Leverage Existing or Create New Reactor

The `Reactor` can manage one or multiple `ReactorChannel` structures. This allows the application to choose to associate OMM Provider connections with an existing `Reactor`, having it manage more than one connection, or to create a new `Reactor` to use with the connection.

If the application is creating a new `Reactor`, the `ReactorFactory.createReactor` method is used. This will create any necessary memory and threads that the Reactor uses to manage `ReactorChannel`s and their content flow. If the application is using an existing `Reactor`, there is nothing additional to do.

Detailed information about the `Reactor` and its creation are available in Section 6.2.1.

4.3 Create a Server

The first step of any Transport API Interactive Provider application is to establish a listening socket, usually on a well-known port so that consumer applications can easily connect. The provider uses the `Transport.bind` method to open the port and listen for incoming connection attempts. This uses the standard Transport API Transport functionality described in the *Transport API Java Developers Guide.*

Whenever an OMM consumer application attempts to connect, the provider will use the `Server` and associate the incoming connections with a `Reactor`, which will accept the connection and perform any initialization. This process is described in the following sections.

4.4 Implement Callbacks and Populate Role

Before accepting an incoming connection with the OMM Provider, the application needs to specify callback methods to use for all inbound content. The callback methods are specified on a per `ReactorChannel` basis so each channel can have its own unique callback methods or existing callback methods can be specified and shared across multiple `ReactorChannel`s.

Several of the callback methods are required for use with a `Reactor`. The application must have a

- `ReactorChannelEventCallback`, which returns information about the `ReactorChannel` and its state (e.g., connection up)
- `DefaultMsgCallback`, which processes all data not handled by other optional callbacks.

In addition to the required callbacks, an OMM Provider can specify several administrative domain specific callback methods. The available domain specific callbacks are

- `RDMLoginMsgCallback`, which processes all data for the RDM Login domain.
- `RDMDirectoryMsgCallback`, which processes all data for the RDM Source Directory domain.
- `RDMDictionaryMsgCallback`, which processes all data for the RDM Dictionary domain.

The `ProviderRole` object should be populated with all callback information for the `ReactorChannel`.

Detailed information about the `ProviderRole` is in Section 6.3.1. Information about the various callback methods and their specifications are available in Section 6.4.6.

4.5 Associate Incoming Connections Using Reactor.accept

Once the `ProviderRole` is populated, the application can use `Reactor.accept` to accept a new inbound connection. `Reactor.accept` will accept an OMM Provider connection from the passed in `Server` using the provided configuration and role information.

Once the underlying connection is established a channel event will be returned to the application's `ReactorChannelEventCallback`; this will provide the `ReactorChannel` and to indicate the current connection state. At this point, the application can begin using the `ReactorChannel.dispatch` method to dispatch directly on this `ReactorChannel`, or use `Reactor.dispatchAll` to dispatch across all channels associated with the `Reactor`.

The `Reactor` will perform all channel initialization and pass any administrative domain information to the application via the callbacks specified with the `ProviderRole`.

The `Reactor.accept` method is described in Section 6.3.2.6. Dispatching is described in Section 6.4.

4.6 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM Interactive Provider must handle the consumer's Login request messages and supply appropriate responses. Login information will be provided to the application via the `RDMLoginMsgCallback`, when specified on the `ProviderRole`.

After receiving a Login request, the Interactive Provider can perform any necessary authentication and permissioning.

- If the Interactive Provider grants access, it should send a `LoginRefresh` to convey that the user successfully connected. This message should indicate the feature set supported by the provider application.
- If the Interactive Provider denies access, it should send a `LoginStatus`, closing the connection and informing the user of the reason for denial.

Login messages can be encoded and decoded using the messages' `encode` and `decode` methods. More details and code examples are in Section 7.3.

All content requested, received, or posted is encoded and decoded using the Transport API Java Codec Package described in the *Transport API Java Developers Guide*.

Information about the Login domain and expected content formatting is available in the *Transport API Java RDM Usage Guide*.

4.7 Provide Source Directory Information

The Source Directory domain model conveys information about all available services in the system. An OMM consumer typically requests a Source Directory to retrieve information about available services and their capabilities. This includes information about supported domain types, the service's state, the QoS, and any item group information associated with the service. Thomson Reuters recommends that at a minimum, an Interactive Provider supply the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains the name and `serviceId` for each available service. The Interactive Provider should populate the filter with information specific to the services it provides.
- The Source Directory State filter contains status information for the service informing the consumer whether the service is Up (available), or Down (unavailable).
- The Source Directory Group filter conveys item group status information, including information about group states, as well as the merging of groups. If a provider determines that a group of items is no longer available, it can convey this information by sending either individual item status messages (for each affected stream) or a Directory message containing the item group status information. Additional information about item groups is available in the *Transport API Java Developers Guide*.

Source Directory messages can be encoded and decoded using the messages' `encode` and `decode` methods. More details and code examples are in Section 7.4.

All content requested, received, or posted is encoded and decoded using the Transport API Java Codec Package described in the *Transport API Java Developers Guide*.

Information about the Source Directory domain and expected content formatting is available in the *Transport API Java RDM Usage Guide*.

4.8 Provide or Download Necessary Dictionaries

Some data requires the use of a dictionary for encoding or decoding. The dictionary typically defines type and formatting information, and tells the application how to encode or decode information. Content that uses the `FieldList` type requires the use of a field dictionary (usually the Thomson Reuters **RDMFieldDictionary**, though it can instead be a user-defined or modified field dictionary).

The Source Directory message should notify the consumer about dictionaries needed to decode content sent by the provider. If the consumer needs a dictionary to decode content, it is ideal that the Interactive Provider application also make this dictionary available to consumers for download. The provider can inform the consumer whether the dictionary is available via the Source Directory.

If consuming from an ADH and then providing the content downstream, a provider application can also download the RWFFld and RWFEnum dictionaries. By downloading these dictionaries, the Transport API can retrieve appropriate dictionary information for providing field list contents. A provider can use this feature to verify whether it is using the appropriate version of the dictionary or to encode data. An ADH that supports provider dictionary downloads sends a Login request message containing the **SupportProviderDictionaryDownload** login element. The Transport API sends the dictionary request using the Dictionary domain model. For details on using the Login domain and expected message content, refer to the *Transport API Java RDM Usage Guide*

Dictionary messages can be encoded and decoded using the messages' `encode` and `decode` methods. More details and code examples are in Section 7.5. Dictionary requests will be provided via the `RDMDictionaryMsgCallback`, when specified on the `ProviderRole`.

Whether the Transport API loads a dictionary from file or requests it from an ADH, the Transport API offers several utility functions for loading, downloading, and managing a properly-formatted field dictionary. The Transport API also has utility functions to help the provider encode into an appropriate format for downloading and decoding dictionaries.

- For details on how the Transport API encodes and decodes content using the its Codec Package, refer to the *Transport API Java Developers Guide*.
- For further Information about the Dictionary domain, dictionary utility functions, and expected content formatting is available in the *Transport API Java RDM Usage Guide*.

4.9 Handle Requests and Post Messages

A provider can receive a request for any domain, though this should typically be limited to the domain capabilities indicated in the Source Directory. When a request is received, the provider application must determine if it can satisfy the request by:

- Comparing `msgKey` identification information
- Determining whether it can provide the requested QoS
- Ensuring that the consumer does not already have a stream open for the requested information

If a provider can service a request, it should send appropriate responses. However, if the provider cannot satisfy the request, the provider should send a `StatusMsg` to indicate the reason and close the stream. All requests and responses should follow specific formatting as defined in the domain model specification. The *Transport API Java RDM Usage Guide* defines all domains provided by Thomson Reuters. This content will be returned to the application via the `DefaultMsgCallback`.

The provider can specify that it supports post messages via the `LoginRefresh`. If a provider application receives a Post message, the provider should determine the correct handling for the post. This depends on the application's role in the system and might involve storing the post in its cache or passing it farther up into the system. If the provider is the destination for the Post, the provider should send any requested acknowledgments, following the guidelines described in the *Transport API Java Developers Guide*. Any posted content will be returned to the application via the `DefaultMsgCallback`.

All content requested, received, or posted is encoded and decoded using the Transport API Java Codec Package described in the *Transport API Java Developers Guide*.

4.10 Disconnect Consumers and Shut Down

If the `Reactor` application must shut down, it can either leave consumer connections intact or shut them down. If the provider decides to close consumer connections, the provider should send a `StatusMsg` on each connection's Login stream closing the stream. At this point, the consumer should assume that its other open streams are also closed.

It can then close the `ReactorChannel`s by calling `ReactorChannel.close`. Prior to closing the `ReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `Reactor`, the `Reactor.shutdown` method can be used to shutdown and cleanup any `Reactor` resources.

Closing a `ReactorChannel` is described in Section 6.3.3. Shutting down a `Reactor` is described in Section 6.2.2

4.11 Additional Interactive Provider Details

For specific details about OMM Interactive Providers, Transport API, and Transport API Value Added Component use, refer to the following locations:

- The Value Add **Provider** application demonstrates one implementation of an OMM Interactive Provider application that uses Transport API Value Added Components. The application's source code and Javadoc have additional information about specific implementation and behaviors.
- Chapter 6 provides a detailed look at the Transport API Reactor.
- Chapter 7 provides more information about the Administration Domain Model Representations.
- The *Transport API Java Developers Guide* provides specific Transport API encoder/decoder and transport usage information.
- The *Transport API Java RDM Usage Guide* provides specific information about the DMMs used by this application type.

Chapter 5 Building an OMM Non-Interactive Provider using Value Added Components

5.1 Overview

This chapter provides an overview of how to create an OMM non-interactive provider application using the Transport API Reactor and Administration Domain Model Representation Value Added Components. The Value Added Components simplify the work done by an OMM Non-Interactive Provider application when establishing a connection to ADH devices. After the Reactor indicates that the connection is ready, an OMM Non-Interactive Provider can publish information into the ADH cache without needing to handle requests for the information. The ADH and other Enterprise Platform components can cache the information , provide the information to any OMM Consumer applications that indicate interest.

The general process can be summarized by the following steps.

- Leverage Existing or Create New `Reactor`
- Implement Callbacks and Populate Role
- Establish connection using `Reactor.connect`
- Provide Dictionary Download
- Provide content
- Log out and shut down

The value-add **NIProvider** example application, included with the Transport API product, provides one implementation of an OMM Consumer application that uses the Transport API Value Added Components. The application is written with simplicity in mind and demonstrates usage of Transport API and Transport API Value Added Components. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

5.2 Leverage Existing or Create New Reactor

The `Reactor` can manage one or multiple `ReactorChannel` objects. This allows the application to choose to associate OMM Non-Interactive Provider connections with an existing `Reactor`, having it manage more than one connection, or to create a new `Reactor` to use with the connection.

If the application is creating a new `Reactor`, the `ReactorFactory.createReactor` method is used. This will create any necessary memory and threads that the Reactor uses to manage `ReactorChannel`s and their content flow. If the application is using an existing `Reactor`, there is nothing additional to do.

Detailed information about the `Reactor` and its creation are available in Section 6.2.1.

5.3 Implement Callbacks and Populate Role

Before creating the OMM Non-Interactive Provider connection, the application needs to specify callback methods to use for all inbound content. The callback methods are specified on a per `ReactorChannel` basis so each channel can have its own unique callback methods or existing callback methods can be specified and shared across multiple `ReactorChannel`s.

Several of the callback methods are required for use with a `Reactor`. The application must have a:

- `ReactorChannelEventCallback`, which returns information about the `ReactorChannel` and its state (e.g., connection up)
- `DefaultMsgCallback`, which processes all data not handled by other optional callbacks.

In addition to the required callbacks, an OMM Non-Interactive Provider can specify administrative domain specific callback methods. The available domain specific callback is `RDMLoginMsgCallback`, which processes all data for the RDM Login domain.

The `NIProviderRole` object should be populated with all callback information for the `ReactorChannel`.

The `NIProviderRole` allows the application to provide login request and initial directory refresh information. This can be initialized with default information. The callback methods are specified on the `NIProviderRole` object or with specific information according to the application and user. The `Reactor` will use this information when starting up the `ReactorChannel`.

Detailed information about the `NIProviderRole` is in Section 6.3.1. Information about the various callback methods and their specifications are available in Section 6.4.6.

5.4 Establish Connection using Reactor.connect

Once the `NIProviderRole` is populated, the application can use `Reactor.connect` to create a new outbound connection. `Reactor.connect` will create an OMM Non-Interactive Provider type connection using the provided configuration and role information.

Once the underlying connection is established a channel event will be returned to the application's `ReactorChannelEventCallback`; this will provide the `ReactorChannel` and to indicate the current connection state. At this point, the application can begin using the `ReactorChannel.dispatch` method to dispatch directly on this `ReactorChannel`, or use `Reactor.dispatchAll` to dispatch across all channels associated with the `Reactor`.

The `Reactor` will use the login and directory information specified on the `NIProviderRole` to perform all channel initialization for the user. After the user is logged in and has sent a source directory response, a channel event is returned to inform the application that the connection is ready.

The `Reactor.connect` method is described in Section 6.3.2.1. Dispatching is described in Section 6.4.

5.5 Download the Dictionary

If connected to a supporting ADH, an OMM NIP can download the RWFFld and RWFEnum dictionaries to retrieve the appropriate dictionary information for providing field list content. An OMM NIP can use this feature to ensure they use the appropriate version of the dictionary or to encode data. To support the Provider Dictionary Download feature, the ADH sends a Login response message containing the `SupportProviderDictionaryDownload` login element. The dictionary request is sent using the Dictionary domain model.

The Transport API offers several utility functions for downloading and managing a properly-formatted field dictionary. There are also utility functions that the provider can use to encode the dictionary into an appropriate format for downloading or decoding.

For details on using the Login domain, expected message content, and available dictionary utility functions, refer to the *Transport API Java Edition RDM Usage Guide*.

5.6 Provide Content

After the `ReactorChannel` is established, it can begin pushing content to the ADH. Each unique information stream should begin with a `RefreshMsg`, conveying all necessary identification information for the content. Because the provider instantiates this information, a negative value `streamId` should be used for all streams. The initial identifying refresh can be followed by other status or update messages.

All content is encoded and decoded using the Transport API Java Codec Package described in the *Transport API Java Developers Guide*.

5.7 Log Out and Shut Down

When the Consumer application is done retrieving or posting content, it can close the `ReactorChannel` by calling `ReactorChannel.close`. This will close all item streams and log out the user. Prior to closing the `ReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `Reactor`, the `Reactor.shutdown` method can be used to shutdown and cleanup any `Reactor` resources.

Closing a `ReactorChannel` is described in Section 6.3.3. Shutting down a `Reactor` is described in Section 6.2.2.

5.8 Additional Non-Interactive Provider Details

The following locations discuss specific details about using OMM Non-Interactive Providers and Transport API:

- The Value Add **NIProvider** application demonstrates one implementation of an OMM Non-Interactive Provider application that uses the Transport API Value Added Components. The application's source code and Javadoc have additional information about the specific implementation and behaviors.
- Chapter 6 provides a detailed look at the Transport API Reactor.
- Chapter 7 provides more information about the Administration Domain Model Representations.
- The *Transport API Java Developers Guide* provides specific Transport API encoder/decoder and transport usage information.
- The *Transport API Java RDM Usage Guide* provides specific information about the DMMs used by this application type.

Chapter 6 Reactor Detailed View

6.1 Concepts

The ***Transport API Reactor*** is a connection management and event processing component that can significantly reduce the amount of code an application must write to leverage OMM. This component helps simplify many aspects of a typical Transport API application, regardless of whether the application is an OMM Consumer, OMM Interactive Provider, or OMM Non-Interactive Provider. The Transport API Reactor can help manage Consumer and Non-Interactive Provider start-up processing, including user log in, source directory establishment, and dictionary download. It also allows for dispatching of events to user implemented callback methods, handles flushing of user written content, and manages network pings on the user's behalf. The connection recovery feature allows the Value Added Reactor to automatically recover from disconnects. Value Added domain representations are coupled with the Reactor, allowing domain specific callbacks to be presented with their respective domain representation for easier, more logical access to content.

FUNCTIONALITY	TRANSPORT API	TRANSPORT API REACTOR
Programmatic Configuration	X	X
Programmatic Logging	X	X
Controlled Fragmentation and Assembly of Large Messages	X	X
Controlled Locking / Threading Model	X	X
Controlled Message Buffers with Ability to Change During Runtime	X	X
Controlled Message Packing	X	X
Support for Unified and Segmented Network Connection Types	X	X
Network Ping Management	***	X
Automatic Flushing of Data	***	X
User-Defined Callbacks for Data	***	X
User Login	***	X
Requesting Source Directory	***	X
Downloading Field Dictionary	***	X
Loading Field Dictionary File	***	X
***: Transport API users can implement this functionality themselves. They can also use or modify the Transport API Reactor functionality.		

Figure 2: Transport API and Transport API Reactor Comparison

The Transport API Reactor internally depends on the Administration Domain Model Representation component. This allows the user to provide and consume the administrative RDM types in a more logical format. This additionally hides encoding and decoding of these domains from the Reactor user, all interaction is via a simple structural representation. More information about the Administration Domain Model Representation value added component is available in Chapter 7. The Transport API Reactor also leverages several utility components, contained in the Value Added Utilities. This includes an iterable queue and a selectable bidirectional queue used to communicate events between the Reactor and Worker threads.

The Transport API Reactor helps to manage the life-cycle of a connection on the user's behalf. When a channel is associated with a Reactor, the Reactor will perform all necessary transport level initialization and alert the user, via a callback, when the connection is up, ready for use, and down. An application can simultaneously run multiple unique reactor instances, where each Reactor instance can associate and manage a single channel or multiple channels. This functionality allows users to quickly and easily horizontally scale their application to leverage multi-core systems or distribute content across multiple connections.

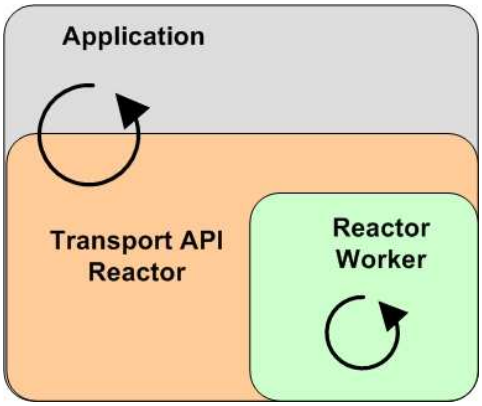

Figure 3: Transport API Reactor Thread Model

Each instance of the Transport API Reactor leverages multiple threads to help manage inbound and outbound data efficiently. Figure 3 provides a high level view of the Reactor threading model. There are two main threads associated with each Transport API Reactor instance. The application thread is the main driver of the Reactor; all event dispatching (e.g., reading), callback processing, and submitting of data to the Transport API is done from this thread. This is done to reduce latency and simplify any threading model associated with the user defined callback methods – because callbacks happen from the application thread, a single threaded application does not need to have additional mutex locking. The Transport API Reactor also leverages an internal worker thread. The worker thread flushes any queued outbound data and manages outbound network pings for all channels associated with the Reactor. It also attempts to recover any connections that are lost.

The application drives the Reactor with the use of a dispatch method. The dispatch method reads content from the network, performs some light processing to handle inbound network pings, and provides the information to the user through a series of per-channel, user defined callback methods. Callback methods are separated based on whether they are Reactor callbacks or channel callbacks. Channel callbacks are separated by domain, with a default callback where all unhandled domains or non-OMM content are provided to the user. The application can choose whether to dispatch on a single channel or across all channels being managed by the Reactor. The application can leverage an I/O notification mechanism (e.g. select, poll) or periodically call dispatch – it is all up to the user.

6.1.1 Reactor Error Handling

The `ReactorErrorInfo` object is used to return error or warning information to the application. This can be returned from the various Reactor methods as well as part of a callback method. When returned directly from a Reactor method, this indicates that an error occurred while processing in that method. If returned as part of a callback method, this indicates that an error has occurred on one of the channels that the Reactor is managing.

The `ReactorErrorInfo` members are described in the following table.

CLASS MEMBER	DESCRIPTION
code	An informational code about this error. Indicates whether it reports a failure condition or is intended to provide non-failure related information to the user. Refer to Table 3 for details on the available codes.
error	The underlying error information from the Transport API. This includes a pointer to the `Channel` that the error occurred on, both a Transport API and a system error number, and more descriptive error text. The `Error` and its values are described in the *Transport API Java Developers Guide*.
location	Provides information about the file and line that the error occurred at. Detailed error text is provided via the `Error` portion of this object.

Table 2: `ReactorErrorInfo` Class Members

6.1.2 Reactor Error Info Codes

It is important that the application monitors return values from the `Reactor` callbacks and methods. The error info codes indicate whether the returned `ReactorErrorInfo` indicates a failure condition or is providing information of a successful operation.

RETURN CODE	DESCRIPTION
SUCCESS	Indicates a success code. Used to inform the user of the success and provide additional information.
FAILURE	A general failure has occurred. The `ReactorErrorInfo` code contains more information about the specific error.
WRITE_CALL_AGAIN	This is a transport success code. `ReactorChannel.submit` is fragmenting the buffer and needs to be called again with the same buffer. This indicates that Write was unable to send all fragments with the current call and must continue fragmenting. Information.
NO_BUFFERS	There are no buffers available from the buffer pool. Returned from ReactorChannel.submit. Use ReactorChannel.ioctl to increase pool size or wait for the Reactor's Worker thread to flush data and return buffers to pool. Use ReactorChannel.bufferUsage to monitor for free buffers.
PARAMETER_OUT_OF_RANGE	Indicates that a parameter was out of range.
PARAMETER_INVALID	Indicates that a parameter was invalid.

Table 3: Reactor Error Info Codes

6.1.3 Transport API Reactor Application Lifecycle

The following figure depicts the typical lifecycle of an application using the Transport API Reactor, as well as the associated method calls. The subsequent sections in this document provide more detailed information.

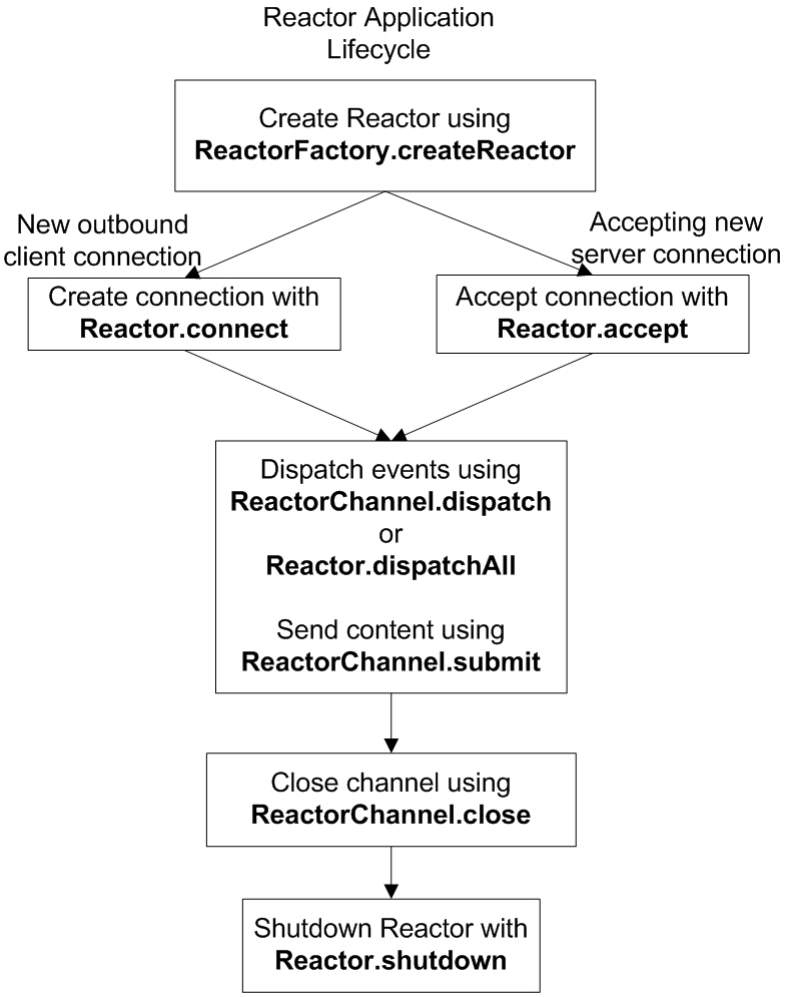

Figure 4: Transport API Reactor Application Lifecycle

6.2 Reactor Use

This section describes use of the Reactor, or `Reactor`. The `Reactor` manages `ReactorChannel`s which are described in Section 6.3. An understanding of both constructs is necessary for application writers.

The `Reactor` is represented by a class as defined in Table 4.

CLASS MEMBER	DESCRIPTION
reactorChannel	The Reactor's internal channel that's used for `Reactor` specific events to communicate with the worker thread. The application must register this `ReactorChannel`'s selectable channel with its selector if using select notification. All `ReactorChannel` data event notification occurs on the `ReactorChannel`'s specific `selectableChannel`, as detailed in Section 6.3.
userSpecObj	An object that can be set by the user of the `Reactor`. This value can be set via the `ReactorOptions` during creation of the `Reactor`. This information can be useful for identifying a specific instance of a `Reactor` or coupling this `Reactor` with other user created information.

Table 4: `Reactor` Class Members

Note: An application can leverage multiple `Reactor` instances to scale across multiple cores and distribute their `ReactorChannel`s as needed.

6.2.1 Reactor Creation

The lifecycle of a `Reactor` is controlled by the application, which controls creation and destruction of each reactor instance. The following sections describe the creation functionality in more detail.

6.2.1.1 Reactor Creation

The creation of a `Reactor` instance can be accomplished through the use of the following method.

METHOD NAME	DESCRIPTION
ReactorFactory.createReactor	Creates a `Reactor` instance, including all necessary internal memory and threads. Once the `Reactor` is created, `ReactorChannel`s can be associated, as described in Section 6.3. Options are passed in via the `ReactorOptions`, as defined in Table 6.

Table 5: `Reactor` Creation Method

6.2.1.2 ReactorOptions Class Members

CLASS MEMBER	DESCRIPTION
userSpecObj	An object that can be set by the application. This value is preserved and stored in the `userSpecObj` of the `Reactor` returned from `ReactorFactory.createReactor`. This information can be useful for identifying a specific instance of a Reactor or coupling this Reactor with other user created information.
enableXmlTracing	Enables XML tracing for the `Reactor`. The `Reactor` prints the XML representation of all OMM message when enabled.

Table 6: `ReactorOptions` Class Members

6.2.1.3 ReactorOptions Utility Method

The Transport API provides the following utility method for use with the `ReactorOptions`.

METHOD NAME	DESCRIPTION
clear	Clears the `ReactorOptions` object. Useful for object reuse.

Table 7: `ReactorOptions` Utility Method

6.2.2 Reactor Destruction

The lifecycle of a `Reactor` is controlled by the application, which controls creation and destruction of each reactor instance. The following sections describe the destruction functionality in more detail.

6.2.2.1 Reactor Destruction

When the application no longer requires a `Reactor` instance, it can destroy it using the following method.

METHOD NAME	DESCRIPTION
shutdown	Shuts down and cleans up a `Reactor`. This also sends `ReactorChannelEvent`s, indicating channel down, to all `ReactorChannel`s associated with this `Reactor`.

Table 8: `Reactor` Destruction Method

6.2.2.2 Reactor Creation and Destruction Example

```
ReactorOptions reactorCreateOptions = ReactorFactory.createReactorOptions();

reactorCreateOptions.clear();

/* Create the Reactor. */
reactor = ReactorFactory.createReactor(reactorCreateOptions, errorInfo);


/* Any use of the reactor occurs here -- see following sections for all other functionality */


/* Destroy the Reactor. */
reactor.shutdown(pReactor, &rsslErrorInfo);
```

Code Example 1: Reactor Creation and Destruction Example

6.3 Reactor Channel Use

The `ReactorChannel` object is used to represent a connection that can send or receive information across a network. This object is used to represent a connection, regardless of if that was an outbound connection or a connection that was accepted by a listening socket via a `Server`. The `ReactorChannel` is the application's point of access, used to perform any action on the connection that it represents (e.g. dispatching events, writing, disconnecting, etc). See the subsequent sections for more information about `ReactorChannel` and how to associate with a `Reactor`.

Note: Only Transport API Reactor methods, like those defined in this chapter, should be called on a channel managed by a Reactor.

The following table describes the members of the `ReactorChannel` class.

