

Enterprise Message API

C++ Edition

3.9.2.L1

ENTERPRISE MESSAGE API DEVELOPERS GUIDE

Document Version: 3.9.2.L1
Date of issue: December 2025
Document ID: EMAC392L1UM.250



© LSEG 2015 - 2025. All rights reserved.

Republication or redistribution of LSEG Data & Analytics content, including by framing or similar means, is prohibited without the prior written consent of LSEG Data & Analytics. 'LSEG Data & Analytics' and the LSEG Data & Analytics logo are registered trademarks and trademarks of LSEG Data & Analytics.

Any software, including but not limited to: the code, screen, structure, sequence, and organization thereof, and its documentation are protected by national copyright laws and international treaty provisions. This manual is subject to U.S. and other national export regulations.

LSEG Data & Analytics, by publishing this document, does not guarantee that any information contained herein is and will remain accurate or that use of the information will ensure correct and faultless operation of the relevant service or equipment. LSEG Data & Analytics, its agents, and its employees, shall not be held liable to or through any user for any loss or damage whatsoever resulting from reliance on the information contained herein.

Contents

1	Introduction	1
1.1	About this Manual	1
1.2	Audience	1
1.3	Programming Language.....	1
1.4	Acronyms and Abbreviations	1
1.5	References.....	2
1.6	Documentation Feedback	3
1.7	Document Conventions	3
2	Product Overview.....	4
2.1	Enterprise Message API Product Description	4
2.2	Product Documentation and Learning the Enterprise Message API.....	4
2.2.1	<i>Consumer Examples</i>	5
2.2.2	<i>Provider Examples</i>	5
2.3	Product Architecture.....	6
2.3.1	<i>Enterprise Message API Consumer Architecture</i>	6
2.3.2	<i>Enterprise Message API Provider Architecture</i>	6
2.3.3	<i>Enterprise Message API Codec Architecture</i>	7
2.3.4	<i>Enterprise Message API Operational Models</i>	7
2.3.5	<i>Enterprise Message API Error Handling</i>	8
2.4	Tunnel Streams	9
3	OMM Containers and Messages	10
3.1	Overview	10
3.2	Classes	11
3.2.1	<i>DataType Class</i>	11
3.2.2	<i>DataCode Class</i>	11
3.2.3	<i>Data Class</i>	11
3.2.4	<i>Msg Class</i>	12
3.2.5	<i>OmmError Class</i>	12
3.2.6	<i>TunnelStreamRequest and ClassOfService Classes</i>	12
3.3	Working with OMM Containers	13
3.3.1	<i>Example: Populating a FieldList Class</i>	13
3.3.2	<i>Example: Populating a Map Class Relying on the FieldList Memory Buffer</i>	14
3.3.3	<i>Example: Populating a Map Class Relying on the Map Class Buffer</i>	14
3.3.4	<i>Example: Extracting Information from a FieldList Class</i>	15
3.3.5	<i>Example: Application Filtering on the FieldList Class</i>	16
3.3.6	<i>Example: Extracting FieldList information using a Downcast Operation</i>	16
3.4	Working with OMM Messages	18
3.4.1	<i>Example: Populating the GenericMsg with an ElementList Payload</i>	18
3.4.2	<i>Example: Extracting Information from the GenericMsg Class</i>	18
3.4.3	<i>Example: Working with the TunnelStreamRequest Class</i>	19
4	Consumer Classes	20
4.1	OmmConsumer Class	20
4.1.1	<i>Connecting to a Server and Opening Items</i>	20
4.1.2	<i>Opening Items Immediately After OmmConsumer Object Instantiation</i>	21
4.1.3	<i>Destroying the OmmConsumer Object</i>	21
4.1.4	<i>Example: Working with the OmmConsumer Class</i>	21
4.1.5	<i>Working with Items</i>	21