CLASS MEMBER	DESCRIPTION
channel	The underlying `Channel` object, as defined in the *Transport API Java Developers Guide*, mainly for reference purposes. All operations should be performed using the Transport API Reactor functionality; the application should not use this `Channel` directly with any Transport functionality.
server	The underlying `Server` object, as defined in the *Transport API Java Developers Guide*, mainly for reference purposes. This will only be populated when the channel was created via the `Reactor.accept` method, as described in Section 6.3.2.6.
selectableChannel	Represents a selectable channel that can be used in select notification to alert users when dispatch is required on a specific `ReactorChannel`. Used to register with a selector.
oldSelectableChannel	It is possible for a selectable channel to change over time, typically due to some kind of connection keep-alive mechanism. If this occurs, this is typically communicated via a callback indicating **ReactorChannelEventTypes.FD_CHANGE**. The previous selectable channel is stored in `oldSelectableChannel` so the application can properly unregister and then register the new `selectableChannel` with their selector.
state	The state of the ReactorChannel.
protocolType	When a `ReactorChannel` is up (**ReactorChannelEventTypes.CHANNEL_UP**), this is populated with the `protocolType` associated with the content being sent on this connection. If the `protocolType` indicated by a server does not match the `protocolType` that a client specifies, the connection will be rejected. The Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information on versioning, refer to the *Transport API Java Developers Guide*.
majorVersion	When a `ReactorChannel` is up (**ReactorChannelEventTypes.CHANNEL_UP**), this is populated with the major version number that is associated with the content being sent on this connection. Typically, a minor version increase is associated with a fully backward compatible change or extension. The Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information on versioning, refer to the *Transport API Java Developers Guide* .

CLASS MEMBER	DESCRIPTION
minorVersion	When a `ReactorChannel` is up (**ReactorChannelEventTypes.CHANNEL_UP**), this is populated with the minor version number that is associated with the content being sent on this connection. Typically, a minor version increase is associated with a fully backward compatible change or extension. The Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information on versioning, refer to the *Transport API Java Developers Guide*.
userSpecObj	An object that can be set by the user of the `ReactorChannel`. This value can be set via the `ReactorConnectOption`s and `ReactorAcceptOption`s. This information can be useful for coupling this `ReactorChannel` with other user created information.

Table 9: `ReactorChannel` Class Members

6.3.1 Reactor Channel Roles

A `ReactorChannel` can be configured to fulfil several specific roles, which overlap with the typical OMM application types. The provided role definitions are:

- `ConsumerRole` for OMM Consumer applications
- `ProviderRole` for OMM Interactive Provider applications
- `NIProviderRole` for OMM Non-Interactive Provider applications

All roles have the same base class, the `ReactorRole`. This contains information and callback methods that are common to all role types.

CLASS MEMBER	DESCRIPTION
type	The role type enumeration value, as defined in Table 11.
channelEventCallback	This `ReactorChannel`'s user-defined callback method to handle all `ReactorChannel` specific events, like **ReactorChannelEventTypes.CHANNEL_UP** or **ReactorChannelEventTypes.CHANNEL_DOWN**. This callback method is required for all role types. This callback is defined in more detail in Section 6.4.6.
defaultMsgCallback	This `ReactorChannel`'s user-defined callback method to handle `Msg` content not handled by another domain-specific callback method. This callback method is required for all role types. This callback is defined in more detail in Section 6.4.6.

Table 10: `ReactorRole` Class Members

ENUMERATED NAME	DESCRIPTION
CONSUMER	Indicates that the `ReactorChannel` should act as an OMM Consumer.
PROVIDER	Indicates that the `ReactorChannel` should act as an OMM Interactive Provider.
NIPROVIDER	Indicates that the `ReactorChannel` should act as an OMM Non-Interactive Provider.

Table 11: `ReactorRoleTypes` Enumerated Values

6.3.1.1 Reactor Channel OMM Consumer Role

When a ReactorChannel is acting as an OMM Consumer application, it connects to an OMM Interactive Provider. As part of this process it is expected to perform a login to the system. Once the login is completed, the consumer acquires a source directory, which provides information about the available services and their capabilities. Additionally, a consumer can download or load field dictionaries, providing information to help decode some types of content. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the *Transport API Java RDM Usage Guide*.

A ReactorChannel in a consumer role helps to simplify this connection process by exchanging these messages on the user's behalf. The user can choose to provide specific information or leverage a default populated messages, which uses the information of the user currently logged into the machine running the application. In addition, the Transport API Reactor allows the application to specify user-defined callback methods to handle the processing of received messages on a per-domain basis.

6.3.1.1.1 OMM Consumer Role

When creating a ReactorChannel, this information can be specified with the ConsumerRole object, as defined below.

CLASS MEMBER	DESCRIPTION
rdmLoginRequest	The LoginRequest, defined in Section 6.4.6, to be sent during the connection establishment process. This can be populated with a user's specific information or invoke the initDefaultRDMLoginRequest method to populate with default information. If this parameter is left empty no login will be sent to the system; useful for systems that do not require a login.
rdmDirectoryRequest	The DirectoryRequest, defined in Section 6.4.6, to be sent during the connection establishment process. This can be populated with specific source directory request information or invoke the initDefaultRDMDirectoryRequest method to populate with default information. If this parameter is specified, an rdmLoginRequest is required. If this parameter is left empty, no directory request will be sent to the system.
dictionaryDownloadMode	Informs the ReactorChannel on the method to use when requesting dictionaries. The allowable modes are defined in Table 13.
loginMsgCallback	This ReactorChannel's user-defined callback method to handle login message content. If not specified, all received login messages will be passed to the defaultMsgCallback. • This callback is defined in more detail in Section 6.4.6. • The login messages are described in Section 7.3.
directoryMsgCallback	This ReactorChannel's user-defined callback method to handle directory message content. If not specified, all received directory messages will be passed to the defaultMsgCallback. • This callback is defined in more detail in Section 6.4.6. • The directory messages are described in Section 7.4.
dictionaryMsgCallback	This ReactorChannel's user-defined callback method to handle dictionary message content. If not specified, all received dictionary messages will be passed to the defaultMsgCallback. • This callback is defined in more detail in Section 6.4.6. • The dictionary messages are described in Section 7.5.

Table 12: ConsumerRole Class Members

6.3.1.1.2 OMM Consumer Role Dictionary Download Modes

There are several dictionary download options available to a `ReactorChannel`. The application can determine which option is desired and specify via the `ConsumerRole.dictionaryDownloadMode` parameter.

ENUMERATED NAME	DESCRIPTION
NONE	The `Reactor` will not request any dictionaries for this `ReactorChannel`. This is typically used when the application has loaded a file based dictionary or has acquired the dictionary elsewhere.
FIRST_AVAILABLE	The `Reactor` will search received directory messages for the RDMFieldDictionary (RWFFld) and the enumtype.def (RWFEnum) dictionaries. Once found, the `Reactor` will request these dictionaries for the application. After transmission is completed, the streams will be closed as this content does not update.

Table 13: `DictionaryDownloadModes` Enumerated Values

6.3.1.1.3 OMM Consumer Role Utility Method

The Transport API provides the following utility method for use with the `ConsumerRole`.

METHOD NAME	DESCRIPTION
clear	Clears the `ConsumerRole` object. Useful for object reuse.

Table 14: `ConsumerRole` Utility Method

6.3.1.2 Reactor Channel OMM Provider Role

When a `ReactorChannel` is acting as an OMM Provider application, it allows connections from OMM Consumer applications. As part of this process it is expected to respond to login requests and source directory information requests. Additionally, a provider can optionally allow consumers to download field dictionaries. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the *Transport API Java RDM Usage Guide*.

A `ReactorChannel` in an interactive provider role allows the application to specify user-defined callback methods to handle the processing of received messages on a per-domain basis.

6.3.1.2.1 OMM Provider Role

When creating a `ReactorChannel`, this information can be specified with the `ProviderRole` class, as defined below.

CLASS MEMBER	DESCRIPTION
loginMsgCallback	This `ReactorChannel`'s user-defined callback method to handle login message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. • This callback is defined in more detail in Section 6.4.6. • The login messages are described in Section 7.3.
directoryMsgCallback	This `ReactorChannel`'s user-defined callback method to handle directory message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. • This callback is defined in more detail in Section 6.4.6. • The directory messages are described in Section 7.4.
dictionaryMsgCallback	This `ReactorChannel`'s user-defined callback method to handle dictionary message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. • This callback is defined in more detail in Section 6.4.6. • The dictionary messages are described in Section 7.5.
listenerCallback	The ReactorChannel's user-defined callback method for accepting or rejecting tunnel streams. For more details on listenerCallback, refer to Section 6.6.6.

Table 15: `ProviderRole` Class Members

6.3.1.2.2 OMM Provider Role Utility Method

The Transport API provides the following utility method for use with the `ProviderRole`.

METHOD NAME	DESCRIPTION
clear	Clears the `ProviderRole` object. Useful for object reuse.

Table 16: `ProviderRole` Utility Method

6.3.1.3 Reactor Channel OMM Non-Interactive Provider Role

When a `ReactorChannel` is acting as an OMM Non-Interactive Provider application, it connects to an Enterprise Platform ADH. As part of this process it is expected to perform a login to the system. Once the login is completed, the non-interactive provider publishes a source directory, which provides information about the available services and their capabilities. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the *Transport API Java RDM Usage Guide*.

A `ReactorChannel` in a non-interactive provider role helps to simplify this connection process by exchanging these messages on the user's behalf. The user can choose to provide specific information or leverage a default populated messages, which uses the information of the user currently logged into the machine running the application. In addition, the Transport API Reactor allows the application to specify user-defined callback methods to handle the processing of received messages on a per-domain basis.

6.3.1.3.1 OMM Non-Interactive Role

When creating a `ReactorChannel`, this information can be specified with the `NIProviderRole` class, as defined below.

CLASS MEMBER	DESCRIPTION
rdmLoginRequest	The `LoginRequest`, defined in Section 7.3.1, to be sent during the connection establishment process. This can be populated with a user's specific information or invoke the `initDefaultRDMLoginRequest` method to populate with default information. If this parameter is left empty no login will be sent to the system; useful for systems that do not require a login.
rdmDirectoryRefresh	The `DirectoryRefresh`, defined in Section 7.4.2, to be sent during the connection establishment process. This can be populated with specific source directory refresh information or invoke the `initDefaultRDMDirectoryRefresh` method to populate with default information. If this parameter is specified, an `rdmLoginRequest` is required. If this parameter is left empty, no directory refresh will be sent to the system.
loginMsgCallback	This `ReactorChannel`'s user-defined callback method to handle login message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. This callback is defined in more detail in Section 6.4.6.

Table 17: `NIProviderRole` Class Members

6.3.1.3.2 OMM Non-Interactive Provider Role Utility Method

The Transport API provides the following utility method for use with the `NIProviderRole`.

METHOD NAME	DESCRIPTION
clear	Clears the `NIProviderRole` object. Useful for object reuse.

Table 18: `NIProviderRole` Utility Method

6.3.2 Adding Reactor Channels

A single `Reactor` instance can manage one or many `ReactorChannel`s. A `ReactorChannel` can be instantiated as an outbound client style connection or as a connection that is accepted from a `Server`. This allows a user to mix connection styles within or across Reactors and have a consistent usage and behavior.

Note: A single `Reactor` can simultaneously manage `ReactorChannel`s from `Reactor.connect` and `Reactor.accept`.

6.3.2.1 Reactor Connect

The `Reactor.connect` method will create a new `ReactorChannel` and associate it with a `Reactor`. This method creates a new outbound connection. The `ReactorChannel` will be returned to the application via a callback, as described in Section 6.4.6, at which point it should begin dispatching.

Client applications can specify that Reactor automatically reconnect a ReactorChannel whenever a connection fails. To enable this, the application sets the appropriate members of the ReactorConnectOptions object. The application can specify that Reactor reconnect ReactorChannel to the same host, or to one of multiple hosts.

METHOD NAME	DESCRIPTION
Reactor.connect	Creates a `ReactorChannel` that makes an outbound connection to the configured host. This establishes a connection in a manner similar to the `Transport.connect` method, as described in the *Transport API Java Developers Guide*. Connection options are passed in via the `ReactorConnectOptions`, as defined in Table 20. `ReactorChannel` specific information, such as the per-channel callback methods, the type of behavior, default RDM messages, and such are passed in via the `ReactorRole`, as defined in Section 6.3.1.

Table 19: `Reactor.connect` Method

6.3.2.2 ReactorConnectOptions Class Members

CLASS MEMBER	DESCRIPTION
connectionList	Specifies a list of `ReactorConnectOptions` as defined in Table 21. When used with `reconnectAttemptLimit`, the `Reactor` attempts to connect to each host in the list with each reconnection attempt.
reconnectAttemptLimit	The maximum number of times the `Reactor` attempts to reconnect a channel when it fails. If set to -1, there is no limit.
reconnectMinDelay	The minimum time the `Reactor` waits (in milliseconds) before attempting to reconnect a failed channel. The time increases with each reconnection attempt, from `reconnectMinDelay` to `reconnectMaxDelay`.
reconnectMaxDelay	The maximum time the `Reactor` waits (in milliseconds) before attempting to reconnect a failed channel. The time increases with each reconnection attempt, from `reconnectMinDelay` to `reconnectMaxDelay`.

Table 20: `ReactorConnectOptions` Class Members

6.3.2.3 ReactorConnectInfo Class Members

CLASS MEMBER	DESCRIPTION
connectOptions	Specifies information (`connectOptions`) about the host or network to which to connect, the type of connection to use, and other transport-specific configuration information associated with the underlying `Transport.connect` method. This is described in more detail in the *Transport API Java Developers Guide*.
initTimeout	The amount of time (in seconds) to wait to establish a `ReactorChannel`. If a timeout occurs, an event is dispatched to the application to indicate that the `ReactorChannel` is down.

Table 21: `ReactorConnectInfo` Class Members

6.3.2.4 ReactorConnectOptions Utility Method

The Transport API provides the following utility method for use with the `ReactorConnectOptions`.

METHOD NAME	DESCRIPTION
clear	Clears the `ReactorConnectOptions` object. Useful for object reuse.

Table 22: `ReactorConnectOptions` Utility Method

6.3.2.5 Reactor.connect Example

```
ReactorConnectOptions connectOpts = ReactorFactory.createReactorConnectOptions();
ConsumerRole consumerRole = ReactorFactory.createConsumerRole();

/* Configure connection options.*/
connectOpts.clear();
connectOpts.connectOptions.connectionList().get(0).connectOptions().unifiedNetworkInfo().address("localhost");
connectOpts.connectOptions.connectionList().get(0).connectOptions().unifiedNetworkInfo().serviceName("14002");

/* Configure a role for this connection as an OMM Consumer. */
consumerRole.clear();

/* Set the methods to which rsslDispatch will deliver events. */
consumerRole.channelEventCallback(channelEventCallback);
consumerRole.defaultMsgCallback(defaultMsgCallback);
consumerRole.loginMsgCallback(loginMsgCallback);
consumerRole.directoryMsgCallback(directoryMsgCallback);
consumerRole.dictionaryMsgCallback(dictionaryMsgCallback);

/* Initialize a default login request. Once the channel is initialized this message will be sent. */
consumerRole.initDefaultRDMLoginRequest();

/* Initialize a default directory request. Once the application has logged in, this message will be sent. */
consumerRole.initDefaultRDMDirectoryRequest();

/* Add the connection to the Reactor. */
ret = reactor.connect(connectOpts, consumerRole, errorInfo);
```

Code Example 2: Reactor.connect Example

6.3.2.6 Reactor Accept

The `Reactor.accept` method will create a new `ReactorChannel` and associate it with a `Reactor`. This method accepts the connection from an already running `Server`. The `ReactorChannel` will be returned to the application via a callback, as described in Section 6.4.6, at which point it can begin dispatching on the channel.

METHOD NAME	DESCRIPTION
Reactor.accept	Creates a `ReactorChannel` by accepting it from a `Server`. This establishes a connection in a manner similar to the `Server.accept` method, as described in the *Transport API Java Developers Guide*. Connection options are passed in via the `ReactorAcceptOptions`, as defined in Table 20. `ReactorChannel` specific information, such as the per-channel callback methods, the type of behavior, default RDM messages, and such are passed in via the `ReactorRole`, as defined in Section 6.3.1.

Table 23: `Reactor.accept` Method

6.3.2.7 ReactorAcceptOptions Class Members

CLASS MEMBER	DESCRIPTION
acceptOptions	The `AcceptOptions` associated with the underlying `Server.accept` method. This includes an option to reject the connection as well as a `userSpecObject`. This is described in more detail in the *Transport API Java Developers Guide*.
initTimeout	The amount of time (in seconds) to wait for the successful connection establishment of a `ReactorChannel`. If a timeout occurs, an event is dispatched to the application to indicate that the `ReactorChannel` is down.

Table 24: `ReactorAcceptOptions` Class Members

6.3.2.8 ReactorAcceptOptions Utility Method

The Transport API provides the following utility method for use with the `ReactorAcceptOptions`.

METHOD NAME	DESCRIPTION
clear	Clears the `ReactorAcceptOptions` object. Useful for object reuse.

Table 25: `ReactorAcceptOptions` Utility Method

6.3.2.9 Reactor.accept Example

```
RsslReactorAcceptOptions reactorAcceptOpts = ReactorFactory.createReactorAcceptOptions();
ProviderRole providerRole = ReactorFactory.createProviderRole();

/* Configure accept options.*/
reactorAcceptOpts.clear();
reactorAcceptOpts.acceptOptions().userSpecObject(server);

/* Configure a role for this connection as an OMM Provider. */
providerRole.clear();
providerRole.channelEventCallback(channelEventCallback);
providerRole.defaultMsgCallback(defaultMsgCallback);
providerRole.loginMsgCallback(loginMsgCallback);
providerRole.directoryMsgCallback(directoryMsgCallback);
```

```
providerRole.dictionaryMsgCallback(dictionaryMsgCallback);

/* Add the connection to the Reactor by accepting it from a Server. */
ret = reactor.accept(server, reactorAcceptOpts, providerRole, errorInfo)
```

Code Example 3: Reactor.accept Example

6.3.3 Removing Reactor Channels

6.3.3.1 ReactorChannel.close Method

The following method can be used to remove a `ReactorChannel` from a `Reactor` instance. It will also close and clean up any resources associated with the `ReactorChannel`.

METHOD NAME	DESCRIPTION
ReactorChannel.close	Removes a `ReactorChannel` from the corresponding `Reactor` and cleans up associated resources. This will additionally invoke the `Channel.close` method, as described in the *Transport API Java Developers Guide*, to clean up any resources associated with the underlying `Channel`. This method can be called from either outside or within a callback.

Table 26: `ReactorChannel.close` Method

6.3.3.2 ReactorChannel.close Example

```
ReactorErrorInfo errorInfo = ReactorFactory.createReactorErrorInfo();

/* Can be used inside or outside of a callback */
ret = reactorChannel.close(errorInfo);
```

Code Example 4: ReactorChannel.close Example

6.4 Dispatching Data

Once an application has a `Reactor`, it can begin dispatching messages. Until there is at least one associated `ReactorChannel`, there is nothing to dispatch. Once there are `ReactorChannel`s to dispatch over, each channel will begin seeing its user-defined per-channel callbacks get invoked. See Section 6.4.6 for more information about the available callbacks and their specifications.

An application can choose to dispatch across all associated `ReactorChannel`s (via `Reactor.dispatchAll`) or to dispatch on a particular `ReactorChannel` (via `ReactorChannel.dispatch`). If dispatching on a single `ReactorChannel`, only data for this channel will be processed and returned via the channel's callback. If dispatching across multiple `ReactorChannel`s, the `Reactor` attempts to fairly dispatch over all channels. In either case, the dispatch call allows the application to specify the maximum number of messages that will be processed and returned via callback.

Typically, an application will register both the `Reactor`'s internal `ReactorChannel`'s selectable channel and each `ReactorChannel`'s selectable channel with a select notifier. The select notifier can help inform the application when data is available on particular `ReactorChannel`s or when channel information is available from the `Reactor` (via its internal `ReactorChannel`). An application can also forgo select notifier use and periodically call the dispatch method to ensure that data is processed.

Note: Applications should not call `Reactor.shutdown`, `Reactor.dispatchAll`, or `ReactorChannel.dispatch` from within a callback method. All other Reactor functionality is safe to use from within a callback.

Events received in callback methods should be assumed to be invalid once the callback method returns. For callbacks that provide `Msg`, `LoginMsg`, `DirectoryMsg`, or `DictionaryMsg` objects, a deep copy of the object should be made if the application wishes to preserve it. To copy a `Msg` object, see the `Msg.copy` method in the *Transport API Java Developers Guide*; for copying a `LoginMsg`, `DirectoryMsg`, or `DictionaryMsg` object, see the copy utility method for the appropriate RDM message type.

6.4.1 Reactor.dispatchAll Method

METHOD NAME	DESCRIPTION
Reactor.dispatchAll	This method process events and messages across the provided `Reactor` and all of its associated `ReactorChannel`s. When channel information or data is available for a `ReactorChannel`, the channels user-defined callback method will be invoked. The application can control the maximum number of messages dispatched with a single call to `Reactor.dispatchAll`. This can be controlled through the passed in `ReactorDispatchOptions`, as described in Table 29.

Table 27: `Reactor.dispatchAll` Method

6.4.2 ReactorChannel.dispatch Method

METHOD NAME	DESCRIPTION
ReactorChannel.dispatch	This method processes a specific channel's events and messages from the `Reactor`. When channel information or data is available for a `ReactorChannel`, the channel's user-defined callback method is invoked. The application can control the maximum number of messages dispatched with a single call to `ReactorChannel.dispatch`. This can be controlled through passed-in `ReactorDispatchOptions` (for details refer to Table 29.

Table 28: `ReactorChannel.dispatch` Method

6.4.3 Reactor Dispatch Options

The `ReactorDispatchOptions` allow the application to control the call to `Reactor.dispatchAll` and `ReactorChannel.dispatch`.

CLASS MEMBER	DESCRIPTION
maxMessages	Controls the maximum number of events or messages processed in this call. If this is larger than the number of available messages, `Reactor.dispatchAll` or `ReactorChannel.dispatch` will return when there is no more data to process. This value is initialized to allow for up to 100 messages to be returned with a single call to `Reactor.dispatchAll` or `ReactorChannel.dispatch`.
readArgs	The `ReadArgs` from the underlying `Channel.read` call.

Table 29: `ReactorDispatchOptions` Class Members

6.4.4 ReactorDispatchOptions Utility Method

The Transport API provides the following utility method for use with the `ReactorDispatchOptions`.

METHOD NAME	DESCRIPTION
clear	Clears the `ReactorDispatchOptions` object. Useful for object reuse.

Table 30: `ReactorDispatchOptions` Utility Method

6.4.5 ReactorChannel.dispatch Example

```
ReactorDispatchOptions dispatchOpts = ReactorFactory.createReactorDispatchOptions();

/* Set dispatching options. */
dispatchOpts.clear();
dispatchOpts.maxMessages(200);

/* Call ReactorChannel.dispatch(). It will keep dispatching events
 * until there is nothing to read or maxMessages is reached. */
ret = reactorChannel.dispatch(dispatchOpts, errorInfo);
```

Code Example 5: ReactorChannel.dispatch Example

6.4.6 Reactor Callback Methods

Information about the state of the `ReactorChannel` connection as well as any messages for that channel are returned to the application via a series of callback methods. Each `ReactorChannel` can define its own unique callback methods or specify callback methods that can be shared across several or all channels.

There are several values that can be returned from a callback method implementation. These can trigger specific `Reactor` behavior based on the outcome of the callback method. The callback return values are described in the following table.

RETURN CODE	DESCRIPTION
SUCCESS	Indicates that the callback method was successful and the message or event has been handled.
FAILURE	Indicates that the message or event has failed to be handled. Returning this code from any callback method will cause the `Reactor` to shutdown.
RAISE	Can be returned from any domain-specific callback (e.g., `RDMLoginMsgCallback`). This will cause the `Reactor` to invoke the `DefaultMsgCallback` for this message upon the domain-specific callbacks return.

Table 31: ReactorCallbackReturnCodes Callback Return Codes

All events communicated to callback methods have the same base class, the `ReactorEvent`, which contains information common to all callback events.

CLASS MEMBER	DESCRIPTION
reactorChannel	The `ReactorChannel` on which the event occurred.
errorInfo	The `ReactorErrorInfo` associated with this event.

Table 32: `ReactorEvent` Class Members

6.4.6.1 Reactor Channel Event Callback

The `ReactorChannelEventCallback` is used to communicate `ReactorChannel` and connection state information to the application. This interface has the following callback method:

`reactorChannelEventCallback(ReactorChannelEvent event)`

When invoked, this will return the `ReactorChannelEvent` object, containing more information about the event.

6.4.6.1.1 Reactor Channel Event

The `ReactorChannelEvent` is returned to the application via the `ReactorChannelEventCallback`.

CLASS MEMBER	DESCRIPTION
eventType	The type of event that has occurred on the `ReactorChannel`. See Table 34.

Table 33: `ReactorChannelEvent` Class Members

6.4.6.1.2 Reactor Channel Event Type Enumeration Values

FLAG ENUMERATION	MEANING
INIT	Channel event initialization value. This should not be used by the application or returned to the application.
CHANNEL_UP	Indicates that the `ReactorChannel` has been successfully initialized and can be directly dispatched on. Where applicable, any specified Login, Directory, or Dictionary messages will now be exchanged by the `Reactor`.
CHANNEL_DOWN	Indicates that the `ReactorChannel` is not available for use. This could be a result of an initialization failure, a ping timeout, or some other kind of connection related issue. The `ReactorErrorInfo` will contain more detailed information about what occurred. There is no connection recovery for this event. The application should call `ReactorChannel.close` to clean up the failed `ReactorChannel`.
CHANNEL_DOWN_RECONNECTING	Indicates that the `ReactorChannel` is temporarily unavailable for use. The Reactor will attempt to reconnect the channel according to the values specified in `ReactorConnectOptions` when `Reactor.connect` was called. This only occurs on client connections since there is no connection recovery for server connections. Before exiting the `channelEventCallback`, the application should release any resources associated with the channel, such as `TransportBuffer`s, and unregister its `selectableChannel`, if valid, from any select notifiers.
CHANNEL_READY	Indicates that the `ReactorChannel` has successfully completed any necessary initialization process. Where applicable, this includes exchange of any provided Login, Directory, or Dictionary content. The application should now be able to consume or provide any content.
WARNING	Indicates that the `ReactorChannel` has experienced an event that did not result in connection failure, but may require the attention of the application. The `ReactorErrorInfo` will contain more detailed information about what occurred.
FD_CHANGE	Indicates that a selectable channel change occurred on the `ReactorChannel`. If the application is using a select notification mechanism, it should unregister the `oldSelectableChannel` and register the `selectableChannel`, both of which can be found on the `ReactorChannel`.

Table 34: `ReactorChannelEventTypes` Enumeration Values

6.4.6.1.3 Reactor Channel Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ReactorChannelEvent` object.

Table 35: `ReactorChannelEvent` Utility Methods

6.4.6.1.4 Reactor Channel Event Callback Example

```
public int reactorChannelEventCallback(ReactorChannelEvent event)
{
    switch(event.eventType())
    {
        case ReactorChannelEventTypes.CHANNEL_UP:
            // register selector with channel event's reactorChannel
            event.reactorChannel().selectableChannel().register(selector,
                                    SelectionKey.OP_READ,
                     event.reactorChannel());
        break;

        case ReactorChannelEventTypes.CHANNEL_DOWN:
                 // close ReactorChannel
                 if (event.reactorChannel() != null)
                 {
                     event.reactorChannel().close(errorInfo);
                 }
        break;

        case ReactorChannelEventTypes.CHANNEL_READY:
            /* Channel has exchanged its initial messages (if any were provided on the role object)
             * and is ready for use. */
            sendItemRequests(reactorChannel);
        break;

        case ReactorChannelEventTypes.FD_CHANGE:
            /* The descriptor representing this channel has changed. Normally the application only
             * needs to update its notification mechanism in response to this event. */
            // cancel old reactorChannel select
                   SelectionKey key = event.reactorChannel().oldSelectableChannel().keyFor(selector);
                   key.cancel();
                   // register selector with channel event's new reactorChannel
            event.reactorChannel().selectableChannel().register(selector,
                                    SelectionKey.OP_READ,
                     event.reactorChannel());
        break;

    }
    return ReactorCallbackReturnCodes.SUCCESS;
}
```

Code Example 6: Reactor Channel Event Callback Example

6.4.6.2 Reactor Default Message Callback

The `Reactor` default message callback is used to communicate all received content that is not handled directly by a domain-specific callback method. This callback will also be invoked after any domain-specific callback that returns the **ReactorCallbackReturnCodes.RAISE** value. This interface has the following callback method:

```
public int defaultMsgCallback(ReactorMsgEvent event)
```

When invoked, this returns a `ReactorMsgEvent` object, containing more information about the event.

6.4.6.2.1 Reactor Message Event

The `ReactorMsgEvent` is returned to the application via the `DefaultMsgCallback`. This is also the base class of the `RDMDictionaryMsgEvent`, `RDMDirectoryMsgEvent`, and `RDMLoginMsgEvent` classes.

CLASS MEMBER	DESCRIPTION
transportBuffer	A `TransportBuffer` containing the raw, undecoded message that was read and processed by the callback.
msg	A `Msg` object that has been populated with the message content by calling `Msg.decode`. If not present, an error was encountered while processing the information.