4.1.6	<i>Example: Working with Items</i>	22
4.1.7	<i>Working with Tunnel Streams</i>	22
4.1.8	<i>Example: Working with Tunnel Streams</i>	23
4.2	OmmConsumerClient Class	24
4.2.1	<i>OmmConsumerClient Description</i>	24
4.2.2	<i>Example: OmmConsumerClient</i>	24
4.3	OmmConsumerConfig Class	25
4.3.1	<i>OmmConsumerConfig Description</i>	25
4.3.2	<i>Unencrypted Connections</i>	25
4.3.3	<i>Encrypted Connections</i>	25
4.3.4	<i>HTTP Proxy Connections</i>	26
5	Provider Classes	27
5.1	OmmProvider Class	27
5.1.1	<i>Connecting to ADH and Submitting Items</i>	27
5.1.2	<i>Interactive Providers: Post OmmProvider Object Instantiation</i>	28
5.1.3	<i>Non-Interactive Providers: Post OmmProvider Object Instantiation</i>	28
5.1.4	<i>Non-Interactive Providers: Encrypted Connections and HTTP Proxy Tunneling</i>	28
5.1.5	<i>Destroying the OmmProvider Object</i>	28
5.1.6	<i>Non-Interactive Example: Working with the OmmProvider Class</i>	29
5.1.7	<i>Interactive Provider Example: Working with the OmmProvider Class</i>	30
5.1.8	<i>Interactive Provider Example: Handling Post Message</i>	30
5.1.9	<i>Interactive Provider Example: Handling RTT Responses from Consumer</i>	31
5.1.10	<i>Working with Items</i>	31
5.1.11	<i>Packing with Providers</i>	32
5.2	OmmProviderClient Class	35
5.2.1	<i>OmmProviderClient Description</i>	35
5.2.2	<i>Non-Interactive Example: OmmProviderClient</i>	35
5.2.3	<i>Interactive Example: OmmProviderClient</i>	36
5.3	OmmNiProviderConfig and OmmlProviderConfig Classes	37
6	Consuming Data from the Cloud	38
6.1	Overview	38
6.2	Encrypted Connections	38
6.3	Credential Management	38
6.4	Version 1 Authentication Using oAuth Password and Refresh_Token	39
6.4.1	<i>Client_ID (AppKey) and Client Secret</i>	39
6.4.2	<i>Obtaining Initial Access and Refresh Tokens</i>	39
6.4.3	<i>Refreshing the Access Token and Sending a Login Reissue</i>	40
6.5	Version 2 Authentication Using OAuth Client Credentials	41
6.5.1	<i>Configuring and Managing Version 2 Credentials</i>	41
6.5.2	<i>Version 2 OAuth Client Credentials Token Lifespan</i>	41
6.6	Service Discovery	41
6.7	Consuming Market Data	43
6.8	HTTP Error Handling for Reactor Token Reissues	43
6.9	Cloud Connection Use Cases	44
6.9.1	<i>Session Management Use Case</i>	44
6.9.2	<i>Query Service Discovery</i>	44
6.10	Logging of Authentication and Service Discovery Interaction	45
6.10.1	<i>Logged Request Information</i>	45
6.10.2	<i>Logged Response Information</i>	45
7	Warm Standby Feature	46
7.1	Overview	46