Table 36: `ReactorMsgEvent` Class Members

6.4.6.2.2 Reactor Message Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ReactorMsgEvent` object.

Table 37: `ReactorMsgEvent` Utility Methods

6.4.6.2.3 Reactor Message Event Callback Example

```
public int defaultMsgCallback(ReactorMsgEvent event)
{
    Msg msg = event.msg();

    /* Received a Msg --- or, if the decode failed, an error. */
    /* The Msg will have already been passed through Msg.decode.
       Only the payload requires additional decoding. */
    if (msg != null)
        processMsg(msg);
    else
        System.out.printf("defaultMsgCallback Error: %s(%s)\n",
                          event.errorInfo().error().text(),
                          event.errorInfo().location());
}
```

Code Example 7: Reactor Message Event Callback Example

6.4.6.3 Reactor RDM Login Message Callback

The `Reactor` RDM Login Message callback is used to communicate all received RDM Login messages. This interface has the following callback method:

```
public int rdmLoginMsgCallback(RDMLoginMsgEvent event)
```

When invoked, this will return the `RDMLoginMsgEvent` object, containing more information about the event.

6.4.6.3.1 Reactor RDM Login Message Event

The `RDMLoginMsgEvent` is returned to the application via the `RDMLoginMsgCallback`.

CLASS MEMBER	DESCRIPTION
rdmLoginMsg	The RDM representation of the decoded Login message. If not present, an error was encountered while processing the information. This message is presented as the `LoginMsg`, described in Section 7.3.

Table 38: `RDMLoginMsgEvent` Class Members

6.4.6.3.2 Reactor RDM Login Message Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears an `RDMLoginMsgEvent` object.

Table 39: `RDMLoginMsgEvent` Utility Methods

6.4.6.3.3 Reactor RDM Login Message Event Callback Example

```
public int rdmLoginMsgCallback(RDMLoginMsgEvent event)
{
    LoginMsg loginMsg = event.rdmLoginMsg();

    /* Received an RDM LoginMsg --- or, if the decode failed, an error. */
    /* The login message will already be fully decoded. */
    if (loginMsg != null)
    {
        switch(loginMsg.rdmMsgType())
        {
            case REFRESH:
                        LoginRefresh refresh = (LoginRefresh)loginMsg;
            break;
            case STATUS:
                        LoginStatus status = (LoginStatus)loginMsg;
            break;
            default:
                        System.out.println("Received unhandled login message.");
            break;
        }
    }
    else
        System.out.printf("rdmLoginMsgCallback Error: %s(%s)\n",
                          event.errorInfo().error().text(),
                          event.errorInfo().location());
}
```

Code Example 8: Reactor RDM Login Message Event Callback Example

6.4.6.4 Reactor RDM Directory Message Callback

The `Reactor` RDM Directory Message callback is used to communicate all received RDM Directory messages. This interface has the following callback method:

`public int rdmDirectoryMsgCallback(RDMDirectoryMsgEvent event)`

When invoked, this will return the `RDMDirectoryMsgEvent` object, containing more information about the event.

6.4.6.4.1 Reactor RDM Directory Message Event

The `RDMDirectoryMsgEvent` is returned to the application via the `RDMDirectoryMsgCallback`.

CLASS MEMBER	DESCRIPTION
rdmDirectoryMsg	The RDM representation of the decoded Source Directory message. If not present, an error was encountered while processing the information. This message is presented as the `DirectoryMsg`, described in Section 7.4.

Table 40: `RDMDirectoryMsgEvent` Class Members

6.4.6.4.2 Reactor RDM Directory Message Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears an `RDMDirectoryMsgEvent` object.

Table 41: `RDMDirectoryMsgEvent` Utility Methods

6.4.6.4.3 Reactor RDM Directory Message Event Callback Example

```
public int rdmDirectoryMsgCallback(RDMDirectoryMsgEvent event)
{
    DirectoryMsg directoryMsg = event.rdmDirectoryMsg();

    /* Received an RDM DirectoryMsg --- or, if the decode failed, an error. */
    /* The directory message will already be fully decoded. */
    if (directoryMsg != null)
    {
        switch(directoryMsg.rdmMsgType())
        {
            case REFRESH:
                DirectoryRefresh refresh = (DirectoryRefresh)directoryMsg;
            break;
            case UPDATE:
                DirectoryUpdate update = (DirectoryUpdate)directoryMsg;
            break;
            case STATUS:
                DirectoryStatus status = (DirectoryStatus)directoryMsg;
            break;
            default:
                System.out.println("Received unhandled directory message.");
        }
    }
    else
        System.out.printf("rdmDirectoryMsgCallback Error: %s(%s)\n",
                          event.errorInfo().error().text(),
                          event.errorInfo().location());
}
```

Code Example 9: Reactor RDM Directory Message Event Callback Example

6.4.6.5 Reactor RDM Dictionary Message Callback

The `Reactor` RDM Dictionary Message callback is used to communicate all received RDM Dictionary messages. This interface has the following callback method:

`public int rdmDictionaryMsgCallback(RDMDictionaryMsgEvent event)`

When invoked, this will return the `RDMDictionaryMsgEvent` object, containing more information about the event.

6.4.6.5.1 Reactor RDM Dictionary Message Event

The `RDMDictionaryMsgEvent` is returned to the application via the `RDMDictionaryMsgCallback`.

CLASS MEMBER	DESCRIPTION
rdmDictionaryMsg	The RDM representation of the decoded Dictionary message. If not present, an error was encountered while processing the information. This message is presented as the `DictionaryMsg`, described in Section 7.5.

Table 42: `RDMDictionaryMsgEvent` Class Members

6.4.6.5.2 Reactor RDM Dictionary Message Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears an `RDMDictionaryMsgEvent` object.

Table 43: `RDMDictionaryMsgEvent` Utility Methods

6.4.6.5.3 Reactor RDM Dictionary Message Event Callback Example

```
public int rdmDictionaryMsgCallback(RDMDictionaryMsgEvent event)
{
    DictionaryMsg dictionaryMsg = event.rdmDictionaryMsg();

    /* Received an RDM DictionaryMsg --- or, if the decode failed, an error. */
    if (dictionaryMsg != null)
    {
        switch(dictionaryMsg.rdmMsgType())
        {
            case REFRESH:
                DictionaryRefresh refresh = (DictionaryRefresh)dictionaryMsg;
            break;
            case STATUS:
                DictionaryStatus status = (DictionaryStatus)dictionaryMsg;
         break;
            default:
                System.out.println("Received unhandled dictionary message.");
        }
    }
    else
       System.out.printf("rdmDictionaryMsgCallback Error: %s(%s)\n",
                         event.errorInfo().error().text(),
                         event.errorInfo().location());
}
```

Code Example 10: Reactor RDM Dictionary Message Event Callback Example

6.5 Writing Data

The Transport API Reactor helps streamline the high performance writing of content. The `Reactor` flushes content to the network so the application does not need to. The `Reactor` does so through the use of a separate worker thread that becomes active whenever there is queued content that needs to be passed to the connection.

The Transport API Reactor offers two methods for writing content: `ReactorChannel.submit(Msg…)` and `ReactorChannel.submit(TransportBuffer…)`. When writing applications to the Reactor, consider which is most appropriate for your needs.

ReactorChannel.submit(Msg…)

- Takes a `Msg` object as part of its options; does not require retrieval of a `TransportBuffer` from the channel.

ReactorChannel.submit(TransportBuffer…)

- Takes a `TransportBuffer` which the application retrieves from the channel.
- More efficient: the application encodes directly into the buffer, and can use buffer packing.

6.5.1 Writing Data using ReactorChannel.submit(Msg…)

`ReactorChannel.submit(Msg…)` provides a simple interface for writing `Msgs`. To send a message, the application populates a `Msg` object, sets any other desired options on a `ReactorSubmitOptions` object, and calls `ReactorChannel.submit(Msg…)` with the object.

A buffer is not needed to use `ReactorChannel.submit(Msg…)`. If the application needs to include any encoded content, it can encode the content into any available memory, and set the appropriate member of the `Msg` to point to the memory (as well as set the length of the encoded content).

6.5.1.1 ReactorChannel.submit(Msg…) Method

METHOD NAME	DESCRIPTION
ReactorChannel.submit	Encodes and submits a Msg to the Reactor. This method expects a properly populated `Msg`.

Table 44: `ReactorChannel.submit(Msg…)` Method

6.5.1.2 ReactorChannel.submit(Msg…) Return Codes

The following table defines the return codes that can occur when using `ReactorChannel.submit(Msg…)`.

RETURN CODE	DESCRIPTION
ReactorReturnCodes.SUCCESS	Indicates that the `submit(Msg…)` method has succeeded.
ReactorReturnCodes.NO_BUFFERS	Indicates that the message was currently unable to be written due to a lack of available pool buffers. The application can try to submit the message later, or it can optionally use `ReactorChannel.ioctl` to increase the number of available pool buffers and try again.
ReactorReturnCodes.WRITE_CALL_AGAIN	Indicates that buffer created for the `Msg` is being fragmented and needs to be called again with the same `Msg`. This indicates that underlying write was unable to send all fragments with the current call and must continue fragmenting.
ReactorReturnCodes.FAILURE	Indicates that a general failure has occurred and the message was not submitted. The ReactorErrorInfo object passed to the method will contain more details.

Table 45: `ReactorChannel.submit(Msg…)` Return Codes

6.5.1.3 ReactorChannel.submit(Msg…) Example

The following example shows typical use of `ReactorChannel.submit(Msg…)`.

```
RequestMsg requestMsg = (RequestMsg)CodecFactory.createMsg();
ReactorSubmitOptions opts = ReactorFactory.createReactorSubmitOptions();
ReactorErrorInfo errorInfo = ReactorFactory.createReactorErrorInfo();
int ret;

requestMsg.clear();
requestMsg.msgClass(MsgClasses.REQUEST);
requestMsg.streamId(2);
requestMsg.domainType(DomainTypes.MARKET_PRICE);
requestMsg.containerType(DataTypes.NO_DATA);
requestMsg.applyStreaming();
requestMsg.applyHasQos();
requestMsg.qos().timeliness(QosTimeliness.REALTIME);
requestMsg.qos().rate(QosRates.TICK_BY_TICK);
requestMsg.msgKey().applyHasName();
requestMsg.msgKey().applyHasServiceId();
requestMsg.msgKey().name().data("TRI.N");
requestMsg.msgKey().serviceId(1);

ret = reactorChannel.submit(requestMsg, opts, errorInfo);
```

6.5.2 Writing data using ReactorChannel.submit(TransportBuffer…)

The `ReactorChannel.submit(TransportBuffer…)` method offers efficient writing of data by using buffers retrieved directly from the Transport API transport buffer pool. It also provides additional features not normally available from ReactorChannel.submit(Msg…), such as buffer packing. When ready to send data, the application acquires a buffer from the Transport API pool. This allows the content to be encoded directly into the output buffer, reducing the number of times the content needs to be copied. Once content is encoded and the buffer is properly populated, the application can submit the data to the Reactor. The Transport API will ensure that successfully submitted buffers reach the network. Applications can also pack multiple messages into a single buffer by following a similar process as described above, however instead of getting a new buffer for each message the application uses the Reactor's pack method instead. The following flow chart depicts the typical write process.

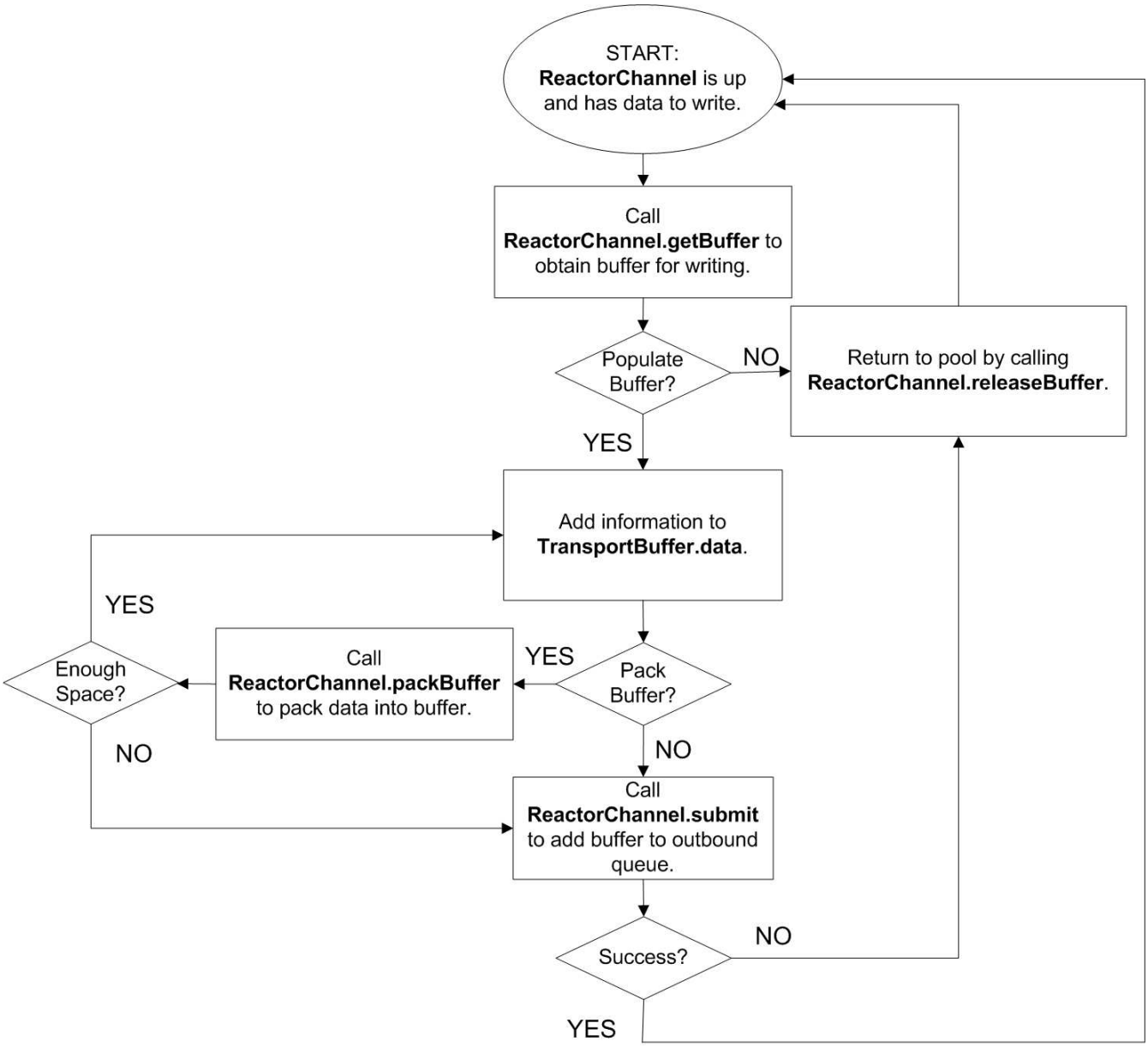

Figure 5: Flow Chart for writing data via ReactorChannel.submit(TransportBuffer…)

6.5.2.1 Obtaining a Buffer for Writing

Before information can be submitted, the user is required to obtain a buffer from the internal Transport API buffer pool, as described in the *Transport API Java Developers Guide*. After a buffer is acquired via `ReactorChannel.getBuffer`, the user can populate the `TransportBuffer.data`. If the buffer is not used or the `ReactorChannel.submit(TransportBuffer…)` method call fails, the buffer must be released back into the pool to ensure proper reuse and cleanup. If the buffer is successfully passed to `ReactorChannel.submit(TransportBuffer…)`, the Transport API Reactor will return the buffer to the pool.

The number of buffers made available to a `ReactorChannel` is configurable through the `ReactorConnectOptions` or `ReactorAcceptOptions`. See Table 20 and Table 24 for more information about available `Reactor.connect` and `Reactor.accept` options.

6.5.2.1.1 ReactorChannel Buffer Management Methods

METHOD NAME	DESCRIPTION
getBuffer	Obtains a buffer of the requested size from the buffer pool. When the `TransportBuffer` is returned, the `length` member indicates the number of bytes available in the buffer (this should match the amount the application requested). When populating, the `length` reflects the number of bytes actually used. This ensures that only the required bytes are written to the network. If the requested size is larger than the `maxFragmentSize`, the transport will create and return the buffer to the user. When written, this buffer will be fragmented by the `submit(TransportBuffer…)` method (See Section 6.5.2.2). Because of some additional book keeping required when packing, the application must specify whether a buffer should be 'packable' when calling `getBuffer`. For more information on packing, see Section 6.5.2.3. For performance purposes, an application is not permitted to request a buffer larger than `maxFragmentSize` and have the buffer be 'packable.' If the buffer is not used or the `submit(TransportBuffer…)` call fails, the buffer must be returned to the pool using `releaseBuffer`. If the `submit(TransportBuffer…)` call is successful, the buffer will be returned to the correct pool by the transport. This method calls the `Channel.getBuffer` method which has its use and return values described in the *Transport API Java Developers Guide*.
releaseBuffer	Releases a buffer back to the correct pool. This should only be called with buffers that originate from `getBuffer` and are not successfully passed to `submit(TransportBuffer…)`. This method calls the Transport API `Channel.releaseBuffer` method which has its use and return values described in the *Transport API Java Developers Guide*.
bufferUsage	Returns the number of buffers currently in use by the `ReactorChannel`, this includes buffers that the application holds and buffers internally queued and waiting to be flushed to the connection by the `Reactor`. This method calls the `Channel.bufferUsage` method which has its use and return values described in the *Transport API Java Developers Guide*.

Table 46: ReactorChannel Buffer Management Methods

6.5.2.1.2 ReactorChannel.getBuffer Return Values

The following table defines return and error code values that can occur while using `ReactorChannel.getBuffer`.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	A `TransportBuffer` is returned to the user. The `TransportBuffer.length` indicates the number of bytes available to populate and the `TransportBuffer.data` provides a ByteBuffer for population.
NULL buffer returned Error Code: NO_BUFFERS	NULL is returned to the user. This value indicates that there are no buffers available to the user. See `ReactorErrorInfo` content for more details. This typically occurs because all available buffers are queued and pending flushing to the connection. The `ReactorChannel.ioctl` method can be used to increase the number of `guaranteedOutputBuffers` (see Section 0).
NULL buffer returned Error Code: FAILURE	NULL is returned to the user. This value indicates that some type of general failure has occurred. The `ReactorChannel` should be closed.
NULL buffer returned Error Code: INIT_NOT_INITIALIZED	Indicates that the underlying Transport API Transport has not been initialized. See the `ReactorErrorInfo` content for more details.

Table 47: `ReactorChannel.getBuffer` Return Values

6.5.2.2 Writing Data using ReactorChannel.submit(TransportBuffer…)

After a `TransportBuffer` is obtained from `getBuffer` and populated with the user's data, the buffer can be passed to the `submit(TransportBuffer…)` method. This method manages queuing and flushing of user content. It will also perform any fragmentation or compression. If an unrecoverable error occurs, any `TransportBuffer` that has not been successfully passed to `submit(TransportBuffer…)` should be released to the pool using `releaseBuffer`. The following table describes the `submit(TransportBuffer…)` method as well as some additional parameters associated with it.

6.5.2.2.1 ReactorChannel.submit(TransportBuffer…) Method

METHOD NAME	DESCRIPTION
submit	Performs writing of data. This method expects the buffer to be properly populated. This method allows for several modifications and additional parameters to be specified via the `ReactorSubmitOptions` object, defined in 6.5.2.2.2. The return values are described in 6.5.2.2.4. This method calls the `Channel.write` method and will also trigger the `Channel.flush` method. These are described in the *Transport API Java Developers Guide*.

Table 48: `ReactorChannel.submit(TransportBuffer…)` Method

6.5.2.2.2 Reactor Submit Options

The `ReactorSubmitOptions` allow the application control various aspects of the call to `ReactorChannel.submit`.

CLASS MEMBER	DESCRIPTION
writeArgs.priority	Controls the priority at which the data will be written. Valid priorities are • **WritePriorities.HIGH** • **WritePriorities.MEDIUM** • **WritePriorities.LOW** More information about write priorities, including an example scenario, are available in the *Transport API Java Developers Guide.*
writeArgs.flags	Flag values that allow the application to modify the behavior of this ReactorChannel.submit call. This includes options to bypass queuing or compression. More information about the specific flag values are available in the *Transport API Java Developers Guide.*
writeArgs.bytesWritten	If specified, will return the number of bytes to be written, including any transport header overhead and taking into account any savings from compression.
writeArgs.uncompressedBytesWritten	If specified, will return the number of bytes to be written, including any transport header overhead but not taking into account any compression savings.

Table 49: ReactorSubmitOptions Class Members

6.5.2.2.3 RsslReactorSubmitOptions Utility Method

The Transport API provides the following utility method for use with the `ReactorSubmitOptions`.

METHOD NAME	DESCRIPTION
Clear	Clears the `ReactorSubmitOptions` object. Useful for object reuse.

Table 50: `ReactorSubmitOptions` Utility Method

6.5.2.2.4 ReactorChannel.submit(TransportBuffer…) Return Codes

The following table defines the return codes that can occur when using `ReactorChannel.submit(TransportBuffer…)`.

RETURN CODE	DESCRIPTION
ReactorReturnCodes.SUCCESS	Indicates that the `submit(TransportBuffer…)` method has succeeded. The `TransportBuffer` will be released by the Transport API Reactor.
ReactorReturnCodes.WRITE_CALL_AGAIN	Indicates that a large buffer could not be fully written with this `submit(TransportBuffer…)` call. This is typically due to all pool buffers becoming unavailable. The `Reactor` will flush for the user in order to free up buffers. The application can optionally use `ReactorChannel.ioctl` to increase the number of available pool buffers. After pool buffers become available again, the same buffer should be used to call `submit(TransportBuffer…)` an additional time (the same priority level must be used to ensure proper ordering of each fragment). This will continue the fragmentation process from where it left off. If the application does not subsequently pass the buffer to `submit(TransportBuffer…)`, the application should release it by calling `ReactorChannel.releaseBuffer`.
ReactorReturnCodes.FAILURE	Indicates that a general write failure has occurred. The `ReactorChannel` should be closed. The application should release the `TransportBuffer` by calling `ReactorChannel.releaseBuffer`.

Table 51: `ReactorChannel.submit(TransportBuffer…)` Return Codes

6.5.2.2.5 ReactorChannel.getBuffer and ReactorChannel.submit(TransportBuffer…) Example

The following example shows typical use of `ReactorChannel.getBuffer` and `ReactorChannel.submit(TransportBuffer…)`.

```
TransportBuffer msgBuffer = null;
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
ReactorSubmitOptions submitOpts = ReactorFactory.createReactorSubmitOptions();

msgBuffer = reactorChannel.getBuffer(1024, false, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(), reactorChannel.minorVersion());
encodeMsgIntoBuffer(encodeIter, msgBuffer);

submitOpts.clear();
ret = reactorChannel.submit(msgBuffer, submitOpts, errorInfo);
// check return code
switch (ret)
{
    case ReactorReturnCodes.SUCCESS:
            // successful write, nothing left to do
            return ReactorReturnCodes.SUCCESS;
    break;
    case ReactorReturnCodes.FAILURE:
            // an error occurred, need to release buffer
            reactorChannel.releaseBuffer(msgBuffer, errorInfo);
    break;
```

```
    case ReactorReturnCodes.WRITE_CALL_AGAIN:
            // large message couldn't be fully written with one call, pass it to submit again
            ret = reactorChannel.submit(msgBuffer, submitOpts, errorInfo);
    break;
}
```

Code Example 11: Writing Data Using `ReactorChannel.submit(TransportBuffer…)`, `ReactorChannel.getBuffer`, and `ReactorChannel.releaseBuffer`

6.5.2.3 Packing Additional Data into a Buffer

If an application is writing many small buffers, it may be advantageous to combine the small buffers into one larger buffer. This can increase efficiency of the transport layer by reducing the overhead associated with each write operation, although it may add to the latency associated with each smaller buffer.

It is up to the writing application to determine when to stop packing, and the mechanism used can vary greatly. One simple algorithm that can be used is to pack a fixed number of messages each time. A slightly more complex technique could use the length returned from `ReactorChannel.packBuffer` to determine the amount of space remaining, and pack until the buffer is nearly full. Both of these mechanisms can introduce a variable amount of latency as they both depend on the rate of arrival of data (e.g. the packed buffer will not be written until enough data arrives to fill it). One method that can balance this would also employ a timer, used to limit the amount of time a packed buffer is held. If the buffer is full prior to the timer expiring, the data is written however when the timer expires the buffer will be written regardless of the amount of data it contains. This can help to limit the latency introduced and hold it to a maximum amount associated with the duration of the timer.

METHOD NAME	DESCRIPTION
`ReactorChannel.packBuffer`	Packs contents of passed in `TransportBuffer` and returns the amount of available bytes remaining in the buffer for packing. An application can use the length returned to determine the amount of space available to continue packing buffers into. For a buffer to allow packing, it must be requested from `ReactorChannel.getBuffer` as 'packable' and cannot exceed the `maxFragmentSize`. `ReactorChannel.packBuffer` return values are defined in Table 53. This method calls the `Channel.packBuffer` method which has its use and return values described in the *Transport API Java Developers Guide*.

Table 52: `ReactorChannel.packBuffer` Method

6.5.2.4 ReactorChannel.packBuffer Return Values

The following table defines return and error code values that can occur when using `ReactorChannel.packBuffer`.

RETURN CODE	DESCRIPTION
Positive value or ReactorReturnCodes.SUCCESS Success Case	The amount of available bytes remaining in the buffer for packing.
Negative value Failure Case	This value indicates that some type of failure has occurred. If the FAILURE return code is returned, the `ReactorChannel` should be closed. This method calls the `Channel.packBuffer` method which has its return values described in the *Transport API Java Developers Guide*.

Table 53: `ReactorChannel.packBuffer` Return Values

6.5.2.5 Example: ReactorChannel.getBuffer, ReactorChannel.packBuffer, and ReactorChannel.submit(TransportBuffer…)

The following example shows typical use of `ReactorChannel.getBuffer`, `ReactorChannel.packBuffer`, and `ReactorChannel.submit(TransportBuffer…)`.

```
int remainingLength = 0;
TransportBuffer msgBuffer = null;
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
ReactorSubmitOptions submitOpts = ReactorFactory.createReactorSubmitOptions();

/* get a packable buffer */
msgBuffer = reactorChannel.getBuffer(1024, true, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(), reactorChannel.minorVersion());
encodeMsgIntoBuffer(encodeIter, msgBuffer);

/* pack first encoded message into buffer */
remainingLength = reactorChannel.packBuffer(msgBuffer, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(), reactorChannel.minorVersion());
encodeMsgIntoBuffer(encodeIter, msgBuffer);

/* pack second encoded message into buffer */
remainingLength = reactorChannel.packBuffer(msgBuffer, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(), reactorChannel.minorVersion());

/* now write packed buffer by passing third buffer to submit */
encodeMsgIntoBuffer(encodeIter, msgBuffer);

submitOpts.clear();
ret = reactorChannel.submit(msgBuffer, submitOpts, errorInfo);
```

Code Example 12: Message Packing using `ReactorChannel.packBuffer`

6.6 Creating and Using Tunnel Streams

The Reactor allows users to create and use tunnel streams. A tunnel stream is a private stream with additional behaviors, such as end-to-end line of sight for authentication and guaranteed delivery. Tunnel streams are founded on the private streams concept, and the Transport API establishes them between consumer and provider endpoints (passing through any intermediate components, such as TREP or EED).

When creating a tunnel, the consumer indicates any additional behaviors to enforce, which is exchanged with the provider application end point. The provider end point acknowledges the creation of the stream as well as any behaviors that it will enforce. After the stream is established, the consumer can exchange any content it wants, though the tunnel stream will enforce behaviors on the transmitted content as negotiated with the provider.

A tunnel stream allows for multiple substreams to exist, where substreams follow from the same general stream concept, except that they flow and coexist within the confines of a single tunnel stream.

In the following diagram, the orange cylinder represents a tunnel stream that connects the consumer application to the provider application. Notice that this passes through intermediate components. The tunnel stream has end-to-end line of sight so the provider and consumer effectively communicate with each other directly, even though they traverse multiple devices in the system. Each black line flowing through the cylinder represents a different substream, with each substream transmitting its own independent stream of information. Each of these could be for different market content; for example: one could be a Time Series request while another could be a request for Market Price content. A substream can also connect to a special provider application called a Queue Provider. A Queue Provider allows for persistence of content exchanged over the tunnel stream and substream and helps provide content beyond the end point visible to the consumer. To interact with a Queue Provider, additional addressing information is required, described in more detail in Section 7.6.

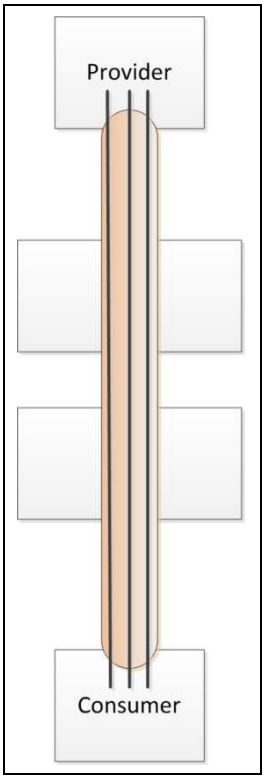

Figure 6: Tunnel Streams

6.6.1 Authenticating a Tunnel Stream

Providers might require the consumer to authenticate itself when establishing the tunnel stream. The type of authentication, if any, is given by the `ClassOfService.authentication.type`. For more information, refer to Section 6.6.3.

The `ClassOfService.authentication.type` may be set to **OMM_LOGIN**. When an OMM Consumer expects this type of authentication, it should set a `LoginRequest` message on the `TunnelStreamOpenOptions.authLoginRequest` member. If the OMM Consumer application does not provide it, the API will use the login request provided on the `ConsumerRole.rdmLoginRequest` when the consumer connected (refer to Section 6.3.1.1). The consumer must provide one of these for authentication of this type.

The login request will be sent to the provider. When the provider sends a Login response to complete the authentication, the `TunnelStreamStatusEvent` event given to the consumer will include a `TunnelStreamAuthInfo` structure with more details (refer to Table 54). OMM Provider applications will see the login request as a normal message within the `TunnelStream` and should respond with a login response message via `TunnelStream.submit`.

Other types of authentication may be specified, but must be performed by both the Provider and Consumer applications by submitting normal `TunnelStream` messages via `TunnelStream.submit`.

The `TunnelStreamAuthInfo` structure contains the following member:

MEMBER	DESCRIPTION
loginMsg	The Login message sent by the tunnel stream's provider application, which resulted in this event.

Table 54: `TunnelStreamAuthInfo` Structure Members

6.6.2 Opening a Tunnel Stream

The user can create one or more tunnel streams via `ReactorChannel`, which opens the private stream connection and negotiates any specified behaviors. Prior to opening a tunnel stream, you must implement `StatusEventCallback`, described in Section 6.6.4.1.

6.6.2.1 `ReactorChannel.openTunnelStream` Method

METHOD NAME	DESCRIPTION
openTunnelStream	Starts the establishment of a tunnel stream. `TunnelStream` is returned via the `StatusEventCallback` as specified on the `TunnelStreamOpenOptions`. For more details, refer to Section 6.6.2.2.

Table 55: `ReactorChannel.openTunnelStream` Method

6.6.2.2 TunnelStreamOpenOptions

`TunnelStreamOpenOptions` contains event handler associations and options for use in creating a tunnel stream.

CLASS MEMBER	DESCRIPTION
domainType	Indicates the domain for which the Transport API establishes the tunnel stream. Set this to the domain specified on the service on which the Transport API opens the tunnel stream.
streamId	Indicates the tunnel stream's ID. Though substreams will flow within this stream ID, each will have their own independent stream ID. For example, a tunnel stream can have an ID of 10. If a substream is opened to retrieve TRI data, the substream can have a stream ID of 5, though it is encapsulated in the tunnel stream whose stream ID is 10.

CLASS MEMBER	DESCRIPTION
serviceId	Indicates the service ID of the service on which you open the tunnel stream.
authLoginRequest	Specifies the login request to send if `ClassOfService.authentication.type` is set to **OMM_LOGIN**. If absent, the API uses the login request provided on the `ConsumerRole.rdmLoginRequest`.
classOfService	The class of service of the tunnel stream to be opened. For details on classOfService, refer to Section 6.6.3.
name	Sets the name of the tunnel stream to open.
guaranteedOutputBuffers	Sets the number of guaranteed output buffers available for the tunnel stream.
responseTimeout	Sets the amount of time (in seconds) to wait for a provider to respond to a TunnelStream open request. If the provider does not respond in time, a `TunnelStreamStatusEvent` is sent to the application to indicate the `TunnelStream` could not be opened.
userSpecObject	Indicates a user-specified object passed in via these options and then associated with the `TunnelStream`.
statusEventCallback	Specifies the instance of the callback for `TunnelStreamStatusEvents`, which provides the `TunnelStream` on initial connection, and after the tunnel stream is established, communicates the tunnel stream's state information. For details on the `StatusEventCallback`, refer to Section 6.6.4.1.
queueMsgCallback	Specifies the instance of the callback used to handle Queue Messages received on this `TunnelStream`. • For details on the `TunnelStreamQueueMsgCallback`, refer to Section 6.6.4.1. • For details on various Queue Messages, refer to Section 7.6.
defaultMsgCallback	Specifies the instance of the callback that handles all other content received on this `TunnelStream`. For details on the `DefaultMsgCallback`, refer to Section 6.6.4.1.

Table 56: Tunnel Stream Open Options

6.6.3 Negotiating Stream Behaviors: Class of Service

`ClassOfService` is used to negotiate behaviors of a `TunnelStream`. Negotiated behaviors are divided into five categories: common, authentication, flow control, data integrity, and guarantee.

- When an OMM consumer application calls `ReactorChannel.openTunnelStream`, it sets the `TunnelStreamOpenOptions.classOfService` members to manage and control tunnel stream behaviors. The consumer passes these settings to the connected OMM Provider.
- When the OMM provider application receives a `TunnelStreamRequestEvent`, the provider calls `TunnelStreamRequestEvent.classOfService` to retrieve the behaviors requested by the consumer.

After tunnel stream negotiation is complete, the provider and consumer each receives a `TunnelStreamStatusEvent` where each can view negotiated behaviors on the `TunnelStream` structure.

Note: Do not modify the `ClassOfService` member of the `TunnelStream`.

The enumerations given for members described in this section can be found in `com.thomsonreuters.upa.rdm.ClassesOfService`.

6.6.3.1 ClassOfService Common Members

Common elements describe options related to the exchange of messages, such as maximum message size and the desired exchange protocol.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
maxMsgSize	6144	1 – 2,147,483,647	The maximum size of messages exchanged on the tunnel stream. This value is only set by providers when accepting a tunnel stream.
protocolType	Codec.protocolType()	0 – 255	Identifies the protocol of the messages exchanged on the tunnel stream.
protocolMajorVersion	Codec.majorVersion()	0 – 255	The major version of the protocol specified by protocolType.
protocolMinorVersion	Codec.minorVersion()	0 – 255	The minor version of the protocol specified by protocolType.

Table 57: `ClassOfService.common` Structure Members

6.6.3.2 ClassOfService Authentication Member

The authentication member contains options to authenticate a consumer to the corresponding provider.

MEMBER	DEFAULT	RANGE/ENUMERATIONS	DESCRIPTION
type	**ClassesOfService. AuthenticationTypes. *NOT_REQUIRED***	**ClassesOfService. AuthenticationTypes. *NOT_REQUIRED* == 0, ClassesOfService. AuthenticationTypes. OMM_LOGIN == 1**	Indicates the type of authentication, if any, to perform on this tunnel stream. For more information on performing authentication, refer to Section 6.6.1.

Table 58: `ClassOfService.authentication` Structure Members

6.6.3.3 ClassOfService Flow Control Members

The flow control member contains options related to flow control, such as the type and the allowed window of outstanding data.

MEMBER	DEFAULT	RANGE/ENUMERATIONS	DESCRIPTION
type	**ClassesOfService. FlowControlTypes. NONE**	**ClassesOfService. FlowControlTypes. NONE == 0, ClassesOfService. FlowControlTypes. BIDIRECTIONAL == 1**	Indicates the type of flow control (if any) to apply to the tunnel stream.
recvWindowSize	-1	0 – 2,147,483,647	Sets the amount of content (in bytes) that the remote peer can send to the application over a reliable tunnel stream. If `type` is set to **NONE**, this parameter has no effect. **-1** indicates that the application wishes to use the default value for the negotiated flow control type. If that type is **BIDIRECTIONAL**, the default is **12288**.

MEMBER	DEFAULT	RANGE/ENUMERATIONS	DESCRIPTION
sendWindowSize	None	0 – 2,147,483,647	Indicates the amount of content (in bytes) that the application can send to its peer on a reliable tunnel stream. This value is provided on the `TunnelStream` object and does not need to be set when opening or accepting a tunnel stream. This value is retrieved from the remote end and is informational, as the flow control is performed by the API. When room is available in the window, the API transmits more content as submitted by the application. If `type` is **NONE**, this parameter has no effect.

Table 59: `ClassOfService.flowControl` Structure Members

6.6.3.4 ClassOfService Data Integrity Member

The data integrity member contains options related to the reliability of content exchanged over the tunnel stream.

MEMBER	DEFAULT	RANGE	DESCRIPTION
type	**ClassesOfService.DataIntegrityTypes.BEST_EFFORT**	**ClassesOfService.DataIntegrityTypes.BEST_EFFORT == 0, ClassesOfService.DataIntegrityTypes.RELIABLE == 1**	Sets the level of reliability for message transmission on the tunnel stream. If set to **RELIABLE**, data is retransmitted as needed over the tunnel stream to ensure that all messages are delivered, and in the correct order. **Note:** At this time, **RELIABLE** is the only supported option.

Table 60: `ClassOfService.dataIntegrity` Structure Members

6.6.3.5 ClassOfService Guarantee Members

The guarantee member contains options related to the guarantee of content submitted over the tunnel stream.

OMM consumers performing queue messaging to a queue provider should set `ClassOfService.guarantee.type` to **ClassesOfService.GuaranteeTypes.PERSISTENT_QUEUE**.

MEMBER	DEFAULT	RANGE	DESCRIPTION
type	**ClassesOfService.GuaranteeTypes.NONE**	**ClassesOfService.GuaranteeTypes.NONE == 0, ClassesOfService.GuaranteeTypes.PERSISTENT_QUEUE == 1**	Indicates the level of guarantee that will be performed on this stream. **PERSISTENT_QUEUE** is not supported for provider applications. **Note:** If `type` is set to **PERSISTENT_QUEUE** for a consumer application, the data integrity `type` must also be set to **RELIABLE** and the flow control `type` to **BIDIRECTIONAL**.
persistLocally	**true**	**false**, **true**	Indicates whether messages are persisted locally on the tunnel stream. When `type` is **NONE**, this member has no effect.

MEMBER	DEFAULT	RANGE	DESCRIPTION
persistenceFilePath	NULL	n/a	File path where files containing persistent messages may be stored. If set to NULL, the current working directory is used. When `type` is **NONE** or `persistLocally` is set to **RSSL_FALSE**, this member has no effect.

Table 61: `ClassOfService.guarantee` Structure Members

6.6.4 Stream Callback Methods and Event Types

6.6.4.1 Tunnel Stream Callback Methods

The `TunnelStream` delivers events via several callback methods implemented by the user (listed in the following table). These callback methods return event objects as defined in Section 6.6.4.2.

CALLBACK METHOD	DESCRIPTION
statusEventCallback	Communicates status information about the tunnel stream. Additionally, this callback delivers the `TunnelStream` object after the enhanced private stream is established. This callback provides a `TunnelStreamStatusEvent` to the application. Details about this event are available in Section 6.6.4.2.
defaultMsgCallback	Similar to `ReactorChannel.defaultMsgCallback`, content received by the tunnel stream is returned via this callback if it is not handled by a more specific content handler (such as the `queueMsgCallback`). This callback provides a `TunnelStreamMsgEvent` to the application. Details about this event are available in Section 6.6.4.2.
queueMsgCallback	Any queue messages are delivered via this callback and presented to the user in their native queue message formats. If not specified, queue messages are delivered via the `DefaultMsgCallback`, however they will not be presented in a Queue Message format. This callback provides a `TunnelStreamQueueMsgEvent` to the application. Details about this event are available in Section 6.6.4.2.

Table 62: Tunnel Stream Callback Methods

6.6.4.2 Tunnel Stream Callback Event Types

Various tunnel stream callbacks return their information via specific event objects.

EVENT	EVENT DESCRIPTION	CLASS MEMBER	CLASS MEMBER DESCRIPTION
TunnelStreamStatusEvent	This event presents the tunnel stream and its status to the user.	tunnelStream	Returns the `TunnelStream` associated with this event. Upon initial opening of the `TunnelStream`, the initial instance of the `TunnelStream` is made available.

EVENT	EVENT DESCRIPTION	CLASS MEMBER	CLASS MEMBER DESCRIPTION
		state	Indicates status information associated with the TunnelStream. For example: • A state of **OPEN** and **OK** indicates that the tunnel stream is established and content is flowing as expected. • A state of **CLOSED_RECOVER** or **SUSPECT** indicates that the connection or tunnel stream might be lost. However if the tunnel stream is performing guaranteed messaging, content might be persisted by the reactor and communicated upon recovery of the tunnel stream.
		authInfo	(Consumers only) Provides information about a received authentication response.
TunnelStreamMsgEvent	This event presents content received on the TunnelStream. If a more specific handler (i.e., the queueMsgCallback) is also configured, messages of that type will go to their specific handler.	tunnelStream	Returns the TunnelStream associated with this event.
		msg	Contains the content being presented to the user. • If content is OMM, it is partially decoded for user convinenence. • If content is opaque, a buffer housing the contents is presented to the user via this object.
		transportBuffer	The transport buffer associated with this event.
		containerType	The container type associated with this event's transport buffer.
TunnelStreamQueueMsgEvent	This event is used to present any queue message content received on the TunnelStream.	tunnelStream	Returns the TunnelStream associated with this event.
		queueMsg	Contains the content being presented to the user, presented as a queue message object.

Table 63: Tunnel Stream Callback Event Types

6.6.5 Code Sample: Opening and Managing a Tunnel Stream

The following code sample is a basic example of opening a tunnel stream. The example assumes that a reactor and ReactorChannel are already open and properly established.

```
// Basic sample for event handlers
class  Sample implements StatusEventCallback, TunnelStreamQueueMsgCallback, TunnelStreamDefaultMsgCallback
{
    ReactorErrorInfo _errorInfo;

    // StatusEventCallback
    public int statusEventCallback(TunnelStreamStatusEvent event)
    {
        System.out.println("Status of Tunnel Stream (" + event.tunnelStream().streamId() + ") is " +
                event.state());
        Return ReactorCallbackReturnCodes.SUCCESS;
```

```
    }

    // TunnelStreamDefaultMsgCallback
    public int TunnelStreamDefaultMsgCallback(TunnelStreamMsgEvent event)
    {
        System.out.println("Received content on Tunnel Stream (" + event.tunnelStream().streamId() +
                ")");
        Return ReactorCallbackReturnCodes.SUCCESS;
    }

    // TunnelStreamQueueMsgCallback
    public int tunnelStreamQueueMsgCallback(TunnelStreamQueueMsgEvent event)
    {
        System.out.println("Received Queue Message on Tunnel Stream (" + event.tunnelStream().streamId()
                + ")");
        Return ReactorCallbackReturnCodes.SUCCESS;
    }
}

int openTunnelStream()
{
    TunnelStreamOpenOptions _openOptions = RectorFactory.createTunnelStreamOpenOptions();

    // populate the options and enable guaranteed delivery for communication with a Queue Provider
    _openOptions.streamId(TUNNEL_STREAM_ID);
    _openOptions.domainType(DomainTypes.QUEUE_MESSAGING);
    _openOptions.serviceId(QUEUE_MESSAGING_SERVICE_ID);
    // specify the event handlers
    _openOptions.statusEventCallback(this);
    _openOptions.TunnelStreamDefaultMsgCallback(this);
    _openOptions.queueMsgCallback(this);

    if ((reactorChannel.openTunnelStream(_openOptions, _errorInfo)) != ReactorReturnCodes.SUCCESS)
    {
        System.out.println("openTunnelStream failed!");
        return ReactorReturnCodes.FAILURE;
    }

    System.out.println("openTunnelStream succeeded!");
    return ReactorReturnCodes.SUCCESS;
}
```

Code Example 13: Opening a Tunnel Stream

6.6.6 Accepting Tunnel Streams

OMM Provider applications can accept tunnel streams provided on a `ReactorChannel` (enabled by specifying a `tunnelStreamListenerCallback` on the `ProviderRole`).

When consumer opens a tunnel stream, the `tunnelStreamListenerCallback` receives a `TunnelStreamRequestEvent`. At this point, the provider should call `TunnelStreamRequestEvent.classOfService` to retrieve the `ClassOfService` requested by the tunnel stream, and ensure that the parameters indicated by the members of that class of service match what the provider allows. The provider can also check the `TunnelStreamRequestEvent.classOfServiceFilter` to determine which behaviors are supported by the consumer. For more information on this filter, refer to Section 6.6.6.1.

- To accept a tunnel stream, the provider must call `ReactorChannel.acceptTunnelStream` with the given `TunnelStreamRequestEvent`. Further events regarding the accepted stream are included in the specified `TunnelStreamAcceptOptions.statusEventCallback`.
- To reject a tunnel stream, the provider calls `ReactorChannel.rejectTunnelStream` with the given `TunnelStreamRequestEvent`. No further events will be received for that tunnel stream.

Queue messaging (a `ClassOfService.guarantee.type` setting of **PERSISTENT_QUEUE**) is not supported for Provider applications.

The API will automatically reject tunnel streams that contain invalid information. If this happens, the provider application receives warnings via a `ReactorChannelEvent`. The type will be set to **ReactorChannelEventTypes WARNING** and the `ReactorErrorInfo` in the event will contain text describing the reason for the rejection.

Note: Ensure that the provider application calls `ReactorChannel.acceptTunnelStream` or `ReactorChannel.rejectTunnelStream` before returning from the `tunnelStreamListenerCallback`. If not, the Provider application will receive a warning via a `ReactorChannelEvent` similar to the above, and the stream will be automatically rejected.

6.6.6.1 Reactor Tunnel Stream Listener Callback

OMM providers that want to handle tunnel streams from connected consumers can specify a `TunnelStreamListenerCallback`. This callback informs the provider application of any consumer tunnel stream requests.

The provider can specify this callback function on the `ProviderRole`, which has the following signature:

```
listenerCallback(TunnelStreamRequestEvent event);
```

For more information on the `ProviderRole`, refer to Section 6.3.1.2.

A `TunnelStreamRequestEvent` is returned to the application via the `TunnelStreamListenerCallback`.

STRUCTURE MEMBER	DESCRIPTION
reactorChannel	Specifies the `ReactorChannel` on which the event was received.
streamId	Specifies the stream ID of the requested tunnel stream.
domainType	Specifies the domain type of the requested tunnel stream.
serviceId	Specifies the service IDd of the requested tunnel stream.
name	Specifies the name of the requested tunnel stream.
classOfServiceFilter	Specifies a filter indicating which `ClassOfService` members are present. The provider can use this filter to determine whether behaviors are supported by the consumer, and if needed reject the tunnel stream before calling `TunnelStreamRequestEvent.classOfService` to get the full `ClassOfService`. For enumerations of the flags present in this filter, see `com.thomsonreuters.upa.rdm.ClassesOfService.FilterFlags`.
classOfService	Specifies the `ClassOfService` for the requested tunnel stream.

Table 64: `TunnelStreamRequestEvent` Structure Members

6.6.6.2 ReactorChannel.acceptTunnelStream Method

METHOD NAME	DESCRIPTION
ReactorChannel.acceptTunnelStream	Accepts a tunnel stream requested by a consumer. The `TunnelStream` will be returned in the `TunnelStreamStatusEventCallback` specified on the `TunnelStreamAcceptOptions`. For more information, refer to Section 6.6.6.3.

Table 65: `ReactorChannel.acceptTunnelStream` Method

6.6.6.3 TunnelStreamAcceptOptions

CLASS MEMBER	DESCRIPTION
statusEventCallback	Specifies the instance of the callback for `TunnelStreamStatusEvents`, which provides the `TunnelStream` on initial connection and then communicates state information about the tunnel stream afterwards. For details on `TunnelStreamStatusEventCallback`, refer to Section 6.6.4.1.
defaultMsgCallback	Specifies the instance of the callback that will handle all other content received on this `TunnelStream`. For details on `TunnelStreamDefaultMsgCallback`, refer to Section 6.6.4.1.
userSpecObject	Specifies a user-defined object passed in via these options and then associated with the `TunnelStream`.
classOfService	A `ClassOfService` with members indicating behaviors the application wants to assign to the `TunnelStream`. For more information on class of service, refer to Section 6.6.3.
guaranteedOutputBuffers	Sets the number of pooled buffers available to the application when writing content to the `TunnelStream`.

Table 66: `TunnelStreamAcceptOptions` Structure Members

6.6.6.4 ReactorChannel.rejectTunnelStream

METHOD NAME	DESCRIPTION
ReactorChannel.rejectTunnelStream	Rejects a tunnel stream requested by a consumer. No futher events will be received for this tunnel stream. For more information, refer to Section 6.6.6.5.

Table 67: `ReactorChannel.rejectTunnelStream` Method

6.6.6.5 TunnelStreamRejectOptions

CLASS MEMBER	DESCRIPTION
state	A `State` to send to the consumer. The application can use the `state.streamState`, `state.dataState`, and `state.text` to indicate the nature of the rejection.
expectedClassOfService	An optional `ClassOfService` to send to the consumer. If rejecting the stream due to a problem with the `ClassOfService` parameters from the `TunnelStreamRequestEvent`, the provider application should populate this with the associated parameters.

Table 68: `TunnelStreamRejectOptions` Structure Members

6.6.6.6 Accepting a Tunnel Stream Code Sample

The following code illustrates how to accept a tunnel stream requested by a consumer. The example assumes that a Reactor and ReactorChannel are already open and properly established.

```
public int listenerCallback(TunnelStreamRequestEvent event)
{
    int ret;
    TunnelStreamAcceptOptions acceptOpts = ReactorFactory.createTunnelStreamAcceptOptions();

    if (isFilterValid(event.classOfServiceFilter()) &&
        isClassOfServiceValid(event.classOfService())
    {
        acceptOpts.clear();

        // set class of service to what this provider supports
        acceptOpts.classOfService().dataIntegrity().type(ClassesOfService.DataIntegrityTypes.RELIABLE);
        acceptOpts.classOfService().flowControl().type((ClassesOfService.FlowControlTypes.BIDIRECTIONAL)
                );

        // Set Authentication to match consumer. This provider will perform OMM Login authentication if
                requested.
        acceptOpts.classOfService().authentication().type(event.classOfService().authentication().type()
                );

        acceptOpts.statusEventCallback(this);
        acceptOpts.defaultMsgCallback(this);

        if ((ret = event.reactorChannel().acceptTunnelStream(event, acceptOpts, event.errorInfo()))
                < ReactorReturnCodes.SUCCESS)
        {
        System.out.println("acceptTunnelStream() failed with return code: " + ret + " <" +
                event.errorInfo().error().text() + ">");
        }
    }

    return ReactorCallbackReturnCodes.SUCCESS
}
```

Code Example 69: Accepting a Tunnel Stream Code Example

6.6.6.7 Rejecting a Tunnel Stream Code Sample

The following code illustrates how to reject a tunnel stream requested by a consumer. The example assumes that a Reactor and ReactorChannel are already open and properly established.

```
public int listenerCallback(TunnelStreamRequestEvent event)
{
    int ret;

    /* Now presuming that the application wishes to reject the tunnel stream
     * because the requested class of service is invalid. */

    if (!isFilterValid(event.classOfServiceFilter()) ||
        !isClassOfServiceValid(event.classOfService())
```

```
    {
        /* Set what the class of service is expected to be. */
        ClassOfService expectedCos = ReactorFactory.createClassOfService();
        expectedCos.clear()
        expectedCos.authentication().type(ClassesOfService.AuthenticationTypes.OMM_LOGIN);
        expectedCos.flowControl().type(ClassesOfService.FlowControlTypes.BIDIRECTIONAL);
        expectedCos.dataIntegrity().type(ClassesOfService.DataIntegrityTypes.RELIABLE);
        /* ... (set additional members, based on what is desired by the provider) */

        TunnelStreamRejectOptions rejectOpts = ReactorFactory.createTunnelStreamRejectOptions();
        rejectOpts.clear();
        rejectOpts.state().streamState(StreamStates.CLOSED);
        rejectOpts.state().dataState(DataStates.SUSPECT);
        rejectOpts.state().code(StateCodes.NONE);
        rejectOpts.state().text().data("Unsupported TunnelStream class of service");
        rejectOpts.expectedClassOfService(expectedCos);

        if ((ret = event.reactorChannel().rejectTunnelStream(event, rejectOpts, event.errorInfo())) <
       ReactorReturnCodes.SUCCESS)
        {
            System.out.println("rejectTunnelStream() failed with return code: " + ret + " <" +
       event.errorInfo().error().text() + ">");
        }
    }

    return ReactorCallbackReturnCodes.SUCCESS
}
```

Code Example 70: Rejecting a Tunnel Stream Code Example

6.6.7 Receiving Content on a TunnelStream

Invoking the `Reactor.dispatch` method reads and processes inbound content, where any information received on this `TunnelStream` is delivered to the application via the tunnel stream callback methods specified via `ReactorChannel.openTunnelStream` or `ReactorChannel.acceptTunnelStream`.

Dispatching this content works in the same manner as dispatching any other content on the Reactor.

- Tunnel stream callback methods are described in Section 6.6.4.1.
- Tunnel stream callback methods deliver the events described in Section 6.6.4.2.

6.6.8 Sending Content on a TunnelStream

When you send content on a `TunnelStream`: get a buffer from the `TunnelStream`, encode your content into the buffer, and then use the `TunnelStream.submit` method to push the content out over the `TunnelStream`. By obtaining a buffer from the `TunnelStream`, the reactor can then properly handle any negotiated behaviors, making this functionality nearly transparent.

6.6.8.1 Tunnel Stream Buffer Methods

METHOD NAME	DESCRIPTION
getBuffer	Obtains a buffer from the `TunnelStream`. To properly enforce negotiated behaviors on content in that buffer, the Transport API associates the buffer with the tunnel stream from which it is obtained.

METHOD NAME	DESCRIPTION
releaseBuffer	Releases a buffer back to the TunnelStream from which it came (release the buffer if you do not submit it). Releasing the buffer ensures it is properly recycled and can be reused. **Note:** If you submit a buffer properly, you do not need to release it, as the submit method will handle this after the content is sent on the TunnelStream.

Table 71: Tunnel Stream Buffer Methods

6.6.8.2 Tunnel Stream Submit

The submit method writes content to the TunnelStream. This method also enforces any specified behaviours on the content being submitted (e.g., if guaranteed messaging is specified, this content will follow all configured persistence options).

METHOD NAME	DESCRIPTION
submit	Use the submit method to pass in opaque or RDM Message content (including Queue Messages) to be processed and sent over the TunnelStream. This method has additional options that can be specified via TunnelStreamSubmitOptions (refer to Table 73).

Table 72: Tunnel Stream Submit Method

This structure provides options when calling TunnelStream.submit with a buffer.

MEMBER	DESCRIPTION
containerType	The type of content in the buffer being submitted. For example: • If the submitted buffer contains a Msg, set containerType to **DataTypes.MSG**. • If non-RWF data is sent, set containerType to a Non-RWF type such as **DataTypes.OPAQUE**. For more information on possible container types, refer to the *Transport API C Edition Developers Guide*.

Table 73: TunnelStreamSubmitOptions Structure Members

6.6.8.3 Submitting Content on a Tunnel Stream Code Sample

The following code sample illustrates writing opaque content to a tunnel stream. You can combine this example with the QueueData message samples in subsequent chapters to send content to a Queue Provider.

```
int submitMessage()
{
    TunnelStreamSubmitOptions _submitOpts = ReactorFactory.createTunnelStreamSubmitOptions();

    // gets a buffer of 50 bytes to put content into.
    TransportBuffer _buffer = tunnelStream.getBuffer(50, _errorInfo);

    // put generic content into the buffer
    _buffer.data().put("Hello World!");
    _submitOpts.containerType(DataTypes.OPAQUE);

    if ((tunnelStream.submit(_buffer, _submitOpts, _errorInfo)) != ReactorReturnCodes.SUCCESS)
```

```
    {
        System.out.println("Content submission failed!");
        // Because submission failed, we need to return the buffer to the tunnel stream
        tunnelStream.releaseBuffer(_buffer, _errorInfo);

        return ReactorReturnCodes.FAILURE;
    }

    System.out.println("Content submission succeeded!");
    // Thanks to successful submission, we do not need to release the buffer because the Reactor will.
    return ReactorReturnCodes.SUCCESS;
}
```

Code Example 14: Submitting Content on a Tunnel Stream

6.6.9 Closing a Tunnel Stream

When an application has completed its use of a TunnelStream, close it using the `close` method.

6.6.9.1 Tunnel Stream Close

METHOD NAME	DESCRIPTION
close	Closes a tunnel stream. After you close a tunnel stream, the Transport API cleans up any data that was stored for guaranteed messaging or reliable delivery. The `finalStatusEvent` argument indicates that the application wants to receive a final `TunnelStreamStatusEvent` whenever the tunnel stream finally closes. If this is set to true, the tunnel stream will be cleaned up after the application receives the final `TunnelStreamStatusEvent` event.

Table 74: Tunnel Closure Method

6.6.9.2 Closing a Tunnel Stream Sample

The following code sample illustrates how to close a tunnel stream.

```
int closeTunnelStream()
{

    if ((tunnelStream.close(true,_errorInfo)) != ReactorReturnCodes.SUCCESS)
    {
        System.out.println("Closing tunnel stream failed!");
        return ReactorReturnCodes.FAILURE;
    }

    System.out.println("Tunnel Stream closed successfully.");
    return ReactorReturnCodes.SUCCESS;
}
```

Code Example 15: Closing a Tunnel Stream

6.7 Reactor Utility Methods

The Transport API Reactor provides several additional utility methods. These methods can be used to query more detailed information for a specific connection or change certain `ReactorChannel` parameters during run-time. These methods are described in the following tables.