7.2	Warm Standby Modes.....	46
7.3	Warm Standby Configuration and Feature Details.....	48
8	Preferred Host Feature	49
8.1	Overview	49
8.2	Preferred Host Reconnection Behavior Changes	49
8.3	Preferred Host Operation Steps.....	49
8.3.1	<i>ChannelSet Behaviors with Preferred Host Options Enabled.</i>	49
8.3.2	<i>Warm Standby Configuration with Preferred Host Options Enabled</i>	50
9	Request Routing	51
9.1	Administrative Domains Behaviors	51
9.1.1	<i>Login Request Timer Handling and Login Response Aggregation</i>	52
9.1.2	<i>Aggregated Login Elements</i>	52
9.1.3	<i>Scenarios for Receiving Aggregated Login Stream</i>	53
9.1.4	<i>Directory Request Timer Handling and Directory Response Aggregation</i>	53
9.1.5	<i>Dictionary Request Timer Handling</i>	54
9.2	Service List.....	54
9.3	Item Request Routing and Recovery	55
9.4	Posting Messages.....	56
9.5	Sending Generic Message.....	56
9.6	Session Channel Information from OmmConsumer and OmmConsumerEvent.....	56
10	Troubleshooting and Debugging.....	57
10.1	Enterprise Message API Logger Usage.....	57
10.2	Omm Error Client Classes	57
10.2.1	<i>Error Client Description</i>	57
10.2.2	<i>Example: Error Client</i>	58
10.3	OmmException Class.....	59
10.4	Creating a DACSLOCK for Publishing Permission Data.....	59

1 Introduction

1.1 About this Manual

This document is authored by Enterprise Message API architects and programmers. Several of its authors have designed, developed, and maintained the Enterprise Message API product and other LSEG products which leverage it.

This guide documents the functionality and capabilities of the Enterprise Message API C++ Edition. The Enterprise Message API can also connect to and leverage many different LSEG and customer components. If you want the Enterprise Message API to interact with other components, consult that specific component's documentation to determine the best way to configure for optimal interaction.

1.2 Audience

This document provides detailed yet supplemental information for application developers writing to the Enterprise Message API.

1.3 Programming Language

The Enterprise Message API is written using the C++ programming language taking advantage of the object oriented approach to design and development of API and applications.

1.4 Acronyms and Abbreviations

ACRONYM / TERM	MEANING
ADH	LSEG Real-Time Advanced Distribution Hub is the horizontally scalable service component within the LSEG Real-Time Distribution System providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	LSEG Real-Time Advanced Distribution Server is the horizontally scalable distribution component within the LSEG Real-Time Distribution System providing highly available services for tailored streaming and snapshot data, publication and contribution messaging with optional persistence, conflation and delay capabilities.
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
Enterprise Message API (EMA)	The Enterprise Message API is an ease of use, open source, Open Message Model API. EMA is designed to provide clients rapid development of applications, minimizing lines of code and providing a broad range of flexibility. It provides flexible configuration with default values to simplify use and deployment. EMA is written on top of the Enterprise Transport API utilizing the Value Added Reactor and Watchlist features of ETA.
Enterprise Transport API (ETA)	Enterprise Transport API is a high performance, low latency, foundation of the LSEG Real-Time SDK. It consists of transport, buffer management, compression, fragmentation and packing over each transport and encoders and decoders that implement the Open Message Model. Applications written to this layer achieve the highest throughput, lowest latency, low memory utilization, and low CPU utilization using a binary Rssl Wire Format when publishing or consuming content to/from LSEG Real-Time Distribution Systems.
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)

Table 1: Acronyms and Abbreviations

ACRONYM / TERM	MEANING
JSON	JavaScript Object Notation
JWK	JSON Web Key. Defined by RFC 7517, a JWK is a JSON formatted public or private key.
JWKS	JSON Web Key Set, This is a set of JWK, placed in a JSON array.
JWT	JSON Web Token. Defined by RFC 7519, JWT allows users to create a signed claim token that can be used to validate a user.
OMM	Open Message Model
QoS	Quality of Service
RDM	Domain Model
DP	Delivery Platform: this platform is used for REST interactions. In the context of Real-Time APIs, an API gets authentication tokens and/or queries Service Discovery to get a list of Real-Time - Optimized endpoints using DP.
LSEG Real-Time Distribution System	LSEG Real-Time Distribution System is LSEG's financial market data distribution platform. It consists of the LSEG Real-Time Advanced Distribution Server and LSEG Real-Time Advanced Distribution Hub. Applications written to the LSEG Real-Time SDK can connect to this distribution system.
Reactor	The Reactor is a low-level, open-source, easy-to-use layer above the Enterprise Transport API. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users.
RMTES	A multi-lingual text encoding standard
RSSL	Source Sink Library
RTT	Round Trip Time, this definition is used for round trip latency monitoring feature.
RWF	Rssl Wire Format, an LSEG proprietary binary format for data representation.
LDF-D	Data Feed Direct
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 References