6.7.1 General Reactor Utility Methods

METHOD NAME	DESCRIPTION
info	Allows the application to query `ReactorChannel` negotiated parameters and settings and retrieve all current settings. This includes `maxFragmentSize` and negotiated compression information as well as many other values. For a full list of available settings, refer to the `ReactorChannelInfo` object defined in Table 76. This method calls the `Channel.info` method which has its use and return values described in the *Transport API Java Developers Guide*.
ioctl	Allows the application to change various settings associated with the `ReactorChannel`. The available options are defined in Section 6.7.3. This method calls the `Channel.ioctl` method which has its use and return values described in the *Transport API Java Developers Guide*.

Table 75: Reactor Utility Methods

6.7.2 ReactorChannelInfo Class Members

The following table describes the values available to the user through using the `ReactorChannel.info` method. This information is returned as part of the `ReactorChannelInfo` object.

CLASS MEMBER	DESCRIPTION
channelInfo	Returns the underlying `Channel` information. This includes `maxFragmentSize`, number of output buffers, compression information, and more. The `ChannelInfo` method object is fully described in the *Transport API Java Developers Guide*.

Table 76: `ReactorChannelInfo` Class Members

6.7.3 ReactorChannel.ioctl Option Values

There are currently no `Reactor` or `ReactorChannel` specific codes for use with the `ReactorChannel.ioctl`. Reactor specific codes may be added in the future. The application can still use any of the codes allowed with Channel.ioctl, which are documented in the *Transport API Java Developers Guide*.

Chapter 7 Administration Domain Models Detailed View

7.1 Concepts

The ***Administration Domain Model Representations*** are RDM specific representations of the OMM administrative domain models. This value added component contains classes and interfaces that represent messages within the Login, Source Directory, and Dictionary domains. This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's object members to send or receive the content. This not only significantly reduces the amount of code an application needs to interact with OMM devices (i.e., TREP infrastructure), but also makes sure that encoding/decoding for these domain models follow OMM-specified formatting rules. Applications can use this value added component directly to help with encoding, decoding, and representation of these domain models.

Where possible, the members of an Administration Domain Model Representation are represented in the class with the same `DataType` specified for the element by the Domain Model. In cases where multiple elements are part of a more complex container such as a `Map` or `ElementList`, the elements are represented with Java JDK collections.

This chapter describes the Administration Domain Model Representation classes and gives examples of their use. The *Transport API Java RDM Usage Guide* defines and describes all domain specific behaviors, usage, and details.

DOMAIN	PURPOSE
Login	Authenticates users and advertises/requests features that are not specific to a particular domain. Login is an administrative domain, content is required and expected by many Thomson Reuters components, and following the domain model definition is expected. Use of and support for this domain is required for all OMM applications. For further details refer to 7.3: **RDM Login** Domain.
Source Directory	Advertises information about available services and their state, QoS, and capabilities. This domain also conveys any group status and group merge information. Source Directory is an administrative domain, and many Thomson Reuters components expect and require content to conform to the domain model definition. Interactive and Non-Interactive OMM Provider applications require support for this domain. Thomson Reuters strongly recommends that OMM Consumers request this domain. For further details, refer to 7.4: **RDM Source Directory** Domain.
Dictionary	Provides dictionaries that may be needed when decoding data. Dicitonary is an administrative domain, content is required and expected by many Thomson Reuters components, and following the domain model definition is expected. Though use of the Dictionary domain is optional, Thomson Reuters recommends that Provider applications support the domain's use. For further details refer to 7.5: **RDM Dictionary** Domain.

Table 77: Administrative Domains

7.2 Message Base

`MsgBase` is the root interface for all Administration Domain Model Representation interfaces. It provides methods for stream identification as well methods that are common for all domain representations.

7.2.1 Message Base Members

The `MsgBase` interface includes the following members and methods:

MEMBER	DESCRIPTION
streamId	A unique, signed-integer identifier associated with all messages flowing in a stream. • Positive values indicate a consumer-instantiated stream, typically via a request message. • Negative values indicate a provider-instantiated stream, often associated with Non-Interactive Providers. `streamId` is required on all messages.

Table 78: `MsgBase` Members

METHOD NAME	DESCRIPTION
clear()	Clears the object for reuse.

Table 79: `MsgBase` Methods

7.2.2 Domain Representations Message Types

The following table provides a reference mapping between the administrative domain type and the domain representations provided in this component.

TRANSPORT APIJ DOMAIN TYPE	DOMAIN REPRESENTATIONS MESSAGE TYPE	DOMAIN REPRESENTATION INTERFACE
LOGIN (`LoginMsg`) See Section 7.3	LoginMsgType.REQUEST	LoginRequest
	LoginMsgType.REFRESH	LoginRefresh
	LoginMsgType.STATUS	LoginStatus
	LoginMsgType.CLOSE	LoginClose
	LoginMSgType.CONSUMER_CONNECTION_STATUS	LoginConsumerConnectionStatus
SOURCE (`DirectoryMsg`) See Section 7.4	DirectoryMsgType.REQUEST	DirectoryRequest
	DirectoryMsgType.REFRESH	DirectoryRefresh
	DirectoryMsgType.UPDATE	DirectoryUpdate
	DirectoryMsgType.STATUS	DirectoryStatus
	DirectoryMsgType.CLOSE	DirectoryClose
	DirectoryMsgType.CONSUMER_STATUS	DirectoryConsumerStatus
DICTIONARY (`DictionaryMsg`) See Section 7.5	DictionaryMsgType.REQUEST	DictionaryRequest
	DictionaryMsgType.REFRESH	DictionaryRefresh
	DictionaryMsgType.STATUS	DictionaryStatus
	DictionaryMsgType.CLOSE	DictionaryClose

Table 80: Domain Representations Message Types

7.3 RDM Login Domain

The Login domain registers (or authenticates) a user with the system, after which the user can request[2], post[3], or provide[4] OMM content.

- A consumer application must log into the system before it can request or post content.
- A non-interactive provider (NIP) application must log into the system before providing any content. An interactive provider application must handle log in requests and provide Login response messages, possibly using DACS to authenticate users.

The following sections detail the layout and use of each message interface in the Login portion of the Administration Domain Message component.

7.3.1 Login Request

A ***Login Request*** message is encoded and sent by the OMM Consumer and OMM non-interactive provider applications. This message registers a user with the system. After receiving a successful login response, applications can then begin consuming or providing additional content. An OMM Provider can use the Login request information to authenticate users with DACS.

The `LoginRequest` represents all members of a Login Request message and allows for simplified use in OMM applications that leverage RDM.

7.3.1.1 Login Request Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional Login Request members. See Table 83: `LoginRequestFlags` for details.
userNameType	**Optional**. Possible values: • **Login.UserIdTypes.NAME = 1** • **Login.UserIdTypes.EMAIL_ADDRESS = 2** • **Login.UserIdTypes.TOKEN = 3** A type of **USER_NAME** typically corresponds to a DACS user name and can authenticate and permission a user. **USER_TOKEN** is specified when using AAAAPI. The user token is retrieved from a AAAAPI gateway. To validate users, a provider application can pass this user token to an Authentication Manager application. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to the AAAAPI documentation. If present, `flags` value of **LoginRequestFlags.HAS_USERNAME_TYPE** should be specified. If absent, a default value of **USER_NAME** is assumed.
userName	**Required**. Populate this member with the user's name, email address, or user token based on the `userNameType` specification. If you initialize a `LoginRequest` using `initDefaultRequest`, it will use the name of the user currently logged into the system on which the application runs.

[2] Consumer applications can request content after logging into the system.

[3] Consumer applications can post content, which is similar to contribution or unmanaged publication, after logging into the system.

[4] Non-interactive provider applications.

MEMBER	DESCRIPTION
attrib	**Optional**. Contains additional login attribute information. If present, a `flags` value of **LoginRequestFlags.HAS_ATTRIB** should be specified. For further details, refer to Table 100: `LoginAttrib` Members.
instanceId	**Optional**. You can use the `instanceId` to differentiate applications running on the same machine. However, because `instanceId` is set by the user logging into the system, it does not guarantee uniqueness across different applications on the same machine. If present, a `flags` value of **LoginRequestFlags.HAS_INSTANCE_ID** should be specified.
role	**Optional**. Indicates the role of the application logging onto the system. • **0**: Indicates the application is a consumer (**Login.Role.CONS**). • **1**: Indicates the application is a provider (**Login.Role.PROV**). If present, a `flags` value of **LoginRequestFlags.HAS_ROLE** should be specified. If absent, it assumes a default value of **Login.Role.CONS**.
downloadConnectionConfig	**Optional**. Enabling this option allows the application to download information about other providers on the network. You can use such downloaded information to load balance connections across multiple providers. • **1**: Indicates that the user wants to download connection configuration information. • **0**: Indicates that the user does not want to download connection information. If present, a `flags` value of **LoginRequestFlags.HAS_DOWNLOAD_CONN_CONFIG** should be specified. If absent, it assumes a default value of **0**.

Table 81: `LoginRequest` Members

7.3.1.2 Login Request Utility Methods

METHOD NAME	DESCRIPTION
initDefaultRequest	Clears a `LoginRequest` object and populates `userName`, `position`, `applicationId`, and `applicationName` with default values.

Table 82: `LoginRequest` Utility Methods

7.3.1.3 Login Request Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_DOWNLOAD_CONN_CONFIG	Indicates the presence of `downloadConnectionConfig`. If absent, a value of **0** is assumed.
HAS_ATTRIB	Indicates the presence of `attib`.
HAS_INSTANCE_ID	Indicates the presence of `instanceId`.
HAS_PASSWORD	Indicates the presence of `password`.
HAS_ROLE	Indicates the presence of `role`. If absent, a role of **Login.RoleTypes.CONS** is assumed.
HAS_USERNAME_TYPE	Indicates the presence of `userNameType`. If absent, a `userNameType` of **Login.UserIdTypes.NAME** is assumed.
PAUSE_ALL	Indicates that the consumer wants to pause all streams associated with the logged in user. For more information on pause and resume behavior, refer to the *Transport API Java Developers Guide*.

FLAG ENUMERATION	MEANING
NO_REFRESH	Indicates that the consumer application does not needs a login refresh for this request. This typically occurs when resuming a stream or changing an AAA token. In some instances, a provider may still see fit to deliver a refresh message, however if not explicitly asked for by the consumer it should be considered unsolicited.

Table 83: `LoginRequestFlags`

7.3.2 Login Refresh

A ***Login Refresh*** message is encoded and sent by OMM interactive provider applications. This message responds to a Login Request message and indicates that the user's login was accepted. An OMM Provider can use Login Request information to authenticate users with DACS. After authentication, a refresh message is sent to convey login acceptance. If the login is rejected, a Login Status message should be sent as described in Section 7.3.3.

The `LoginRefresh` represents all members of a login refresh message and allows for simplified use in OMM applications that leverage RDM.

7.3.2.1 Login Refresh Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Login message type. **LoginMsgType.REFRESH** for login refresh.
flags	**Required**. Indicate the presence of optional login refresh members. For details, refer to Table 85.
state	**Required**. Indicates the state of the login stream. Defaults to a `streamState` of **StreamStates.OPEN** and a `dataState` of **DataStates.OK**. For more information on `State`, refer to the *Transport API Java Developers Guide*.
userNameType	**Optional**. If present, a `flags` value of **LoginRefreshFlags.HAS_USERNAME_TYPE** should be specified. If absent, a default value of **Login.UserIdTypes.NAME** is assumed. Possible values: • **Login.UserIdTypes.NAME = 1** • **Login.UserIdTypes.EMAIL_ADDRESS = 2** • **Login.UserIdTypes.TOKEN = 3** A type of **Login.UserIdTypes.NAME** typically corresponds to a DACS user name, and can be used to authenticate and permission a user. **Login.UserIdTypes.TOKEN** is specified when using the AAAAPI. The user token is retrieved from a AAAAPI gateway. To validate users, a provider application can pass this user token to an Authentication Manager application. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to specific AAAAPI documentation.
userName	**Optional**. Contains appropriate content corresponding to the `userNameType` specification. If populated, it should match the `userName` contained in the login request. If present, a `flags` value of **LoginRefreshFlags.HAS_USERNAME** should be specified.
attrib	**Optional**. Contains additional login attribute information. If present, a `flags` value of **LoginRefreshFlags.HAS_ATTRIB** should be specified. For details, refer to Table 100: `LoginAttrib` Members.
features	**Optional**. Indicates a set of features supported by the provider of the login refresh message. If present, a `flags` value of **LoginRefreshFlags.HAS_FEATURES** should be specified. For details, refer to Table 86: `LoginSupportFeatures` Members.
connectionConfig	**Optional**.Indicates the connection configuration that the consumer uses for its standby servers when setup for Warm Standby. If present, a `flags` value of **LoginRefreshFlags.HAS_CONN_CONFIG** should be specified.
sequenceNumber	**Optional**. A user-specified, item-level sequence number which can be used by the application for sequencing messages within this stream. If present, a `flags` value of **LoginRefreshFlags.HAS_SEQ_NUM** should be specified.

Table 84: `LoginRefresh` Members

7.3.2.2 Login Refresh Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_ATTRIB	Indicates the presence of `attib`.
HAS_FEATURES	Indicates the presence of `features`. If absent, a value of **0** is assumed.
SOLICITED	• If present, the Login Refresh is solicited (e.g., it is in response to a request). • If absent, the Login Refresh is unsolicited.
HAS_USERNAME	Indicates the presence of `userName`.
HAS_USERNAME_TYPE	Indicates the presence of `userNameType`. If absent a `userNameType` of **Login.UserIdTypes.NAME** is assumed.
HAS_SEQ_NUM	Indicates the presence of `sequenceNumber`.
HAS_CONN_CONFIG	Indicates the presence of connection configuration information used with warm standby functionality.
CLEAR_CACHE	Clears any stored payload information associated with the login stream. This may occur if some portion of data is known to be invalid.

Table 85: `LoginRefreshFlags`

7.3.2.3 Login Support Feature Set Members

For detailed information on posting, batch requesting, dynamic view use, and 'pause and resume,' refer to the *Transport API Java Developers Guide*.

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional login refresh members. For details, refer to Table 87: `LoginSupportFeaturesFlags`.
supportOMMPost	**Optional**. Indicates whether the provider supports OMM Posting: • **1**: The provider supports OMM Posting and the user is permissioned. • **0**: The provider supports the OMM Post feature, but the user is not permissioned. • If this element is not present, then the server does not support OMM Post feature. If present, a `flags` value of **LoginSupportFeaturesFlags.HAS_SUPPORT_POST** should be specified. If absent, a default value of **0** is assumed.
supportStandby	**Optional**. Indicates whether the provider supports warm standby functionality. If supported, a provider can run as an active or a standby server, where the active will behave as usual. The standby will respond to item requests only with the message header and will forward any state changing information. When informed of an active's failure, the standby begins sending responses and takes over as active. • **1**: The provider supports a role of active or standby in a warm standby group. • **0**: The provider does not support warm standby functionality. If present, a `flags` value of **LoginSupportFeaturesFlags.HAS_SUPPORT_STANDBY** should be specified. If absent, a default value of **0** is assumed.
supportBatchRequests	**Optional**. Indicates whether the provider supports batch functionality, which allows a consumer to specify multiple items, all with matching attributes, in the same request message. • **1**: The provider supports batch requesting. • **0**: The provider does not support batch requesting. If present, a `flags` value of **LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH** should be specified. If absent, a default value of **0** is assumed.

MEMBER	DESCRIPTION
supportViewRequests	**Optional**. Indicates whether the provider supports dynamic view functionality, which allows a user to request specific response information. • **1**: The provider supports dynamic view functionality. • **0**: The provider does not support dynamic view functionality. If present, a `flags` value of **LoginSupportFeaturesFlags.HAS_SUPPORT_VIEW** should be specified. If absent, a default value of **0** is assumed.
supportOptimizedPauseResume	**Optional**. If present, a `flags` value of **LoginSupportFeaturesFlags.HAS_SUPPORT_OPT_PAUSE** should be specified. If absent, a default value of **0** is assumed. Indicates whether the provider supports the optimized pause and resume feature. Optimized pause and resume can pause/resume individual item streams or all item streams by pausing the login stream. • **1**: The server supports optimized pause and resume. • **0**: The server does not support optimized pause and resume.

Table 86: `LoginSupportFeatures` Members

7.3.2.4 Login Support Feature Set Flag Enumeration Values

For detailed information on batch functionality, posting, 'pause and resume,' and views, refer to the *Transport API Java Developers Guide*.

FLAG ENUMERATION	MEANING
HAS_SUPPORT_BATCH	Indicates the presence of `supportBatchRequests`. If absent, a value of **0** is assumed.
HAS_SUPPORT_POST	Indicates the presence of `supportOMMPost`. If absent, a value of **0** is assumed.
HAS_SUPPORT_OPT_PAUSE	Indicates the presence of `supportOptimizedPauseResume`. If absent, a value of **0** is assumed.
HAS_SUPPORT_VIEW	Indicates the presence of `supportViewRequests`. If absent, a value of **0** is assumed.
HAS_SUPPORT_STANDBY	Indicates the presence of `supportStandby`. If absent, a value of **0** is assumed.

Table 87: `LoginSupportFeaturesFlags`

7.3.2.5 Login Connection Config Members

MEMBER	DESCRIPTION
numStandbyServers	**Required**. Indicates the number of servers in the `serverList` that the consumer can use as standby servers when using warm standby.
serverList	**Required**. A list of servers to which the consumer may connect when using warm standby.

Table 88: `LoginConnectionConfig` Members

7.3.2.6 Login Connection Config Methods

METHOD NAME	DESCRIPTION
clear	Clears a `LoginConnectionConfig` object for reuse.

METHOD NAME	DESCRIPTION
copy	Performs a deep copy of a `LoginConnectionConfig` object.

Table 89: `LoginConnectionConfig` Methods

7.3.2.7 Server Info Members

MEMBER	DESCRIPTION
flags	**Required**. Indicate the presence of optional, server information members. For details, refer to Table 92.
serverIndex	**Required**. Indicates the index value to this server.
hostname	**Required**. Indicates the `hostname` information for this server.
port	**Required**. Indicates the server's port number for connections.
loadFactor	**Optional**. Indicates the load information for this server. If present, a `flags` value of **ServerInfoFlags.HAS_LOAD_FACTOR** should be specified.
serverType	**Optional**. Indicates whether this server is an **active** or **standby** server. If present, a `flags` value of **ServerInfoFlags.HAS_TYPE** should be specified. Populated by `Login.ServerTypes`.

Table 90: `ServerInfo` Members

7.3.2.8 Server Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServerInfo` object for object reuse.
copy	Performs a deep copy of a `ServerInfo` object.

Table 91: `ServerInfo` Methods

7.3.2.9 Server Info Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_LOAD_FACTOR	Indicates the presence of `loadFactor` information.
HAS_TYPE	Indicates the presence of `serverType`.

Table 92: `ServerInfoFlags`

7.3.3 Login Status

OMM Provider and OMM non-interactive provider applications use the ***Login Status*** message to convey state information associated with the login stream. Such state information can indicate that a login stream cannot be established or to inform a consumer of a state change associated with an open login stream.

The Login Status message can also reject a login request or close an existing login stream. When a login stream is closed via a status, any other open streams associated with the user are also closed as a result.

The `LoginStatus` represents all members of a login refresh message and allows for simplified use in OMM applications that leverage RDM.

7.3.3.1 Login Status Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Login message type. **LoginMsgType.STATUS** for login status.
flags	**Required**. Indicates the presence of optional login status members. For details, refer to Table 94: `LoginStatusFlags`.
state	**Optional**. If present, a `flags` value of **LoginStatusFlags.HAS_STATE** should be specified. Indicates the state of the login stream. When rejecting a login the state should be: • `streamState` = **StreamStates.CLOSED** or **StreamStates.CLOSED_RECOVER** • `dataState` = **DataStates.DATA_SUSPECT** • `stateCode` = **DataStates.NOT_ENTITLED** For more information on `State`, refer to the *Transport API Java Developers Guide*.
userNameType	**Optional**. Possible values: • **Login.UserIdTypes.NAME = 1** • **Login.UserIdTypes.EMAIL_ADDRESS == 2** • **Login.UserIdTypes.TOKEN == 3** If present, a `flags` value of **LoginStatusFlags.HAS_USERNAME_TYPE** should be specified. If absent, a default value of **Login.UserIdTypes.NAME** is assumed. A type of **Login.UserIdTypes.NAME** typically corresponds to a DACS user name. This can be used to authenticate and permission a user. **Login.UserIdTypes.TOKEN** is specified when using AAAAPI The user token is retrieved from an AAAAPI gateway. To validate users, a provider application can pass this user token to an Authentication Manager application. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to specific AAAAPI documentation.
userName	**Optional**. If present, a `flags` value of **Login.UserIdTypes.NAME** should be specified. When populated, this should match the `userName` contained in the login request.

Table 93: `LoginStatus` Members

7.3.3.2 Login Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_STATE	Indicates the presence of `state`. If absent, any previously conveyed state continues to apply.
HAS_USERNAME	Indicates the presence of `userName`.
HAS_USERNAME_TYPE	Indicates the presence of `userNameType`. If absent a `userNameType` of **Login.UserIdTypes.NAME** is assumed.

Table 94: `LoginStatusFlags`

7.3.4 Login Close

A ***Login Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to log out of the system. Closing a login stream is equivalent to a ***Close All*** type of message, where all open streams are closed (i.e., all streams associated with the user are closed). A provider can log off a user and close all of that user's streams via a Login Status message, see Section 7.3.3.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Login message type. For a login close, set **LoginMsgType.CLOSE**.

Table 95: `LoginClose` Members

7.3.5 Login Consumer Connection Status

The ***Login Consumer Connection Status*** informs an interactive provider of its role in a ***Warm Standby*** group, either as an ***Active*** or ***Standby*** provider. An active provider behaves normally; however a standby provider responds to requests only with a message header (intended to allow a consumer application to confirm the availability of their requested data across active and standby servers), and forwards any state-related messages (i.e., unsolicited refresh messages, status messages). The standby provider aggregates changes to item streams whenever possible. If a provider changes from Standby to Active via this message, all aggregated update messages are passed along. When aggregation is not possible, a full, unsolicited refresh message is passed along.

The consumer application is responsible for ensuring that items are available and equivalent across all providers in a warm standby group. This includes managing state and availability differences as well as item group differences.

The `LoginConsumerConnectionStatus` relies on the `GenericMsg` and represents all members necessary for applications that leverage RDM.

7.3.5.1 Login Consumer Connection Status Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Indicates the Login Message type. For login connection status, set to **LoginMsgType.CONSUMER_CONNECTION_STATUS**.
flags	**Required**. Indicates the presence of optional login consumer connection status members. For details, refer to Table 97: `LoginConsumerConnectionStatusFlags`.
warmStandbyInfo	**Optional**. Includes `LoginWarmStandbyInfo` to convey the state of the upstream provider. If present, a `flags` value of **LoginConsumerConnectionStatusFlags.HAS_WARM_STANDBY_INFO** should be specified. For details, refer to Table 98.

Table 96: `LoginConsumerConnectionStatus` Members

7.3.5.2 Login Consumer Connection Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_WARM_STANDBY_INFO	Indicates the presence of `warmStandbyInfo`.

Table 97: `LoginConsumerConnectionStatusFlags`

7.3.5.3 Login Warm Standby Info Members

MEMBER	DESCRIPTION
action	**Required**. Indicates how a cache of Warm Standby content should apply this information. For information on `MapEntryActions`, refer to the *Transport API Java Developers Guide*.
warmStandbyMode	**Required**. Indicates whether a server is active (**Login.ServerTypes.ACTIVE**) or standby (**Login.ServerTypes.SERVER**).

Table 98: `LoginWarmStandbyInfo` Members

7.3.5.4 Login Warm Standby Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a `LoginWarmStandbyInfo` object for reuse.
copy	Performs a deep copy of a `LoginWarmStandbyInfo` object.

Table 99: `LoginWarmStandbyInfo` Methods

7.3.6 Login Post Message Use

OMM consumer applications can encode and send data for any item via Post messages on the item's login stream. This is known as ***off-stream posting*** because items are posted without using that item's dedicated stream. Posting an item on its own dedicated stream is referred to as ***on-stream posting***. When an application is off-stream posting, `msgKey` information is required in the `PostMsg`.

For more details on posting, refer to the *Transport API Java Developers Guide*.

7.3.7 Login Ack Message Use

OMM Provider applications encode and send Ack messages to acknowledge the receipt of Post messages. An Ack message is used whenever a consumer posts and asks for acknowledgments.

For more details on posting, see the *Transport API Java Developers Guide*.

7.3.8 Login Attributes

On a Login Request or Login Refresh message, the `LoginAttrib` can send additional authentication information and user preferences between components.

7.3.8.1 Login Attrib Members

The following table lists the elements available on a `LoginAttrib`.

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional login attribute members. For details, refer to Table 102: `LoginAttribFlags`.
applicationId	**Optional**. Indicates the application ID. If present, a `flags` value of **LoginAttribFlags.HAS_APPLICATION_ID** should be specified. • If populated in a login request, `applicationId` should be set to the DACS `applicationId`. If the server authenticates with DACS, the consumer application may be required to pass in a valid application id. If initializing `LoginRequest` using `initDefaultRequest`, an `applicationId` of **256** will be used. • If populated in a login refresh, `applicationId` should match the `applicationId` used in the login request.
applicationName	**Optional**. Indicates the application name. If present, a `flags` value of **LoginAttribFlags.HAS_APPLICATION_NAME** should be specified. • If populated in a login request, the `applicationName` identifies the OMM consumer or OMM non-interactive provider. If initializing `LoginRequest` using `initDefaultRequest`, the `applicationName` is set to **upa**. • If populated in a login refresh, the `applicationName` identifies the OMM provider.

MEMBER	DESCRIPTION
position	**Optional**. Indicates the DACS position. If present, a `flags` value of **LoginAttribFlags.HAS_POSITION** should be specified. • When populated in a login request, `position` should match the `position` contained in the login request and the DACS `position` (if using DACS). If the server is authenticating with DACS, the consumer application might be required to pass in a valid position. If initializing `LoginRequest` using `initDefaultRequest`, the IP address of the system on which the application runs will be used. • When populated in a login refresh, this should match the `position` contained in the login request
providePermissionProfile	**Optional**. Indicates whether the consumer desires the permission profile. An application can use the permission profile to perform proxy permissioning. If present, a `flags` value of **LoginAttribFlags.HAS_PROVIDE_PERM_PROFILE** should be specified. • **1**: Indicates that the consumer want the permission profile. If absent, a default value of 1 is assumed. • **0**: Indicates the consumer does not want the permission profile.
providePermissionExpressions	**Optional**. Indicates whether the consumer wants permission expression information. Permission expressions allow for items to be proxy-permissioned by a consumer via content-based entitlements. If present, a `flags` value of **LoginAttribFlags.HAS_PROVIDE_PERM_EXPR** should be specified. • **1**: Requests that permission expression information be sent with responses. If absent, a default value of **1** is assumed. • **0**: Indicates that the consumer does not want permission expression information.
singleOpen	**Optional**. Indicates which application the consumer wants to drive stream recovery. If present, a `flags` value of **LoginAttribFlags.HAS_SINGLE_OPEN** should be specified. • **1**: Indicates that the consumer application wants the provider to drive stream recovery. If absent, a default value of **1** is assumed. • **0**: Indicates that the consumer application drives stream recovery.
allowSuspectData	**Optional**. Indicates how the consumer application wants to handle suspect data. If present, `flags` value of **LoginAttribFlags.HAS_ALLOW_SUSPECT_DATA** should be specified. • **1**: Indicates that the consumer application allows for suspect `streamState` information. If absent, a default value of **1** is assumed. • **0**: Indicates that the consumer application wants suspect data to cause the stream to close with a **StreamStates.CLOSED_RECOVER** state.

Table 100: `LoginAttrib` Members

7.3.8.2 Login Attrib Methods

METHOD NAME	DESCRIPTION
clear	Clears a `LoginAttrib` object for object reuse.
copy	Performs a deep copy of a `LoginAttrib` object.

Table 101: `LoginAttrib` Methods

7.3.8.3 Login Attrib Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_ALLOW_SUSPECT_DATA	Indicates the presence of `allowSuspectData`. If absent, a value of **1** is assumed.
HAS_APPLICATION_ID	Indicates the presence of `applicationId`.

FLAG ENUMERATION	MEANING
HAS_APPLICATION_NAME	Indicates the presence of `applicationName`.
HAS_POSITION	Indicates the presence of `position`.
HAS_PROVIDE_PERM_EXPR	Indicates the presence of `providePermissionExpressions`. If absent, a value of **1** is assumed.
HAS_PROVIDE_PERM_PROFILE	Indicates the presence of `providePermissionProfile`. If absent, a value of **1** is assumed.
HAS_SINGLE_OPEN	Indicates the presence of `singleOpen`. If absent, a value of **1** is assumed.

Table 102: `LoginAttribFlags`

7.3.9 Login Message

`LoginMsg` is the base interface for all Login messages. It is provided for use with general login-specific functionality. The following table summarizes different login messages.

INTERFACE	DESCRIPTION
`LoginRequest`	RDM Login Request.
`LoginRefresh`	RDM Login Refresh.
`LoginStatus`	RDM Login Status.
`LoginClose`	RDM Login Close.
`LoginConsumerConnectionStatus`	RDM Login Consumer Connection Status.

Table 103: `LoginMsg` Interfaces

7.3.10 Login Message Utility Methods

FUNCTION NAME	DESCRIPTION
clear	Clears a `LoginMsg` object for reuse.
copy	Performs a deep copy of a `LoginMsg` object.

Table 104: `LoginMsg` Utility Methods

7.3.11 Login Encoding and Decoding

7.3.11.1 Login Encoding and Decoding Methods

FUNCTION NAME	DESCRIPTION
encode	Encodes a login message. This method takes the **EncodeIterator** as a parameter into which the encoded content is populated. Each login subinterface overrides this method to encode specific login message.
decode	Decodes a login message. The decoded message may refer to encoded data from the original `message`. If you want to store the message, use the appropriate copy method for the decoded message to create a full copy. Each login subinterface overrides this method to encode specific login message.

Table 105: Login Encoding and Decoding Methods

7.3.11.2 Encoding a Login Request

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
LoginRequest loginRequest = (LoginRequest)LoginMsgFactory.createMsg();

/* Clear the Login Request object. */
loginRequest.clear();

/* Set login message type - required as object created by LoginMsgFactory.createMsg() is generic Login object.*/
loginRequest.rdmMsgType(LoginMsgType.REQUEST);

/* Set stream id. */
loginRequest.streamId(streamId);

/* Set flags indicating presence of optional members. */
loginRequest.applyHasAttrib();

/* Set UserName. */
loginRequest.userName().data("username");

/* Set ApplicationName */
loginRequest.attrib().applyHasApplicationName();
loginRequest.attrib().applicationName().data("upa");

/* Set ApplicationId */
loginRequest.attrib().applyHasApplicationId();
loginRequest.attrib().applicationId().data("256");

/* Set Position */
loginRequest.attrib().applyHasPosition();
loginRequest.attrib().position().data("127.0.0.1/net");

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
encodeIter.clear();
ret = encodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Encode the message. */
ret = loginRequest.encode(encodeIter);
```

Code Example 16: Login Request Encoding Example

7.3.11.3 Decoding a Login Request

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
LoginRequest loginRequest = (LoginRequest)LoginMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();
ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS &&
    msg.domainType() == DomainTypes.LOGIN && msg.msgClass() ==  MsgClasses.REQUEST)
{
```

```
    loginRequest.clear();
    loginRequest.rdmMsgType(LoginMsgType.REQUEST);

    ret = loginRequest.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {
        /* Print username. */
        printf("Username: “ + loginRequest.userName());

        if (loginRequest.checkHasAttrib())
        {
            LoginAttrib attrib = loginRequest.attrib();

            /* Print ApplicationName if present. */
            if(attrib.checkHasApplicationName())
                System.out.println(“ApplicationName: " + attrib.applicationName().toString());

            /* Print ApplicationId if present. */
            if(attrib.checkHasApplicationId())
                System.out.println(“ApplicationId: " + attrib.applicationId().toString());

            /* Print Position if present. */
            if(attrib.checkHasPosition())
                System.out.println(“Position: " + attrib.position().toString());
        }
    }
}
```

Code Example 17: Login Request Decoding Example

7.3.11.4 Encoding a Login Refresh

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
LoginRefresh loginRefresh = (LoginRefresh)LoginMsgFactory.createMsg();

/* Clear the Login Refresh object. */
loginRefresh.clear();

/* Set login message type - required as object created by LoginMsgFactory.createMsg() is generic Login object.*/
encodeIter.rdmMsgType(LoginMsgType.REFRESH);

/* Set stream id. */
loginRefresh.streamId(streamId);

/* Set flags indicating presence of optional members. */
loginRefresh.applyHasAttrib();
loginRefresh.applyHasUserName();

/* Set UserName. */
loginRefresh.userName().data("username");

/* Set ApplicationName */
loginRefresh.attrib().applyHasApplicationName();
loginRefresh.attrib().applicationName().data("upa");

/* Set ApplicationId */
```

```
loginRefresh.attrib().applyHasApplicationId();
loginRefresh.attrib().applicationId().data("256");

/* Set Position */
loginRefresh.attrib().applyHasPosition();
loginRefresh.attrib().position().data("127.0.0.1/net");

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
encodeIter.clear();
ret = encodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Encode the message. */
ret = loginRefresh.encode(encodeIter);
```

Code Example 18: Login Refresh Encoding Example

7.3.11.5 Decoding a Login Refresh

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
LoginRefresh loginRefresh = (LoginRefresh)LoginMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();
ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS &&
    msg.domainType() == DomainTypes.LOGIN && msg.msgClass() ==  MsgClasses.REFRESH)
{
    loginRefresh.clear();
    loginRefresh.rdmMsgType(LoginMsgType.REFRESH);

    ret = loginRefresh.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {
        /* Print username. */
        if(loginRefresh.checkHasUserName())
            printf("Username: " + loginRefresh.userName().toString());

        if (loginRefresh.checkHasAttrib())
        {
            LoginAttrib attrib = loginRefresh.attrib();

            /* Print ApplicationName if present. */
            if(attrib.checkHasApplicationName())
                System.out.println("ApplicationName: " + attrib.applicationName().toString());

            /* Print ApplicationId if present. */
            if(attrib.checkHasApplicationId())
                System.out.println("ApplicationId: " + attrib.applicationId().toString());

            /* Print Position if present. */
            if(attrib.checkHasPosition())
                System.out.println("Position: " + attrib.position().toString());
```

```
        }
    }
}
```

Code Example 19: Login Refresh Decoding Example

7.4 RDM Source Directory Domain

The Source Directory domain model conveys:

- Information about all available services and their capabilities, including the domain types supported within a service, the service's state, the QoS, and any item group information associated with the service. Each service is associated with a unique `serviceId`.
- Status information associated with item groups. This allows a single message to change the state of all associated items, avoiding the need to send a status message for each individual item. The consumer is responsible for applying any changes to its open items. For details, refer to Section 7.4.10: Directory Service Group State.
- Source Mirroring information between an ADH and OMM interactive provider applications exchanged via a specifically-formatted generic message as described in Section 7.4.6.

7.4.1 Directory Request

A ***Directory Request*** message is encoded and sent by OMM Consumer applications. This message is used to request information from an OMM Provider about available services. A consumer may request information about all services by omitting the `serviceId` member, or request information about a specific service by setting it to the ID of the desired service.

The `DirectoryRequest` represents all members of a directory request message and is easily used in OMM applications that leverage RDM.

7.4.1.1 Directory Request Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Directory message type. For a directory request, send **DirectoryMsgType.REQUEST**.
flags	**Required**. Indicates whether optional directory request members are present. For details, refer to Table 107: `DirectoryRequestFlags`.
serviceId	**Optional**. Indicates the ID of the service for which the consumer wants information. If present, a `flags` value of **DirectoryRequestFlags.HAS_SERVICE_ID** should be specified. • If absent, this indicates the consumer wants information about all available services. • If present, this indicates the consumer wants information only about the specified service.
filter	**Required**. Indicates the service information in which the consumer is interested. Available flags are: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP == 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 0x20** In most cases, you should set the **Directory.ServiceFilterFlags.INFO, Directory.ServiceFilterFlags.STATE** and **Directory.ServiceFilterFlags.GROUP**.

Table 106: `DirectoryRequest` Members

7.4.1.2 Directory Request Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_SERVICE_ID	Indicates the presence of `serviceId`.
STREAMING	Indicates that the consumer wants to receive updates about directory information after the initial refresh.

Table 107: `DirectoryRequestFlags`

7.4.1.3 Directory Request Methods

METHOD NAME	DESCRIPTION
clear	Clears a `DirectoryRequest` object for reuse.
copy	Performs a deep copy of a `DirectoryRequest` object.

Table 108: `DirectoryRequest` Methods

7.4.2 Directory Refresh

A ***Directory Refresh*** message is encoded and sent by OMM Provider and OMM non-interactive provider applications. This message can provide information about the services supported by the provider application.

The `DirectoryRefresh` represents all members of a directory refresh message and is easily used in OMM applications that leverage RDM.

7.4.2.1 Directory Refresh Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Directory message type. For a directory refresh, send **DirectoryMsgType.REFRESH**.
flags	**Required**. Indicates the presence of optional directory refresh members. See Table 110.
state	**Required**. Indicates stream and data state information. For information about `State`, refer to the *Transport API Java Developers Guide*..
filter	**Required**. Indicates the information being provided about supported services. This should match the `filter` of the consumer's `DirectoryRequest`. Available flags are: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP == 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 0x20**
serviceList	**Optional**. Contains a list of information about available services.
serviceId	**Optional**. If present, a `flags` value of **DirectoryRefreshFlags.HAS_SERVICE_ID** should be specified, which should match the `serviceId` of the consumer's `DirectoryRequest`.
sequenceNumber	**Optional**. Specified by the user, `sequenceNumber` is an item-level sequence number that the application can use to sequence messages in the stream. If present, a `flags` value of **DirectoryRefreshFlags.HAS_SEQ_NUM** should be specified.

Table 109: `DirectoryRefresh` Members

7.4.2.2 Directory Refresh Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_SERVICE_ID	Indicates the presence of `serviceId`.
SOLICITED	If this flag is present, it indicates that the login refresh is solicited (e.g., it is in response to a request). If the flag is not present, this refresh is unsolicited.
HAS_SEQ_NUM	Indicates the presence of `sequenceNumber`.
CLEAR_CACHE	Indicates that stored payload information associated with the login stream should be cleared. This might happen if data is known to be invalid.

Table 110: `DirectoryRefreshFlags`

7.4.3 Directory Update

A ***Directory Update*** message is encoded and sent by OMM Provider and OMM non-interactive provider applications. This message may be used to provide about new or removed services, or changes to existing services.

The `DirectoryUpdate` represents all members of a directory update message and allows for simplified use in OMM applications that leverage RDM.

7.4.3.1 Directory Update Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**.Directory message type. For a directory update, send **DirectoryMsgType.UPDATE**.
flags	**Required**. Indicates the presence of optional directory update members. See Table 112 for details.
filter	**Optional**. Indicates what information is provided about supported services. This should match the `filter` of the consumer's `DirectoryRequest`. If present, a `flags` value of **DirectoryUpdateFlags.HAS_FILTER** should be specified. Available flags are: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP == 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 0x20**
serviceList	**Optional**. Contains a list of information about available services.
serviceId	**Optional**. Must match the `serviceId` in the consumer's `DirectoryRequest`. If present, a `flags` value of **DirectoryUpdateFlags.HAS_SERVICE_ID** should be specified.
sequenceNumber	**Optional**. Specified by the user, `sequenceNumber` is an item-level sequence number that the application can use to sequence messages in the stream. If present, a `flags` value of **DirectoryUpdateFlags.HAS_SEQ_NUM** should be specified.

Table 111: `DirectoryUpdate` Members

7.4.3.2 Directory Update Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_SERVICE_ID	Indicates the presence of `serviceId`.
HAS_FILTER	Indicates the presence of `filter`.
HAS_SEQ_NUM	Indicates the presence of `sequenceNumber`.

Table 112: `DirectoryUpdateFlags`

7.4.4 Directory Status

OMM Provider and OMM non-interactive provider applications use the ***Directory Status*** message to convey state information associated with the directory stream. Such state information can indicate that a directory stream cannot be established or to inform a consumer of a state change associated with an open directory stream. An application can also use the Directory Status message to close an existing directory stream.

The `DirectoryStatus` represents all members of a directory status message and allows for simplified use in OMM applications that leverage RDM.

7.4.4.1 Directory Status Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Directory message type. For a directory status, send **DirectoryMsgType.STATUS**.
flags	**Required**. Indicates the presence of optional directory status members. See Table 114 for details.
filter	**Optional**. If present, a `flags` value of **DirectoryStatusFlags.HAS_FILTER** should be specified. Indicates the information being provided about supported services and should match the `filter` of the consumer's `DirectoryRequest`. Available flags are: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP == 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 0x10**
state	**Optional**. If present, a flags value of **DirectoryStatusFlags.HAS_STATE** should be specified. Indicates the state of the directory stream. For more information on `State`, refer to the *Transport API Java Developers Guide*.
serviceId	**Optional**. If present, a `flags` value of **DirectoryStatusFlags.HAS_SERVICE_ID** should be specified. `serviceId` should match the `serviceId` included in the consumer's `DirectoryRequest`.

Table 113: `DirectoryStatus` Members

7.4.4.2 Directory Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_STATE	Indicates the presence of `state`. If absent, any previously conveyed state should continue to apply.
HAS_FILTER	Indicates the presence of `filter`.

FLAG ENUMERATION	MEANING
HAS_SERVICE_ID	Indicates the presence of `serviceId`.

Table 114: `DirectoryStatus` Flags

7.4.4.3 Directory Status Methods

METHOD NAME	DESCRIPTION
clear	Clears a `DirectoryStatus` object for reuse.
copy	Performs a deep copy of a `DirectoryStatus` object.

Table 115: `DirectoryStatus` Methods

7.4.5 Directory Close

A ***Directory Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to close an open directory stream. A provider can close the directory stream via a Directory Status message; for details, refer to Section 7.4.4.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Directory message type. For a directory close, set **DirectoryMsgType.CLOSE**.

Table 116: `DirectoryClose` Members

7.4.6 Directory Consumer Status

The ***Directory Consumer Status*** is sent by an OMM consumer application to inform a service of how the consumer application is used in ***Source Mirroring***. This message is primarily informational.

The `DirectoryConsumerStatus` relies on the `GenericMsg` and represents all members necessary for applications that leverage RDM.

7.4.6.1 Directory Consumer Status Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Directory message type. For a directory consumer status, set **DirectoryMsgType.CONSUMER_STATUS**.
consumerServiceStatusList	**Optional**. Contains a list of `ConsumerStatusService` objects.

Table 117: `DirectoryConsumerStatus` Members

7.4.6.2 Directory Consumer Status Service Members

MEMBER	DESCRIPTION
serviceId	**Required**. Indicates the service associated with this status.
action	**Required**. Indicates how a cache of Source Mirroring content should apply this information. For information on `MapEntryActions`, refer to the *Transport API Java Developers Guide*.

MEMBER	DESCRIPTION
sourceMirroringMode	**Required**. Indicates how the consumer is using the service. Available values are: • **Directory.SourceMirroringMode.ACTIVE_NO_STANDBY = 0,** • **Directory.SourceMirroringMode.ACTIVE_WITH_STANDBY = 1,** • **Directory.SourceMirroringMode.STANDBY = 2**

Table 118: `ConsumerStatusService` Members

7.4.7 Directory Service

A `Service` object conveys information about a service. A list of `Services` forms the `serviceList` member of the `DirectoryRefresh` and `DirectoryUpdate` messages.

The members of a Service represent the different filters used to categorize service information.

7.4.7.1 Service Members

MEMBER	DESCRIPTION
Flags	**Required**. Indicates the presence of optional service members. For details, refer to Table 120.
action	**Required**. Indicates how a cache of the service should apply this information. See the *Transport API Java Developers Guide* for information on `MapEntryActions`.
serviceId	**Required**. Indicates the service associated with this `Service`.
info	**Optional**. Contains information related to the Source Directory Info Filter. If present, a `flags` value of **ServiceFlags.HAS_INFO** should be specified.
state	**Optional**. Contains information related to the Source Directory State Filter. If present, a `flags` value of **ServiceFlags.HAS_STATE** should be specified.
groupStateList	**Optional**. Contains a list of elements indicating changes to item groups and represents the Source Directory Group filter.
load	**Optional**. Contains information about the service's operating workload and represents the Source Directory Load Filter. If present, a `flags` value of **ServiceFlags.HAS_LOAD** should be specified.
data	**Optional**. Contains data that applies to the items requested from the service and represents the Source Directory Data Filter. If present, a `flags` value of **ServiceFlags.HAS_DATA** should be specified.
linkInfo	**Optional**. Contains information about upstream sources that provide data to this service and represents the Source Directory Link Filter. If present, `flags` value of **ServiceFlags.HAS_LINK** should be specified.

Table 119: `Service` Members

7.4.7.2 Service Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_INFO	Indicates the presence of `info`.
HAS_STATE	Indicates the presence of `state`.
HAS_LOAD	Indicates the presence of `load`.
HAS_DATA	Indicates the presence of `data`.

FLAG ENUMERATION	MEANING
HAS_LINK	Indicates the presence of `linkInfo`.

Table 120: `ServiceFlags`

7.4.7.3 Service Methods

METHOD NAME	DESCRIPTION
clear	Clears a `Service` object. Useful for object reuse.
copy	Performs a deep copy of a `Service` object.

Table 121: `Service` Methods

7.4.8 Directory Service Info Filter

A `ServiceInfo` object conveys information that identifies the service and the content it provides. The `ServiceInfo` object represents the Source Directory Info filter.

More information about the Info filter is available in the *Transport API Java RDM Usage Guide.*

7.4.8.1 Service Info Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional service info members. For details, refer to Table 123: `ServiceInfoFlag`.
action	**Required**. Indicates how a cache of the service information should apply this information For information on `FilterEntryActions`, refer to the *Transport API Java Developers Guide*.
serviceName	**Required**. Indicates the name of the service.
vendor	**Optional**. Identifies the vendor of the data. If present, a `flags` value of **ServiceInfoFlags.HAS_VENDOR** should be specified.
isSource	**Optional**. Indicates whether the service is provided directly by a source or represents a group of sources. If present, a `flags` value of **ServiceInfoFlags.HAS_IS_SOURCE** should be specified. Available values are: • **1**: The service is provided directly by a source • **0**: The service represents a group of sources. If absent, a value of **0** is assumed.
dictionariesProvidedCount	**Optional**. Indicates the number of elements present in `dictionariesProvided`. If present, a `flags` value of **ServiceInfoFlags.HAS_DICTS_PROVIDED** should be specified.
dictionariesProvidedList	**Optional**. Contains a list of elements that identify dictionaries which can be requested from this service. If present, a `flags` value of **ServiceInfoFlags.HAS_DICTS_PROVIDED** and the `dictionariesProvidedCount` should be specified.
dictionariesUsedCount	**Optional**. Indicates the number of elements present in `dictionariesUsed`. If present, a `flags` value of **ServiceInfoFlags.HAS_DICTS_USED** should be specified.
dictionariesUsedList	**Optional**. Contains a list of elements that identify dictionaries used to decode data from this service. If present, a `flags` value of **ServiceInfoFlags.HAS_DICTS_USED** and `dictionariesUsedCount` should be specified.

MEMBER	DESCRIPTION
qosList	**Optional**. Specifies a list of elements that identify the available Qualities of Service. If present, a `flags` value of **ServiceInfoFlags.HAS_QOS** should be specified.
itemList	**Optional**. Specifies a name that can be requested on the **DomainTypes.SYMBOL_LIST** domain to get a list of all items available from this service. If present, a `flags` value of **ServiceInfoFlags.HAS_ITEM_LIST** should be specified.
supportsQosRange	**Optional**. Indicates whether this service supports specifying a range of Qualities of Service (QoS) when requesting an item. If present, `flags` value of **ServiceInfoFlags.HAS_SUPPORT_QOS_RANGE** should be specified. Available values are: • **1**: QoS Range requests are supported. • **0**: QoS Range requests are not supported. If absent, a value of **0** is assumed. For information, see the `qos` and `worstQos` members of the `RequestMsg` in the *Transport API Java Developers Guide*.
supportsOutOfBandSnapshots	**Optional**. Indicates whether this service supports making snapshot requests even when the **OpenLimit** is reached. If present, a `flags` value of **ServiceInfoFlags.HAS_SUPPORT_OOB_SNAPSHOTS** should be specified. Available values are: • **1**: QoS Range requests are allowed. • **0**: QoS Range requests are not allowed. If absent, a value of **1** is assumed.
acceptingConsumerStatus	**Optional**. Indicates whether this service supports accepting `DirectoryConsumerStatus` messages for Source Mirroring. If present, a `flags` value of **ServiceInfoFlags.HAS_ACCEPTING_CONS_STATUS** should be specified. Available values are: • **1**: The serivce will accept Consumer Status messages. If absent, a value of **1** is assumed. • **0**: The serivce will not accept Consumer Status messages.

Table 122: `ServiceInfo` Members

7.4.8.2 Service Info Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_VENDOR	Indicates the presence of `vendor`.
HAS_IS_SOURCE	Indicates the presence of `isSource`.
HAS_DICTS_PROVIDED	Indicates the presence of `dictionariesProvidedList` and `dictionariesProvidedCount`.
HAS_DICTS_USED	Indicates the presence of `dictionariesUsedList` and `dictionariesUsedCount`.
HAS_QOS	Indicates the presence of `qosList` and `qosCount`.
HAS_SUPPORT_QOS_RANGE	Indicates the presence of `supportsQosRange`.
HAS_ITEM_LIST	Indicates the presence of `itemList`.
HAS_SUPPORT_OOB_SNAPSHOTS	Indicates the presence of `supportsOutOfBandSnapshots`.

FLAG ENUMERATION	MEANING
HAS_ACCEPTING_CONS_STATUS	Indicates the presence of `acceptingConsumerStatus`.

Table 123: `ServiceInfoFlags`

7.4.8.3 Service Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceInfo` object for reuse.
copy	Performs a deep copy of a `ServiceInfo` object.

Table 124: `ServiceInfo` Methods

7.4.9 Directory Service State Filter

A `ServiceState` object conveys information about the service's current state. It represents the Source Directory State filter. For more information about the State filter, refer to the *Transport API RDM Usage Guide.*

7.4.9.1 Service State Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional service info members. For details, refer to Table 126: `ServiceStateFlags`.
action	**Required**. Indicates how a cache of the service state should apply this information. For information on `FilterEntryActions`, refer to the *Transport API Java Developers Guide*.
serviceState	**Required**. Indicates whether the original provider of the data can respond to new requests. Requests can still be made if so indicated by `acceptingRequests`. Available values are: • **1**: The original provider of the data is available. • **0**: The original provider of the data is not currently available.
acceptingRequests	**Optional**. Indicates whether the immediate provider (to which the consumer is directly connected)can handle the request. If present, a `flags` value of **ServiceStateFlags.HAS_ACCEPTING_REQS** should be specified. Available values are: • **1**: The service accepts new requests. • **0**: The service currently does not accept new requests.
status	**Optional**. This status should be applied to all open items associated with this service. If present, a `flags` value of **ServiceStateFlags.HAS_STATUS** should be specified.

Table 125: `ServiceState` Members

7.4.9.2 Service State Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_ACCEPTING_REQS	Indicates the presence of `acceptingRequests`.
HAS_STATUS	Indicates the presence of `status`.

Table 126: `ServiceStateFlags`

7.4.9.3 Service State Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceState` object for reuse.
copy	Performs a deep copy of a `ServiceState` object.

Table 127: `ServiceState` Methods

7.4.10 Directory Service Group State Filter

A `ServiceGroupState` object conveys status and name changes for an item group. It represents the Source Directory Group filter. For more information about the Group State filter, refer to the *Transport API Java RDM Usage Guide.*

7.4.10.1 Service Group State Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional service info members. For details, refer to Table 129: `ServiceGroupStateFlags`.
action	**Required**. Indicates how a cache of the service group state should apply this information. For more information on `FilterEntryActions`, refer to the *Transport API Java Developers Guide*.
group	**Required**. Identifies the name of the item group being changed.
mergedToGroup	**Optional**. Indicates the new group name to which items with the specified group name (`group`) should be changed. If present, a `flags` value of **ServiceGroupFlags.HAS_MERGED_TO_GROUP** should be specified.
status	**Optional**. This status is applied to all open items associated with the specified group (`group`). If present, a `flags` value of **ServiceGroupFlags.HAS_STATUS** should be specified.

Table 128: `ServiceGroupState` Members

7.4.10.2 Service Group State Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_MERGED_TO_GROUP	Indicates the presence of `mergedToGroup`.
HAS_STATUS	Indicates the presence of `status`.

Table 129: `ServiceGroupStateFlags`

7.4.10.3 Service Group State Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceGroupState` object for reuse.
Copy	Performs a deep copy of a `ServiceGroupState` object.

Table 130: `ServiceGroupState` Methods

7.4.11 Directory Service Load Filter

A `ServiceLoad` object is used to convey the workload of a service. It represents the Source Directory Load filter. More information about the Service Load filter is available in the *Transport API Java RDM Usage Guide.*

7.4.11.1 Service Load Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional service info members. For details, refer to Table 132.
action	**Required**. Indicates how a cache of the service load should apply this information. For information on `FilterEntryActions`, refer to the *Transport API Java Developers Guide*.
openLimit	**Optional**. Specifies the maximum number of streaming requests that the service allows. If present, a `flags` value of **ServiceLoadFlags.HAS_OPEN_LIMIT** should be specified.
openWindow	**Optional**. Specifies the maximum number of outstanding requests (ones awaiting a refresh) that the service allows. If present, a `flags` value of **ServiceLoadFlags.HAS_OPEN_WINDOW** should be specified.
loadFactor	**Optional**. Indicates the current workload on the source that provides data. A higher load factor indicates a higher workload. If present, a `flags` value of **ServiceLoadFlags.HAS_LOAD_FACTOR** should be specified. For more information, refer to the *Transport API Java RDM Usage Guide.*

Table 131: `ServiceLoad` Members

7.4.11.2 Service Load Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_OPEN_LIMIT	Indicates the presence of `openLimit`.
HAS_OPEN_WINDOW	Indicates the presence of `openWindow`.
HAS_LOAD_FACTOR	Indicates the presence of `loadFactor`.

Table 132: `ServiceLoadFlags`

7.4.11.3 Service Load Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceLoad` object for reuse.
copy	Performs a deep copy of a `ServiceLoad` object.

Table 133: `ServiceLoad` Methods

7.4.12 Directory Service Data Filter

A `ServiceData` object conveys data to apply to all items of a service. It represents the Source Directory Data filter.

For more information about the Data filter, refer to the *Transport API Java RDM Usage Guide.*

7.4.12.1 Service Data Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional service data members. For details, refer to Table 135.
action	**Required**. Indicates how a cache of the service data should apply this information. For information on `FilterEntryActions`, refer to the *Transport API Java Developers Guide*.
type	**Optional**. Indicates the type of content present in `data`. If present, a `flags` value of **ServiceDataFlags.HAS_DATA** should be specified. Available enumerations are: • **Directory.DataTypes.TIME = 1** • **Directory.DataTypes.ALERT = 2** • **Directory.DataTypes.HEADLINE = 3** • **Directory.DataTypes.STATUS = 4**
dataType	**Optional**. Specifies the `DataType` of the data. If present, a `flags` value of **ServiceDataFlags.HAS_DATA** should be specified. For information on `DataTypes`, refer to the *Transport API Java Developers Guide.*
data	**Optional**. Contains the encoded `Buffer` representing the data. The type of the data is given by `dataType`. If present, a `flags` value of **ServiceDataFlags.HAS_DATA** should be specified.

Table 134: `ServiceData` Members

7.4.12.2 Service Load Data Enumeration Values

FLAG ENUMERATION	MEANING
HAS_DATA	Indicates the presence of `type`, `dataType`, and `data`.

Table 135: `ServiceDataFlags`

7.4.12.3 Service Data Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceData` object for reuse.
copy	Performs a deep copy of a `ServiceData` object.

Table 136: `ServiceData` Methods

7.4.13 Directory Service Link Info Filter

A `ServiceLinkInfo` object conveys information about upstream sources that form a service. It represents the Source Directory Link filter. The `ServiceLinkInfo` object contains an list of `ServiceLink` objects that each represents an upstream source.

For more information about the Service Link filter content, refer to the *Transport API Java RDM Usage Guide.*

7.4.13.1 Service Link Info Members

MEMBER	DESCRIPTION
action	**Required**. Indicates how a cache of the service link information should apply this information. For information on `FilterEntryActions`, refer to the *Transport API Java Developers Guide*.
linkList	**Optional**. Contains a list of `ServiceLink` objects, each representing a source.

Table 137: `ServiceLinkInfo` Members

7.4.13.2 Service Link Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceLinkInfo` object for reuse.
copy	Performs a deep copy of a `ServiceLinkInfo` object.

Table 138: `ServiceLinkInfo` Methods

7.4.14 Directory Service Link

A `ServiceLink` object conveys information about an upstream source. It represents an entry in the Source Directory Link filter and is used by the `linkList` member of the `ServiceLinkInfo` object.