- Enterprise Message API C++ Edition *LSEG Domain Model Usage Guide*
- *API Concepts Guide*
- Enterprise Message API C++ Edition *Configuration Guide*
- Enterprise Message API C++ Edition *Developers Guide*
- The [LSEG Developer Community](#)

1.6 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 ProductDocumentation@lseg.com.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to LSEG by clicking **Send File** in the **File** menu. Use the ProductDocumentation@lseg.com address.

1.7 Document Conventions

This document uses the following types of conventions:

- C++ classes, methods, in-line code snippets, and types are shown in **Courier New** 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 are shown in Courier New font against a gray background. For example:

```
AppClient client;
OmmConsumer consumer( OmmConsumerConfig().operationModel(
    OmmConsumerConfig::UserDispatchEnum ).host( "localhost:14002" ).username( "user" ) );
consumer.registerClient( ReqMsg().domainType( MMT_MARKET_BY_PRICE ).serviceName(
    "DIRECT_FEED" ).name( "BBH.ITC" ).privateStream( true ), client );
unsigned long long startTime = getCurrentTime();
```

2 Product Overview

2.1 Enterprise Message API Product Description

The Enterprise Message API is a data-neutral, multi-threaded, ease-of-use API providing access to OMM and RWF data. As part of the LSEG Real-Time Software Development Kit, or RTSDK, the Enterprise Message API allows applications to consume and provide OMM data at the message level of the API stack. The message level is set on top of the transport level which is handled by the Enterprise Transport API.

The Enterprise Message API:

- Provides a set of easy-to-use and intuitive interfaces and features intended to aid in message-level application development. These interfaces simplify the setting of information in and getting information from OMM containers and messages. Other interfaces abstract the behavior of consumer-type and provider-type applications.
- Enables applications to source market data from, and provide it to, different components that support OMM and/or RWF (e.g. Real-Time, LSEG Real-Time Distribution System, LSEG Real-Time Advanced Transformation Server, Data Feed Direct, etc.).
- Leaves a minimal code footprint in applications written to it. The design of the Enterprise Message API and its interfaces allows application development to focus more on the application business logic than on the usage of the Enterprise Message API.
- Includes training applications that provide basic, yet still functional, examples of Enterprise Message API applications.
- Presents applications with simplified access to OMM messages and containers while providing all necessary transport level functionalities. Generally, Enterprise Message API applications are meant to process market data items (e.g. open and receive item data or provide item data).
- Abstracts and hides all the transport level functionality minimizing application involvement to just optional transport level configuration and server address specification.
- Provides simple **set-** and **get-**type functionality to populate and read OMM containers and messages. Enterprise Message API takes advantage of fluent interface design, which users can leverage to set disparate values of the same message or container by stringing respective interface methods together, one after the other. Fluent interfaces provide the means for visual code simplification which helps in understanding and debugging applications.

Transport level functionality is abstracted, specialized, and encapsulated by the Enterprise Message API in a few classes whose functionality is implied by their class name.

2.2 Product Documentation and Learning the Enterprise Message API

When learning the Enterprise Message API, LSEG recommends you set up a sandbox environment where developers can experiment with various iterations of Enterprise Message API applications. Enterprise Message API is designed to facilitate a hands-on (experiment-based) learning experience (versus a documentation-based methodology). To support a hands-on learning methodology, the Enterprise Message API package provides a set of training examples which showcase the usage of Enterprise Message API interfaces in increasing levels of complexity and sophistication. While coding and debugging applications, developers are encouraged to refer to the *Enterprise Message API C++ Edition Reference Manual* and/or to the features provided by their IDE (e.g., IntelliSense).