For more information about the Service Link filter content, refer to the *Transport API Java RDM Usage Guide.*

7.4.14.1 Service Link Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional service link members. For details, refer to Table 140.
action	**Required**. Indicates how a cache of the service link should apply this information. For information on `MapEntryActions`, refer to the *Transport API Java Developers Guide*.
name	**Required**. Specifies the name of the source. Sources with identical names indicates that sources are load-balanced.
type	**Optional**. Specifies whether the source is interactive or broadcast. If present, a `flags` value of **ServiceLinkFlags.HAS_TYPE** should be specified. Available values are: • **Directory.LinkTypes.INTERACTIVE = 1** • **Directory.LinkTypes.BROADCAST = 2**
linkState	**Required**. Indicates whether the source is up or down. Available values are: • **Directory.LinkStates.DOWN = 0** • **Directory.LinkStates.UP = 1**

MEMBER	DESCRIPTION
linkCode	**Optional**. Indicates additional information about the status of a source. If present, a `flags` value of **ServiceLinkFlags.HAS_CODE** should be specified. Available values are: • **Directory.LinkCodes.NONE = 0** • **Directory.LinkCodes.OK = 1** • **Directory.LinkCodes.RECOVERY_STARTED = 2** • **Directory.LinkCodes.RECOVERY_COMPLETED = 3**
text	**Optional**. Further describes the status of a source. If present, a `flags` value of **ServiceLinkFlags.HAS_TEXT** should be specified.

Table 139: `ServiceLink` Members

7.4.14.2 Service Link Enumeration Values

FLAG ENUMERATION	MEANING
HAS_TYPE	Indicates the presence of `type`.
HAS_CODE	Indicates the presence of `code`.
HAS_TEXT	Indicates the presence of `text`.

Table 140: `ServiceLinkFlags`

7.4.14.3 Service Link Methods

METHOD NAME	DESCRIPTION
clear	Clears a `ServiceLink` object for reuse.
copy	Performs a deep copy of a `ServiceLink` object.

Table 141: `ServiceLink` Methods

7.4.15 Directory Message

`DirectoryMsg` is the general purpose base interface for all directory messages. Different directory messages are summarized in the following table.

INTERFACE	DESCRIPTION
`DirectoryRequest`	RDM Directory Request.
`DirectoryRefresh`	RDM Directory Refresh.
`DirectoryUpdate`	RDM Directory Update.
`DirectoryStatus`	RDM Directory Status.
`DirectoryClose`	RDM Directory Close.
`DirectoryConsumerConnectionStatus`	RDM Directory Consumer Status.

Table 142: `DirectoryMsg` Interfaces

7.4.16 Directory Message Utility Methods

FUNCTION NAME	DESCRIPTION
clear	Clears a `DirectoryMsg` object for reuse.
copy	Performs a deep copy of a `DirectoryMsg` object.

Table 143: `DirectoryMsg` Utility Methods

7.4.17 Directory Encoding and Decoding

7.4.17.1 Directory Encoding and Decoding Methods

FUNCTION NAME	DESCRIPTION
encode	Encodes a source directory message. This method takes the **EncodeIterator** as a parameter into which the encoded content is populated.
decode	Decodes a source directory message. The decoded message may refer to encoded data from the original message. If the message is to be stored for later use, use the copy method of the decoded message to create a full copy.

Table 144: Directory Encoding and Decoding Methods

7.4.17.2 Encoding a Source Directory Request

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DirectoryRequest directoryRequest = (DirectoryRequest)DirectoryMsgFactory.createMsg();

/* Clear the Directory Request object. */
directoryRequest.clear();

/* Set directory message type - required as object created by DirectoryMsgFactory.createMsg() is generic
       Directory object. */
directoryRequest.rdmMsgType(DirectoryMsgType.REQUEST);

/* Set stream id. */
directoryRequest.streamId(streamId);

/* Set flags indicating presence of optional members. */
directoryRequest.flags(DirectoryRequestFlags.HAS_SERVICE_ID | STREAMING);

/* Set Service ID. */
directoryRequest.serviceId(273);

/* Set service filter. */
directoryRequest.filter(Directory.ServiceFilterFlags.INFO | Directory.ServiceFilterFlags.STATE |
       Directory.ServiceFilterFlags.GROUP);

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
encodeIter.clear();
ret = encodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Encode the message. */
ret = directoryRequest.encode(encodeIter);
```

Code Example 20: Directory Request Encoding Example

7.4.17.3 Decoding a Source Directory Request

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DirectoryRequest directoryRequest = (DirectoryRequest)DirectoryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();

ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS &&
    msg.domainType() == DomainTypes.SOURCE && msg.msgClass() ==  MsgClasses.REQUEST)
{
    directoryRequest.clear();
    directoryRequest.rdmMsgType(DirectoryMsgType.REQUEST);

    ret = directoryRequest.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {

        /* Print if Info filter was requested. */
        if ((directoryRequest.filter() & Directory.ServiceFilterFlags.INFO) != 0)
            System.out.println("Info filter requested.");

        /* Print if State filter was requested. */
        if ((directoryRequest.filter() & Directory.ServiceFilterFlags.STATE) != 0)
            System.out.println("State filter requested.");

        /* Print if Group filter was requested. */
        if ((directoryRequest.filter() & Directory.ServiceFilterFlags.GROUP) != 0)
            System.out.println("Group filter requested.");

        /* Print service ID if present. */
        if (directoryRequest.checkHasServiceId())
            System.out.println("Service ID: “ + directoryRequest->serviceId);
    }
}
```

Code Example 21: Directory Request Decoding Example

7.4.17.4 Encoding a Source Directory Refresh

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DirectoryRefresh directoryRefresh = (DirectoryRefresh)DirectoryMsgFactory.createMsg();

/* Clear the Directory Refresh object. */
directoryRefresh.clear();
```

```
/* Set directory message type - required as object created by DirectoryMsgFactory.createMsg() is generic
       Directory object. */
directoryRefresh.rdmMsgType(DirectoryMsgType.REFRESH);

/* Set stream id. */
directoryRefresh.streamId(streamId);

/* Set flags for optional members */
directoryRefresh.applySolicited();

/* Set state. */
directoryRefresh.state().streamState(StreamStates.OPEN);
directoryRefresh.state().dataState(DataStates.OK);
directoryRefresh.state().code(StateCodes.NONE);

/* Set filter to say the Info, State, and Group filters are supported. */
directoryRefresh.filter(Directory.ServiceFilterFlags.INFO | Directory.ServiceFilterFlags.STATE |
Directory.ServiceFilterFlags.GROUP);


/* List of services to be used.
 * This example will show encoding of one service. Additional services
 * can be set up using the same method shown below. */
/*** Create Service ***/
Service service = CodecFactory.createService();

/*** Build Service MY_SERVICE. ***/
service.clear();

/* Set flags to indicate Info and State filter are present. */
service.flags(ServiceFlags.HAS_INFO | ServiceFlags.HAS_STATE);

/* Set action to indicate adding a new service. */
service.info().action(MapEntryActions.ADD);

/* Set flags to indicate optional members. */
service.info().flags(ServiceInfoFlags.HAS_VENDOR | ServiceInfoFlags.HAS_DICTS_PROVIDED | ServiceInfoFlags.
       HAS_DICTS_USED |  ServiceInfoFlags.HAS_QOS);

/* Set service name. */
service.info().serviceName().data("MY_SERVICE");

/* Set vendor name. */
service.info().vendor().data("Thomson Reuters");

/* Set capabilities list. */
service.info().capabilitiesList().add(DomainTypes.DICTIONARY);
service.info().capabilitiesList().add(DomainTypes.MARKET_PRICE);
service.info().capabilitiesList().add(DomainTypes.MARKET_BY_ORDER);


/* Set dictionaries provided. */
service.info().dictionariesProvidedList().add("RWFFld");
service.info().dictionariesProvidedList().add("RWFEnum");

/* Set dictionaries used. */
```

```
service.info().dictionariesUsedList ().add("RWFFld");
service.info().dictionariesUsedList ().add("RWFEnum");

/* Build QoS list. */
Qos qos1 = CodecFactory.createQos();
qos1.timeliness(QosTimeliness.REALTIME);
qos1.rate(QosRates.TICK_BY_TICK);
Qos qos2 = CodecFactory.createQos();
service.info().qosList().add(qos2);
qos2.timeliness(QosTimeliness.REALTIME);
qos2.rate(QosRates.JIT_CONFLATED);

/* Set QoS list. */
service.info().qosList().add(qos1);
service.info().qosList().add(qos2);

/*** Build Service State for MY_SERVICE ***/
service.state().flags(ServiceStateFlags.HAS_ACCEPTING_REQS);
service.state().serviceState(1);
service.state().acceptingRequests(1);

/*** Finish and encode. ***/

/* Set the list of services on the message.*/
directoryRefresh.serviceList().add(service);

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
encodeIter.clear();
ret = encodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Encode the message. */
ret = directoryRefresh.encode(encodeIter);
```

Code Example 22: Directory Refresh Encoding Example

7.4.17.5 Decoding a Source Directory Refresh

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DirectoryRefresh directoryRefresh = (DirectoryRefresh)DirectoryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();

ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS &&
    msg.domainType() == DomainTypes.SOURCE && msg.msgClass() ==  MsgClasses.REFRESH)
{
    directoryRefresh.clear();
    directoryRefresh.rdmMsgType(DirectoryMsgType.REFRESH);
```

```
ret = directoryRefresh.decode(decodeIter, msg);

if(ret == CodecReturnCodes.SUCCESS)
{
    /* Print serviceId if present. */
    if (directoryRefresh.checkHasServiceId())
        System.out.println("Service ID: " +  directoryRefresh.serviceId());

    /* Print information about each service present in the refresh. */
    for(Service service : directoryRefresh.serviceList())
    {
        /* Print Service Info if present */
        if (service.checkHasInfo())
        {
            ServiceInfo info = service.info();

            /* Print service name. */
            System.out.println("Service Name: " + info.serviceName().toString());

            /* Print vendor name if present.*/
            if (info.checkHasVendor())
                System.out.println("Vendor: " + info.vendor().toString());

            /* Print supported domains if present.*/
            for(long capability : info.capabilityList())
                System.out.println("Capability: " +  DomainTypes.toString(capability));

            /* Print dictionaries provided if present.*/
            if (info.checkHasDictionariesProvided())
            {
                for (String dictProv : info.dictionariesProvidedList())
                System.out.println("Dictionary Provided: " +  dictProv);
            }

            /* Print dictionaries used if present. */
            if (info.checkHasDictionariesUsed())
            {
                for (String dictUsed : info.dictionariesUsedList())
                    System.out.println("Dictionary Used: " +  dictUsed);
            }

            /* Print qualities of service supported if present. */
            if (info.checkHasQos())
            {
                for (Qos qos : info.qosList())
                    System.out.println ("QoS: " + qos.toString());
            }
        }

        if (service.checkHasState())
        {
            ServiceState state = service.state();
            System.out.println("Service state: " + state.serviceState());
            if(state.checkHasAcceptingRequests())
                System.out.println("Accepting Requests: " + state.acceptingRequests());
        }
```

```
        }
    }
}
```

Code Example 23: Directory Refresh Decoding Example

7.5 RDM Dictionary Domain

The Dictionary domain model conveys information needed to parse the published content. Dictionaries provide additional meta-data, such as data needed to decode the content of a `FieldEntry` or additional content related to its `fieldId`. For more information about the different types of dictionaries and their usage, refer to the *Transport API Java RDM Usage Guide*.

This domain's interface makes it easier to use existing utilities for encoding, decoding, and caching dictionary information. For more information on these utilities, refer to the *Transport API Java RDM Usage Guide*.

7.5.1 Dictionary Request

A ***Dictionary Request*** message is encoded and sent by OMM Consumer applications. This message requests a dictionary from a service.

The `DictionaryRequest` represents all members of a dictionary request message and is easily used in OMM applications that leverage RDM.

7.5.1.1 Dictionary Request Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Dictionary message type. For a dictionary request, send **DictionaryMsgType.REQUEST**.
flags	**Required**. Indicates the presence of optional dicitonary request members. See Table 146 for details.
serviceId	**Required**. Specifies the service from which to request the dictionary.
verbosity	**Required**. Indicates the amount of information desired from the dictionary. Available values are: • **Dictionary.VerbosityValues.INFO = 0x00**: Version information only. • **Dictionary.VerbosityValues.MINIMAL == 0x03**: Provides information needed for caching. • **Dictionary.VerbosityValues.NORMAL == 0x07**: Provides all information needed for decoding. • **Dictionary.VerbosityValues.VERBOSE == 0x0F**: Provides all information(including comments). Providers are not required to support the **MINIMAL** and **VERBOSE** filters.
dictionaryName	**Required**. Indicates the name of the requested dictionary.

Table 145: `DictionaryRequest` Members

7.5.1.2 Dictionary Request Flag Enumeration Values

FLAG ENUMERATION	MEANING
STREAMING	Indicates that the dictionary stream should remain open after the initial refresh. An open stream can listen for status messages that indicate changes to the dictionary version. For more information, refer to the *Transport API Java RDM Usage Guide*.

Table 146: `DictionaryRequestFlags` Flags

7.5.2 Dictionary Refresh

A ***Dictionary Refresh*** message is encoded and sent by OMM Provider applications. This message is used to send dictionary content in response to a request.

The `DictionaryRefresh` represents all members of a dictionary refresh message and allows for simplified use in OMM applications that leverage RDM.

7.5.2.1 Dictionary Refresh Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Dictionary message type. For a dictionary refresh, set to **DictionaryMsgType.REFRESH**.
flags	**Required**. Indicates the presence of optional dictionary refresh members. See Table 148 for details.
state	**Required**. Indicates the state of the login stream. Defaults to a `streamState` of **StreamStates.OPEN** and a `dataState` of **DataStates.OK**. For more information on `State`, refer to the *Transport API Java Developers Guide*.
dictionaryName	**Required**. Indicates the name of the dictionary being provided.
serviceId	**Required**. Indicates the service that provides the dictionary.
verbosity	**Required**. Indicates the amount of information desired from the dictionary. Available values are: • **Dictionary.VerbosityValues.INFO = 0x00**: Provides version information only • **Dictionary.VerbosityValues.MINIMAL = 0x03**: Provides information needed for caching • **Dictionary.VerbosityValues.NORMAL = 0x07**: Provides all information needed for decoding • **Dictionary.VerbosityValues.VERBOSE = 0x0F**: Provides all information(including comments) Providers do not need to support the **MINIMAL** and **VERBOSE** filters.
dictionaryType	**Required**. Indicates the type of dictionary provided. The dictionary encoder and decoder support the following types: • **Dictionary.Types.FIELD_DEFINITIONS = 1** • **Dictionary.Types.ENUM_TABLES = 2**
sequenceNumber	**Optional**. A user-specified, item-level sequence number that the application can use to sequence messages in this stream. If present, a `flags` value of **DictionaryRefreshFlags.HAS_SEQ_NUM** should be specified.
dictionary	**Required** when encoding. Points to a `DataDictionary` object that contains content to encode. For more information on the `DataDictionary`, see the *Transport API Java RDM Usage Guide*. **Not used** when decoding.
startFid	Maintains the state when encoding a dictionary across multiple messages. **Warning!** To ensure that all dictionary content is correctly encoded, the application should not modify this.
dataBody	When decoding, this points to the encoded data buffer with dictionary content. This buffer should be set on a `DecodeIterator` and passed to the appropriate decode method according to the `type`. **Not used** when encoding. The dictionary is retrieved from the `DataDictionary`.

Table 147: `DictionaryRefresh` Members

7.5.2.2 Dictionary Refresh Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_INFO	When decoding, Indicates the presence of `dictionaryType`. **Not used** when encoding. The encode method adds information to the encoded message when appropriate.
IS_COMPLETE	When decoding, if this flag is present, it indicates that this is the final fragment and that the consumer has received all content for this dictionary. **Not used** when encoding. The encode method adds information to the encoded message when appropriate.
SOLICITED	If this flag is present, it indicates that the directory refresh is solicited (i.e., it is in response to a request). If the flag is not present, this refresh is unsolicited.
HAS_SEQ_NUM	Indicates the presence of `sequenceNumber`.
CLEAR_CACHE	Clears any stored payload information associated with the dictionary stream. This might happen if some portion of data is invalid.

Table 148: `DictionaryRefreshFlags`

7.5.3 Dictionary Status

OMM Provider and OMM non-interactive provider applications use the ***Dictionary Status*** message to convey state information associated with the dictionary stream. Such state information can indicate that a dictionary stream cannot be established or to inform a consumer of a state change associated with an open dictionary stream. The Dictionary status message can also indicate that a new dictionary should be retrieved. The `DictionaryStatus` represents all members of a dictionary status message and is easily used in OMM applications that leverage RDM.

For more information on handling Dictionary versions, see the *Transport API Java RDM Usage Guide*.

7.5.3.1 Dictionary Status Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Dictionary message type. For a dictionary status, set to **DictionaryMsgType.STATUS**.
flags	**Required**. Indicates the presence of optional login status members. For details, refer to Table 150.
state	**Optional**. Indicates the state of the dictionary stream. If present, a `flags` value of **DictionaryStatusFlags.HAS_STATE** should be specified. For more information on `State`, refer to the *Transport API Java Developers Guide*.

Table 149: `DictionaryStatus` Members

7.5.3.2 Dictionary Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_STATE	Indicates the presence of `state`. If absent, any previously conveyed state continues to apply.

Table 150: `DictionaryStatusFlags`

7.5.4 Dictionary Close

A ***Dictionary Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to close an open dictionary stream. A provider can close the directory stream via a Dictionary Status message; for details, refer to Section 7.5.3.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Dictionary message type. For a dictionary close, set to **DictionaryMsgType.CLOSE**.

Table 151: `DictionaryClose` Members

7.5.5 Dictionary Messages

`DictionaryMsg` is the base interface for all Dictionary messages. It is provided for use with general dictionary-specific functionality.

MESSAGES	DESCRIPTION
`DictionaryRequest`	RDM Dictionary Request.
`DictionaryRefresh`	RDM Dictionary Refresh.
`DictionaryStatus`	RDM Dictionary Status.
`DictionaryClose`	RDM Dictionary Close.

Table 152: `DictionaryMsg` Interfaces

7.5.6 Dictionary Message: Utility Methods

FUNCTION NAME	DESCRIPTION
clear	Clears a `DictionaryMsg` object for reuse.
copy	Performs a deep copy of a `DictionaryMsg` object.

Table 153: `DictionaryMsg` Utility Methods

7.5.7 Dictionary Encoding and Decoding

FUNCTION NAME	DESCRIPTION
encode	Encodes a dictionary message. This method takes the **EncodeIterator** as a parameter into which the encoded content is populated.
decode	Decodes a dictionary message. The decoded message may refer to encoded data from the original message. If the message is to be stored for later use, use the copy method of the decoded message to create a full copy..

Table 154: Dictionary Encoding and Decoding Methods

7.5.8 Dictionary Encoding and Decoding

7.5.8.1 Encoding a Dictionary Request

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DictionaryRequest dictionaryRequest = (DictionaryRequest)DictionaryMsgFactory.createMsg();
```

```
/* Clear the Dictionary Request object. */
dictionaryRequest.clear();

/* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg() is generic
       Dictionary object. */
dictionaryRefresh.rdmMsgType(DictionaryMsgType.REQUEST);

/* Set stream id. */
dictionaryRefresh.streamId(streamId);

/* Set streaming flag. */
dictionaryRequest.applyStreaming();

/* Set serviceId. */
dictionaryRequest.serviceId(273);

/* Set verbosity. */
dictionaryRequest.verbosity(Dictionary.VerbosityValues.NORMAL);

/* Set dictionary name. */
dictionaryRequest.dictionaryName().data("RWFFld");

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
encodeIter.clear();
ret = encodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);
/* Encode the message. */
ret = dictionaryRequest.encode(encodeIter);
```

Code Example 24: Dictionary Request Encoding Example

7.5.8.2 Decoding a Dictionary Request

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DictionaryRequest dictionaryRequest = (DictionaryRequest)DictionaryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();
ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS &&
    msg.domainType() == DomainTypes.DICTIONARY && msg.msgClass() ==  MsgClasses.REQUEST)
{
    dictionaryRequest.clear();

    /* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg() is generic
       Dictionary object. */
    dictionaryRequest.rdmMsgType(DictionaryMsgType.REQUEST);

    ret = dictionaryRequest.decode(decodeIter, msg);
```

```
    if(ret == CodecReturnCodes.SUCCESS)
    {
        if(dictionaryRequest.checkStreaming())
            System.out.println("Request is streaming");

        /* Print serviceId. */
        System.out.println ("Service ID: " + dictionaryRequest.serviceId());

        /* Print verbosity. */
        System.out.println ("Verbosity: " + dictionaryRequest.verbosity());

        /* Print dictionary name. */
        System.out.println ("Dictionary Name: " + dictionaryRequest.dictionaryName().toString());
    }
}
```

Code Example 25: Dictionary Request Decoding Example

7.5.8.3 Encoding a Dictionary Refresh

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DictionaryRefresh dictionaryRefresh = (DictionaryRefresh)DictionaryMsgFactory.createMsg();

/* Clear the Dictionary Refresh object. */
dictionaryRefresh.clear();
dictionaryRefresh.rdmMsgType(DictionaryMsgType.REFRESH);

DataDictionary dataDictionary = CodecFactory.createDataDictionary();
dataDictionary.clear();

ret = dataDictionary.loadFieldDictionary("RDMFieldDictionary", errorText);

/* Clear the Dictionary Refresh object. */
dictionaryRefresh.clear();

/* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg() is generic
       Dictionary object. */
dictionaryRefresh.rdmMsgType(DictionaryMsgType.REFRESH);

/* Set stream id. */
dictionaryRefresh.streamId(streamId);

/* Set state fields to state object managed by dictionary refresh. */
dictionaryRefresh.state().streamState(StreamStates.OPEN);
dictionaryRefresh.state().dataState(DataStates.OK);
dictionaryRefresh.state().code(StateCodes.NONE);

/* Set flags. */
dictionaryRefresh.applySolicited();

/* Set dictionary name. */
dictionaryRefresh.dictionaryName().data("RWFFld");

/* Set dictionary type. */
dictionaryRefresh.dictionaryType(Dictionary.Types.FIELD_DEFINITIONS);
```

```
/* Set the dictionary. */
dictionaryRefresh.dictionary(dataDictionary);

/* Set serviceId. */
dictionaryRefresh.serviceId(273);

/* Set verbosity. */
dictionaryRefresh.verbosity(Dictionary.VerbosityValues.NORMAL);

do
{
    /* (Represents the application getting a new buffer to encode the message into.) */
    getNextEncodeBuffer(msgBuffer);


    /* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
    encodeIter.clear();
    ret = encodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

    /* Encode the message. This will return CodecReturnCodes.DICT_PART_ENCODED if it only a part
    * was encoded. We must keep encoding the message until CodecReturnCodes.SUCCESS is returned. */
    ret = dictionaryRefresh.encode(encodeIter);
} while (ret == CodecReturnCodes.DICT_PART_ENCODED);
```

Code Example 26: Dictionary Refresh Encoding Example

7.5.8.4 Decoding a Dictionary Refresh

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DictionaryRefresh dictionaryRefresh = (DictionaryRefresh)DictionaryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

DataDictionary dataDictionary = CodecFactory.createDataDictionary();
dataDictionary.clear();

int dictionaryTypeForThisStreamId = 0;

do
{
    /* (Represents the application getting the next buffer to decode.) */
    getNextEncodeBuffer(msgBuf);

    /* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
    decodeIter.clear();
    ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

    /* Decode the message to a Msg object. */
    sret = msg.decode(decodeIter);
    if (ret == CodecReturnCodes.SUCCESS &&
        msg.domainType() == DomainTypes.DICTIONARY && msg.msgClass() ==  MsgClasses.REFRESH)
    {
        dictionaryRefresh.clear();

        /* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg()
```

```
            is generic Dictionary object. */
        dictionaryRefresh.rdmMsgType(DictionaryMsgType.REFRESH);

        ret = dictionaryRefresh.decode(decodeIter, msg);

        if(ret == CodecReturnCodes.SUCCESS)
        {

            /* Print if request is streaming. */
            if (dictionaryRefresh.checkSolicited())
                System.out.println("Refresh is solicited.");

            /* Print info if present. If the dictionary is split into parts, this is normally only
                present on the first part. */
            if (dictionaryRefresh.checkHasInfo())
            {
                /* Remember the dictionary type for this stream since subsequent parts will not
                    indicate it. */
                dictionaryTypeForThisStreamId = pDictionaryRefresh.dictionaryType();

                /* Print version. */
                System.out.println("Version: “ + dictionaryRefresh.version.toString());

                /* Print dictionary ID. */
                System.out.println("Dictionary ID: " + dictionaryRefresh.dictionaryId());
            }

            /* Print serviceId. */
            System.out.println("Service ID: " + dictionaryRefresh.serviceId());

            /* Print verbosity. */
            System.out.println("Verbosity: " + dictionaryRefresh.verbosity());

            /* Print dictionary name. */
            System.out.println("Dictionary Name: " + dictionaryRefresh.dictionaryName().toString());

            if (dictionaryTypeForThisStreamId ==  Dictionary.Types.FIELD_DEFINITIONS)
            {
                /* Decode the dictionary content into the DataDictionary object. */
                decodeIter.clear();
                ret = decodeIter.setBufferAndRWFVersion(dictionaryRefresh.dataBody(),
                    channelMajorVersion, channelMinorVersion);

                ret = dataDictionary.decodeFieldDictionary(decodeIter,
                    Dictionary.VerbosityValues.NORMAL, errorText);
            }
        }
    }
} while(!(dictionarRefresh.checkRefreshComplete());
```

Code Example 27: Dictionary Refresh Decoding Example

7.6 RDM Queue Messages

The Queue Messaging domain model is a series of message constructs that you use to interact with a Queue Provider. The Queue Provider can help persist any content for which users want to have guaranteed delivery, and it can also help user's content reach destinations with which users cannot directly communicate.

7.6.1 Queue Data Message Persistence

When a queue messaging stream opens from a consumer to a Queue Provider, use of a persistence file can guarantee delivery of messages sent by an OMM consumer on that queue stream. The queue file will be named after the name of the queue stream (specified in the `QueueRequest` message that opened the stream). When the consumer submits `QueueData` messages, the consumer stores these messages in the persistence file in case the tunnel stream to the queue provider is lost and reconnected. As `QueueAck` messages are received from the queue provider, space in the persistence file is freed for additional messages. If at any time the application submits a `QueueData` message but the the persistence file has no more room for it, the application receives the **ReactorReturnCodes.PERSISTENCE_FULL** return code.

The `ClassOfService.guarantee.persistLocally` option (set when opening the tunnel stream) specifies whether to create and maintain persistence files. The location for storage of persistent files is specified by the `ClassOfService.guarantee.persistenceFilePath` option. For more information on class of service and its options, refer to Section 6.6.3.

Note: Thomson Reuters recommends that the `ClassOfService.guarantee.persistenceFilePath` be set to a local storage device.

If a particular queue stream is no longer needed, the user can delete the persistence file with that queue stream's name.

Warning! If you delete a persistence file that stores messages that were not successfully transmitted, the messages will be lost.

7.6.2 Queue Request

A ***Queue Request*** message is encoded and sent by OMM applications to a Queue Provider to open a user queue. By opening a queue with a `QueueRequest`, the user receives any content previously sent to and persisted on a Queue Provider. To send content to other users' queues, a user must first open their own queue.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the queue message type. For a queue request, send **QueueMsgType.REQUEST**
sourceName	**Required**. Indicates the name of the queue you want to open.

Table 155: `QueueRequest` Members

7.6.3 Queue Refresh

Queue Providers encode and send ***Queue Refresh messages to OMM applications to inform users about queue open*** requests and give state information pertaining to specific `QueueRequest` attempts.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the queue message type. For a queue refresh, use **QueueMsgType.REFRESH**
sourceName	**Required**. Indicates the name of the queue being opened. This should match the name specified in the corresponding `QueueRequest`.

MEMBER	DESCRIPTION
state	**Required**. Indicates the state of the queue. States of Open and Ok indicate the queue was successfully opened. Other state combinations indicate that an issue has occurred; the resultant code and text should provide information about the issue. For more information on `State`, refer to the *Transport API Java Developers Guide*.
queueDepth	Indicates the number of messages remaining in the queue for this stream.

Table 156: `QueueRefresh` Members

7.6.4 Queue Status

Queue providers encode and send queue status messages to OMM applications to convey state information about a user's queue.

7.6.4.1 Queue Status Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the queue message type. For a queue status, use **QueueMsgType.STATUS**
flags	**Required**. Indicates the presence of any optional queue status members. For details refer to Table 158.
state	**Required**. Indicates the current state of the queue. States of Open and Ok indicate the queue is in a good state. Other state combinations indicate that an issue has occurred; the resultant code and text should provide information about the issue. For more information on the `State` object, refer to the *Transport API Java Developers Guide*.

Table 157: `QueueStatus` Members

7.6.4.2 Queue Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_STATE	Indicates the presence of `state`. If absent, any previously conveyed state continues to apply.

Table 158: `QueueStatus` Flag

7.6.5 Queue Close

OMM applications encode and send ***Queue Close*** messages to a Queue Provider to close a user's queue.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the queue message type. For a queue close, use **QueueMsgType.CLOSE**.

Table 159: `QueueClose` Member

7.6.6 Queue Data

Both OMM applications and Queue Providers can send and receive ***Queue Data*** messages to exchange data content between queue users and also to communicate whether content was undeliverable.

7.6.6.1 Queue Data Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the queue message type. For queue data, use **QueueMsgType.DATA**.
sourceName	**Required**. Indicates the name of the queue from which the content was sourced. `sourceName` should match the name specified in the `QueueRequest` for this substream.
destName	**Required**. Indicates the name of the queue to which the content is sent.
identifier	**Required**. A user-specified unique identifier for the message being sent. The identifier is used when acknowledging this content via a `QueueAck`.
timeout	**Optional**. Indicates the desired timeout for this content. You can indicate any of the **QueueMsgTimeoutCodes**, as defined in Table 162, or a specific time interval (in milliseconds). If a user-specified timeout value expires during the course of delivery, the content is returned as a **QueueDataExpired** message. If not specified, this defaults to **QueueMsgTimeoutCodes.INFINITE**.
containerType	**Required**. Indicates the type of contents in this `QueueData` message.
flags	**Required**. `flags` indicate more information about this message. For further details, refer to Section 7.6.6.2.
queueDepth	**Required**. Indicates the number of Queue Data or Queue Data Expired messages inbound on this queue stream (following this message).
encodedDataBody	**Optional**. • When sending a message, populate this with pre-encoded content. If sending a message without pre-encoded contents, you can use the encoding methods described in Table 163. • If receiving a message, this can be used to access the payload contents for decoding.

Table 160: `QueueData` Members

7.6.6.2 Queue Data Flags

`QueueData` and `QueueDataExpired` messages use the following flag:

FLAG	DESCRIPTION
QueueDataFlags.POSSIBLE_DUPLICATE == 0x1	Indicates that the message was retransmitted, and the application might have already received it.

Table 161: Queue Data Flags

7.6.6.3 Queue Data Message Timeout Codes

ENUMERATION	MEANING
INFINITE	This message can persist in the system for an infinite amount of time.
IMMEDIATE	This message immediately times out if any portion of its delivery path is unavailable.
PROVIDER_DEFAULT	This message persists in the system for a period of time determined by the provider.

Table 162: Queue Data Message Timeout Codes

7.6.6.4 Queue Data Encoding

FUNCTION NAME	DESCRIPTION
encodeInit	Begins the process of encoding content into the QueueData message. This method takes an EncodeIterator as a parameter, where the EncodeIterator is associated with the buffer into which content is encoded. After this method returns, users should call additional methods required to encode the content. After all encoding is completed, call the encodeComplete method.
encodeComplete	Completes the encoding of content into the QueueData message.
encode	When sending no payload or payload content is preencoded and specified on the QueueData.encodedDataBody buffer, this method encodes the QueueData message in a single call.

Table 163: Queue Data Message Encoding Methods

7.6.6.5 Queue Data Message Encoding Code Sample

```
EncodeIterator _msgEncIter = CodecFactory.createEncodeIterator();
QueueData _queueData = QueueMsgFactory.createQueueData();

// initialize the QueueData encoding
_queueData.clear();
_queueData.streamId(QUEUE_MSG_STREAM_ID);
_queueData.identifier(124);
_queueData.sourceName().data("MY_QUEUE");
_queueData.destName().data("DESTINATION_QUEUE");
_queueData.timeout(QueueMsgTimeoutCodes.INFINITE);
_queueData.containerType(DataTypes.FIELD_LIST);

_msgEncIter.clear();
_msgEncIter.setBufferAndRWFVersion(buffer, tunnelStream.classOfService().common().protocolMajorVersion(),
        tunnelStream.classOfService().common().protocolMinorVersion());

// begin encoding content into QueueData message
if ((ret = _queueData.encodeInit(_msgEncIter)) < ReactorReturnCodes.SUCCESS)
{
    System.out.println("QueueData.encodeInit() failed");
    return;
}

// Start Content Encoding - follow standard field list encoding
// as shown in the Transport API Java Developers Guide examples
//

// when content encoding is done, complete the QueueData encoding
if ((ret = _queueData.encodeComplete(_msgEncIter)) < ReactorReturnCodes.SUCCESS)
{
    System.out.println("QueueData.encodeComplete() failed");
    return;
}
```

Code Example 28: Queue Data Message Encoding Code Sample

7.6.7 QueueDataExpired

If queue data messages sent on a queue stream cannot be successfully delivered, the queue provider sends QueueDataExpired messages on the queue stream to OMM consumer applications.

OMM consumer applications do not send this message.

7.6.7.1 QueueDataExpired Structure Members

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the queue message type. For expired queue data, use **QueueMsgType.DATAEXPIRED**.
flags	**Required**. `flags` indicate more information about this message. For details, refer to Section 7.6.6.2.
sourceName	**Required**. Reversed from the original QueueData message, `sourceName` specifies the name of the queue to which content was sent.
destName	**Required**. Reversed from the original QueueData message (from `sourceName`), `destName` specifies the name of the queue from which content is sourced.
identifier	**Required**. A user-specified, unique identifier for the message. This identifier will be the same as the original `QueueData` message.
undeliverableCode	**Required**. Specifies a code explaining why the content was undeliverable. For more information on undeliverable codes and their meanings, refer to refer to Section 7.6.7.2.
containerType	**Required**. Indicates the type of contents in the message.
encodedDataBody	**Optional**. Contains the payload contents (if any) of the original Queue Data message.
queueDepth	**Required**. Indicates how many Queue Data or Queue Data Expired messages are still inbound on this queue stream (following this message).

Table 164: QueueDataExpired Structure Members

7.6.7.2 Queue Data Message Undeliverable Codes

ENUMERATION	MEANING
EXPIRED	Indicates that the timeout value specified for this message has expired.
NO_PERMISSION	Indicates that the source/sender of this message is not permitted to send or is not permitted to send to the specified destination.
INVALID_TARGET	Indicates that the specified destination of this message does not exist.
QUEUE_FULL	Indicates that the specified destination of this message has a full queue and cannot receive additional content.
QUEUE_DISABLED	Indicates that the specified destination of this message has a disabled queue.
INVALID_SENDER	Indicates that the sender of this message is now invalid.
MAX_MSG_SIZE	Indicates that the message was too large.
TARGET_DELETED	Indicates that the target queue was deleted after sending the message, but before it was delivered.

Table 165: Queue Data Message Undeliverable Codes

7.6.8 Queue Ack

Queue providers encode and send ***Queue Ack*** messages to OMM applications. A queue ack acknowledges that a queue data message is persisted by the queue provider. After a queue provider acknowledges persistence, the OMM application no longer needs to persist the acknowledged content.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Indicates the queue message type. For a queue ack, this is set to **QueueMsgType.ACK**.
identifier	**Required**. The identifier of the message being acknowledged. This should match the `QueueData.identifier` for the acknowledged message.
sourceName	**Required**. Indicates the name of the queue from which the content was sourced. `sourceName` should match the name specified in the `QueueData` message.
destName	**Required**. Indicates the name of the queue to which content was sent.

Table 166: Queue Ack Members

Chapter 8 Payload Cache Detailed View

8.1 Concepts

The Value Added Payload Cache component provides a facility for storing OMM containers (the data payload of OMM messages). Typical use of a payload cache is to store the current image of OMM data streams, where each entry in the cache corresponds to a single data stream. The initial content of a cache entry is defined by the payload of a refresh message. The current (or last) value of the entry is defined by the cumulative application of all refresh and update messages applied to the cache entry container. Values are stored in and retrieved from the cache as encoded OMM containers.

A cache is defined as a collection of OMM data containers. An application may create multiple cache collections, or instances, depending on how it wants to organize the data. The only restriction on cache organization is that all entries in a cache must use the same RDM Field Dictionary to define the set of field definitions it will use. At minimum, a separate cache would be required for each field dictionary in use by the application. However, since cache instances can also share the same field dictionary, partitioning is not restricted to dictionary usage. Some examples of how cache instances can be organized in an application include: all item streams on an RSSL connection; all items belonging to a particular service; all items across the entire application.

The application is responsible for organizing cache instances, managing the lifecycle of all entries in each cache, and applying and retrieving data from the cache. Figure 1 shows an example consumer type application which has created two cache instances to store data from two services on an OMM provider.

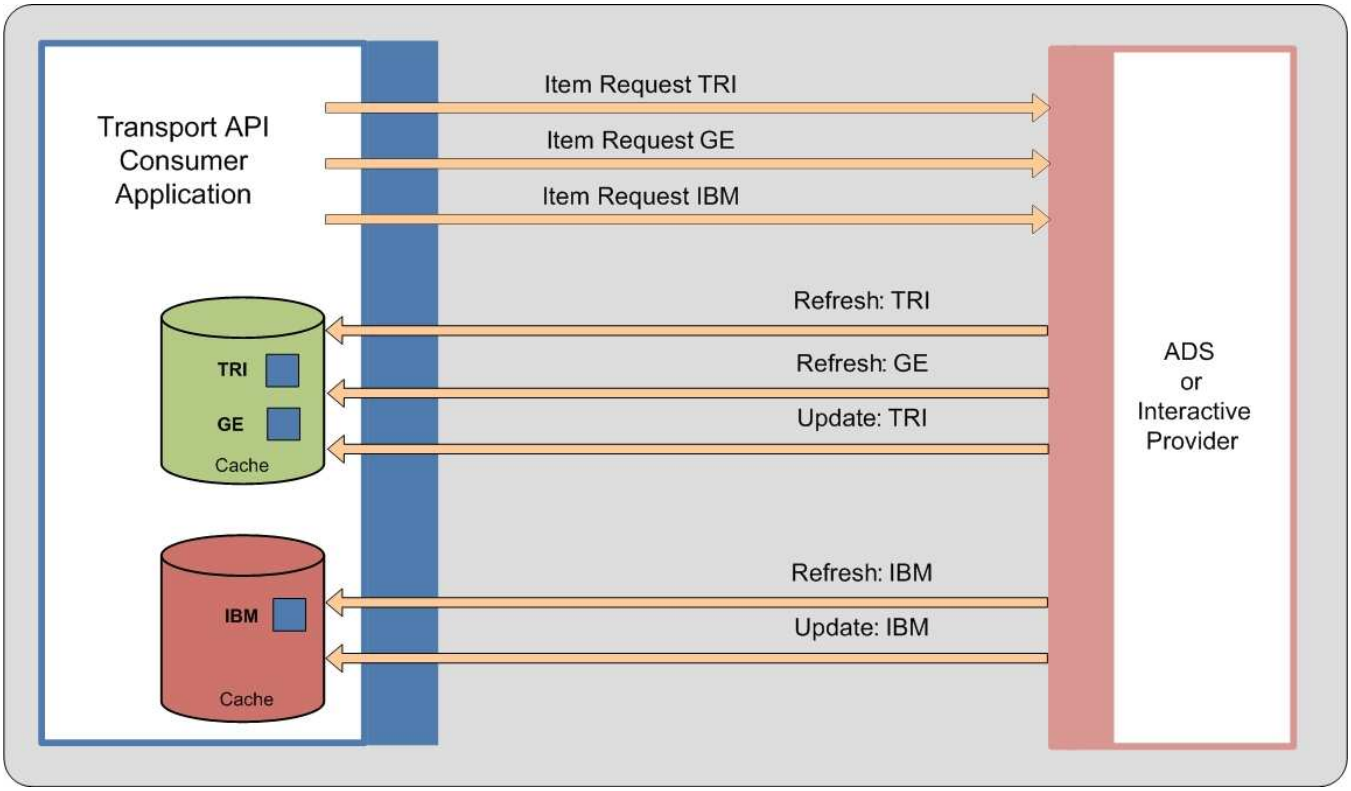

Figure 7: Consumer Application using Cache to Store Payload Data for Item Streams

8.2 Payload Cache

This section describes how the payload cache is managed (initialization and uninitialization), and how instances of cache (collections of payload entries) are created and destroyed.

8.2.1 Payload Cache Management

After the first Value Added Payload Cache instance is created, all global static resources used by the cache are initialized. When the application destroys the last cache instance, the cache releases all the resources it used.

8.2.2 Cache Error Handling

Some of the methods on the payload cache interface use the `CacheError` structure to return error information. This structure will be populated with additional information if an error occurred during the method call. The application should check the return value from methods. The application can optionally provide the `CacheError` structure to obtain additional information.

`CacheError` has the following class members.

CLASS MEMBER	DESCRIPTION
errorId	Range of values is defined by the set of UPA Codec return codes (from the `CodecReturnCodes` enumeration).
text	This `String` will contain text with additional information when a method call returns a failed result.

Table 167: `CacheError` Class Members

The following method handles the `CacheError`.

METHOD NAME	DESCRIPTION
clear	Clears the `CacheError` structure. Use this method prior to passing the structure to a cache interface method.

Table 168: `CacheError` Utility Method

8.2.3 Payload Cache Instances

A payload cache instance is a collection of payload data containers. An empty cache instance must be created before any data can be stored in the cache. When a cache or its entries are no longer needed, it can be destroyed. Use the following methods to create and destroy caches.

METHOD NAME	DESCRIPTION
CacheFactory.createPayloadCache	Creates a payload cache instance. Options are passed in via the `PayloadCacheConfigOptions` defined in Table 170.
destroy	Destroys a payload cache instance. Any entries remaining in the cache are also destroyed at this time.
destroyAll	Destroys all payload cache instances. All entries remaining in the application are also destroyed at this time.

Table 169: `PayloadCache` Management Methods

CLASS MEMBER	DESCRIPTION
maxItems	The maximum number of entries allowed in cache. When the maximum number of items is reached, the cache refuses new entries until entries are removed. The `CacheFactory.createPayloadEntry` method will return a null `PayloadEntry` when the maximum number of items is reached. When set to zero, the cache allows an unlimited number of items. See Table 173.

Table 170: `PayloadCacheConfigOptions` Class Members

8.2.4 Managing RDM Field Dictionaries for Payload Cache

Each cache instance requires an RDM Field Dictionary, to define the set of fields that may be encoded in the OMM containers stored in the cache.

A cache is associated with a field dictionary through a setting process, which requires a `DataDictionary` structure loaded with the field dictionary. The dictionary structure can be loaded from a file (using the `loadFieldDictionary` method), or from an encoded dictionary message from a provider (using the `decodeFieldDictionary` method). The cache does not use the enumerated dictionary content, so loading the enumeration dictionary is not required for cache use. For more information on using `DataDictionary`, refer to the UPA Reference Manual.

Once the `DataDictionary` is loaded, it is set to a cache instance using a key (an arbitrary string identifier assigned by the application to name the dictionary). The key allows multiple cache instances to share the same dictionary. After the first setting of a dictionary, it can be set to additional cache instances by simply providing the same key on additional settings. For a list of methods used to set a dictionary to a cache, refer to Table 171.

The cache builds its own field definition database from the `DataDictionary` definitions. After setting, the application does not need to retain the dictionary structure, since the cache does not refer to the `DataDictionary` used during the setting. In typical usage, the application will likely retain the dictionary for use with other encoding and decoding operations.

Note: A cache can be set to a dictionary only once during its lifetime. While a cache cannot be switched to a new dictionary, the dictionary in use may be extended with new definitions. Refer to Section 8.2.4.3.

8.2.4.1 Setting Methods

METHOD NAME	DESCRIPTION
setDictionary	This method sets a DataDictionary to a cache instance. Use this method the first time a dictionary is set to a cache. The application must provide a key parameter to this method to name the dictionary for future reference. This key will be used in future setting operations when the application wants to share a dictionary between cache instances, or to extend the definitions in the dictionary. The first time a particular key is used with this method will be the initial setting of that dictionary to a cache. The second time the same key is used in this method; it will reload the field definitions from the given DataDictionary structure, enabling the dictionary to be extended. Refer to Section 8.2.4.3.
setSharedDictionaryKey	Use this method when sharing a dictionary among multiple caches. This method will set a cache to a previously set dictionary (identified by the dictionary key name). To share a dictionary, the dictionary must have previously had an initial setting to another cache using the setDictionary method. This method does not require the DataDictionary structure, since that was already loaded during the initial setting with this dictionary key.

Table 171: Methods for Setting Dictionary to Cache

8.2.4.2 Setting Example

In the following example, two cache instances are created and set to a single, shared field dictionary.

```
PayloadCacheConfigOptions cacheConfig = CacheFactory.createPayloadCacheConfig();
cacheConfig.maxItems(0); /* unlimited */

/* For simplicity in this code fragment, CHK is assumed to be a macro for
error handling (performing cleanup and returning from method). */

/* create cache instances */
CacheError cacheError = CacheFactory.createCacheError();
PayloadCache cacheInstance1 = CacheFactory.createPayloadCache(cacheConfig, cacheError);
if (cacheInstance1 == null)
{
    System.out.println("CacheFactory.createPayloadCache failure: " + cacheError.text());
    CHK(cacheError.errorId());
}

PayloadCache cacheInstance2 = CacheFactory.createPayloadCache(cacheConfig, cacheError);
if (cacheInstance2 == null)
{
    System.out.println("CacheFactory.createPayloadCache failure: " + cacheError.text());
    CHK(cacheError.errorId());
}

/* Load an RDM Field Dictionary structure from file: set to each cache. */
DataDictionary dataDictionary = CodecFactory.createDataDictionary();
com.thomsonreuters.upa.transport.Error error = TransportFactory.createError();
int ret = dataDictionary.loadFieldDictionary("RDMFieldDictionary", error); CHK(ret);

String dictionaryKey = "SharedKey1";

/* Initial setting of the dictionary to the first cache */
ret = cacheInstance1.setDictionary(dataDictionary, dictionaryKey, cacheError); CHK(ret);
/* Shared setting of the same dictionary to the second cache */
```

```
ret = cacheInstance2.setSharedDictionaryKey(dictionaryKey, cacheError); CHK(ret);
/* The dataDictionary can be destroyed after setting, but is typically retained by the
       application for encoding and decoding. */

/* Two cache instances are now ready for applying and retrieving data */

/* Cleanup */
cacheInstance1.destroy(); /* destroys all entries and the cache instance */
cacheInstance2.destroy();
```

Code Example 29: Creating Cache and Setting to Dictionary

8.2.4.3 Extending the Cache Field Dictionary

While a cache can only be set to a single dictionary during it's lifetime, the set of field definitions defined by the dictionary can be extended. This is accomplished by reloading the cache field definition database with another call to the `setDictionary` method. When extending the field dictionary, the `DataDictionary` must contain the original field definitions, plus any new definitions the application wishes to use. Changes or deletions to the original field definitions are not supported; only additions are allowed. Using the same `PayloadCache instance` and dictionary key that were previously set, call the `setDictionary` method again with extended dictionary structure.

Note: When extending a field dictionary that is shared, all caches sharing that same dictionary key will see the extension, with only a single call to `setDictionary`. There is no need to set the shared dictionary key again to each cache after a dictionary is extended.

8.2.5 Payload Cache Utilities

Use the methods in the following table to manage each cache instance. These utilities provide a count of the cache entries, and a list of cache entries.

METHOD NAME	DESCRIPTION
entryCount	Returns the number of item payload entries in this cache instance.
entryList	Populates an array list for this cache instance. Because each cache entry is likely associated with an entry in the application's item list, an application would typically manage the entire set of entry instances. This utility provides access to the entire entry instance list if needed.
Clear	Destroys all entries in the cache instance. The empty cache can be reused and remains bound to its data dictionary.

Table 172: `PayloadCache` Utility Methods

8.3 Payload Cache Entries

A payload cache entry stores a single OMM container (containers are defined by `DataTypes`). While a cache entry can store any arbitrary OMM data, the primary use case is to maintain the last known value of an item data stream by applying the sequence of refresh and update messages in the stream to the cache entry. The initial data applied to a container must be a refresh message payload, which will define the container type to be stored (e.g. Map). As refresh and update messages from the item stream are applied to the cache entry, the cache decodes the OMM data and sets the current value by following the OMM rules for the container (e.g. adding, deleting or updating map entries in a Map, or updating fields in a field list). The last value of the data stream can be retrieved from cache at any time as an encoded OMM container.

8.3.1 Managing Payload Cache Entries

Payload cache entries are created within a cache instance. Use the `CacheFactory.createPayloadEntry` method to create a cache entry instance. You cannot move entries between different cache instances, due to their dependency on the field dictionary set to the cache where they are created.

Cache entries only store the payload container of an item. Maintain other item data (e.g. message key attributes, domain, state) as needed in an item list managed by the application, which will identify the source or sink associated with the cache entry data. This item list will likely include the `PayloadEntry instance` if the payload of the item is cached.

For a list of basic utilities provided by the payload cache to manage the collection of entries in the cache, refer to Section 8.2.5.

Manage cache entries with the following methods:

METHOD NAME	DESCRIPTION
CacheFactory.createPayloadEntry	Returns a newly created entry in the cache defined by the given `PayloadCache` instance. This method will return a null instance if it cannot create the entry (e.g. if the maximum number of entries as defined in `PayloadCacheConfigOptions` would be exceeded).
destroy	Destroys the cache entry defined `PayloadEntry` instance and removes it from its cache.
clear	Any data contained in the cache entry instance is deleted, and the entry is returned to its initial state. The entry itself remains in the cache and can be re-used.

Table 173: PayloadEntry Management Methods

8.3.2 Applying Data

Data is applied to a cache entry from the payload of an OMM message by using the `apply` method. The decoded `Msg` and a `DecodeIterator` are passed to the apply method. The iterator (positioned at the start of the encoded payload data `Msg.encodeDataBody`) will be used to decode the OMM data so that the cache entry data can be set or updated.

Some caching behaviours are controlled by flags in the `Msg`. When a `RefreshMsg` is applied to the cache entry, the following `RefreshMsgFlags` will be followed during the application:

- `CLEAR_CACHE`: The cache entry data will be cleared prior to applying this message.
- `DO_NOT_CACHE`: The payload will not be applied to the cache entry.

When an `UpdateMsg` is applied to cache, the following `UpdateMsgFlags` will be followed:

- `DO_NOT_CACHE`: The payload data will not be applied to the cache entry.
- `DO_NOT_RIPPLE`: When this data is applied, entry rippling is not performed .

The following example demonstrates how to create a payload entry in a cache instance and apply the payload of an `Msg` to the cache entry.

```
/* For simplicity in this code fragment, CHK is assumed to be a macro for
error handling (performing cleanup and returning from method). */

CacheError cacheError = CacheFactory.createCacheError();
PayloadEntry entryInstance = CacheFactory.createPayloadEntry(cacheInstance, cacheError);
if (entryInstance == null)
{
    System.out.println("Error " + cacheError.errorId() + " creating cache entry: " + cacheError.text());
    CHK(cacheError.errorId());
}

/* Apply buffer containing an encoded Msg to cache entry */
int applyBufferToCache(Channel channel, TransportBuffer buffer, PayloadEntry entryInstance)
{
    /* Perform message decoding. */
    DecodeIterator dIter = CodecFactory.createDecodeIterator();
    dIter.clear();
    dIter.setBufferAndRWFVersion(buffer, channel.majorVersion(), channel.minorVersion());
    Msg msg = CodecFactory.createMsg();
    int ret = msg.decode(dIter);
    if (ret < CodecReturnCodes.SUCCESS)
    {
        System.out.println("Failure decoding message from buffer");
        CHK(ret);
    }

    /* Apply the decoded Msg to cache, with iterator positioned at the start of the payload */
    CacheError cacheError = CacheFactory.createCacheError();
    ret = entryInstance.apply(dIter, msg, cacheError);
    if (ret < CodecReturnCodes.SUCCESS)
    {
        System.out.println("Error " + cacheError.errorId() + " applying data to cache entry: " +
        cacheError.text());
        CHK(ret);
    }

    CHK(ret);
}
```

Code Example 30: Applying Data to a Payload Cache Entry

8.3.3 Retrieving Data

Data is retrieved from a cache entry as an encoded OMM container by using the `retrieve` method. The application provides the data buffer (via an `EncodeIterator`) where the container will be encoded. The retrieve method supports both UPA encoding scenarios. When using `Msg.encodedDataBody`, the encoded content retrieved from the cache entry can be set on the Msg data body. If using `encodeInit` and `encodeComplete` encoding, the cache retrieve method can be used to encode the message payload prior to `encodeComplete`.

There are two options for using the `retrieve` method. For single-part retrieval, the buffer provided by the application must be large enough to hold the entire encoded container. For multi-part retrieval, the application makes a series of calls to `retrieve` to get the OMM container in fragments (e.g. a sequence of maps are retrieved which together contain the entire set of map entries for the OMM container). In this usage, the optional `PayloadCursor instance` is required to maintain the state of the multi-part retrieval. Container types `FieldList` and `ElementList` cannot be fragmented, so the buffer size must be large enough to retrieve the entire container.

The following methods describe data-related operations on a cache entry.

METHOD NAME	DESCRIPTION
dataType	Returns the `DataType` stored in the cache entry instance. When initially created (or after the entry is cleared), the data type will be `UNKNOWN`. The data type is defined by the container type of the first refresh message applied to the entry.
apply	Applies the OMM data in the payload of the `Msg` to the cache entry instance. The first message applied must be a refresh message (class `REFRESH`).
retrieve	Retrieves data from the cache entry by encoding the OMM container into the buffer provided with the `EncodeIterator` given by the application. The buffer can be `Buffer` or `TransportBuffer`. For single-part retrieval, the `PayloadCursor` parameter is optional. For details on multi-part retrieval, refer to Section 8.3.3.1.

Table 174: Methods for Applying and Retrieving Cache Entry Data

8.3.3.1 Multi-Part Retrieval

For data types that support fragmentation, the container can be retrieved in multiple parts by calling `retrieve` until the complete container is returned. To support multi-part retrieval, the optional `PayloadCursor` parameter is required when calling `retrieve`. The cursor is used to maintain the position where the next retrieval will resume. The application must check the state of the cursor after each call to `retrieve` to determine when the retrieval is complete. The following methods are needed when using the payload cursor.

METHOD NAME	DESCRIPTION
CacheFactory.createPayloadCursor	Creates a cursor for optional use in the `retrieve` method (required for multi-part retrieval). Returns the `PayloadCursor` instance.
destroy	Destroys the cursor instance.
clear	Clears the state of the cursor instance. Whenever retrieving data from a cache entry, the cursor must be cleared prior to the first call to `retrieve`. Clearing the cursor also allows it to be reused with a retrieval on a different container.
isComplete	Returns the completion state of a retrieval where the `PayloadCursor` instance was used. The state must be checked after each call to `retrieve` to determine if there is any additional data to be encoded for the cache entry container. When the cursor state is complete, the entire container of the cache entry has been retrieved.

Table 175: Methods for Using the Payload Cursor

8.3.3.2 Buffer Management

In multi-part usage, the size of the buffer used in the calls to `retrieve` will affect how many fragments are required to retrieve the entire image of the cache entry. The retrieve method will continue to encode OMM entries from the cache container until it runs out of room in the buffer to encode the next entry. To progress during a multi-part retrieval, the buffer size must be at least large enough to encode a single OMM entry from the payload container. For example, if retrieving a map in multiple parts, the buffer must be large enough to encode at least one MapEntry on each retrieval.

There are three general outcomes when using the `retrieve` method:

- Full cache container is encoded into the buffer. This can occur with or without the use of the optional `PayloadCursor` instance. If used in this scenario, the cursor state would indicate the retrieval is complete.
- Partial container encoded into the buffer. This is only possible when using the `PayloadCursor` instance for container types that support fragmentation. The application must check the cursor to test if this is the final part.
- No data encoded into container due to insufficient buffer size. This can occur with or without the use of the optional `PayloadCursor` instance. The application may retrieve again with a larger buffer.

8.3.3.3 Example: Cache Retrieval with Multi-Part Support

The following example illustrates data retrieval from a cache entry, which supports multi-part encoding of a container.

```
/*Code fragment showing use of retrieve for multi-part retrieval.*/
com.thomsonreuters.upa.transport.Error error = TransportFactory.createError();
TransportBuffer buffer = channel.getBuffer(DEFAULT_BUFFER_SIZE, false, error);

int ret;
CacheError cacheError = CacheFactory.createCacheError();
PayloadCursor cursorInstance = CacheFactory.createPayloadCursor();
cursorInstance.clear();
EncodeIterator eIter = CodecFactory.createEncodeIterator();
while (!cursorInstance.isComplete())
{
    eIter.clear();
    eIter.setBufferAndRWFVersion(buffer, channel.majorVersion(), channel.minorVersion());

    /* entryInstance created outside the scope of this code fragment */
    ret = entryInstance.retrieve(eIter, cursorInstance, cacheError);
    if (ret == CodecReturnCodes.SUCCESS)
    /* buffer is big enough to hold whole container data. Application can used encoded data, e.g. set the
           payload on Msg.encodedDataBody and encode a message to be transmitted. */
    else if (ret == CodecReturnCodes.BUFFER_TOO_SMALL)
        /* Increase buffer size and reallocate buffer. */
    else
        /* Handle terminal error condition. See cacheError.text() for additional information. */
}

cursorInstance.destroy();
```

Code Example 31: Cache Retrieval with Multi-Part Support

Appendix A Value Added Utilities

The Value Added Utilities are a collection of common classes, mainly used by the Transport API Reactor. Included is a selectable bidirectional queue used to communicate events between the Reactor and Worker threads. Other Value Added Utilities include a simple queue along with iterable and concurrent versions of it.

The Value Added Utilities are internally leveraged by the Transport API Reactor and cache so applications need not be familiar with their use.

THOMSON REUTERS

Document Version: 1.0
Date of Issue: 30 April 2016