NOTE: Enterprise Message API application developers should already be familiar with OMM and Market Data distribution systems.

2.2.1 Consumer Examples

The complexity of a consumer example is reflected in its series number as follows:

- 100-series examples simply open an item and print its received content to the screen (using the `Data::toString()` method). Applications in this series illustrate Enterprise Message API support for stringification, containers, and primitives. Though useful for learning, debugging, and writing display applications, stringification by itself is not sufficient to develop more sophisticated applications.
- The 200-series examples illustrate how to extract information from OMM containers and messages in native data formats, (e.g., `UInt64`, `EmaString`, and `EmaBuffer`).
- The 300- and 400- series examples depict usage of particular Enterprise Message API features such as posting, generic message, programmatic configuration, and etc.
- The 500-series examples illustrate how to use the Preferred Host and Request Routing features.

2.2.2 Provider Examples

The complexity of an example is reflected in its series number. Each provider type (i.e., non-interactive versus interactive) has its own directory structure in the product package:

- 100-series examples simply create streaming items and submit their refreshes and updates. Applications in this series use the hardcoded Enterprise Message API configuration.
- The 200-series examples showcase the submission of multiple, streaming items from different market domains. Applications in this series use the `EmaConfig.xml` file to modify its configuration.
- The 300- and 400- series examples depict usage of particular Enterprise Message API features such as user control of the source directory domain, login streaming, connection recovery, programmatic configuration, and etc.

2.3 Product Architecture

2.3.1 Enterprise Message API Consumer Architecture

The Enterprise Message API incorporates the ValueAdded Reactor component (called the Transport API VA Reactor) from the Transport API, which provides the watchlist and transport-level functionality. The Enterprise Message API wraps up the reactor component in its own class of **OmmConsumer**. **OmmConsumer** provides interfaces to open, modify, and close market items or instruments, as well as submit Post and Generic messages. To complete the set of consumer application functionalities, the **OmmConsumer** class provides the **dispatch()** method. Depending on its design and configuration, an application might need to call this method to dispatch received messages. The **OmmConsumerConfig** class configures the reactor and **OmmConsumer**.

The **OmmConsumerClient** class provides the callback mechanism for Enterprise Message API to send incoming messages to the application. The application needs to implement a class inheriting from the **OmmConsumerClient** class to receive and process messages. By default, **OmmConsumerClient** callback methods are executed in Enterprise Message API's thread of control. However, you can use the **OmmConsumerConfig::operationModel()** interface to execute callback methods on the application thread. If you choose to execute callback methods in this manner, the application must also call the **OmmConsumer::dispatch()** method to dispatch received messages.

The Enterprise Message API consumer will always have at least one thread, which is implemented by the VA Reactor and runs the internal, VA Reactor logic. For details on this thread, refer to the *Transport API C++ Edition Value Added Component Developers Guide*. Additionally, you can configure the Enterprise Message API to create a second, internal thread to dispatch received messages. To create a second thread, set the **OmmConsumerConfig** operation model to **OmmConsumerConfig::ApiDispatchEnum**. If the **OmmConsumerConfig** operation model is set to the **OmmConsumerConfig::UserDispatch**, the Enterprise Message API will not run a second thread. Without running a second thread, the application is responsible for calling the **Ommconsumer::dispatch()** method to dispatch all received messages.

 **WARNING!** If the application delays in dispatching messages, it can result in slow consumer behavior.

2.3.2 Enterprise Message API Provider Architecture

The Enterprise Message API provider incorporates the Value Added (VA) Reactor component from the Enterprise Transport API, which provides transport-level functionality. The Enterprise Message API wraps the reactor component in its own class of **OmmProvider**. **OmmProvider** provides interfaces to submit item messages as well as handling login, directory, and dictionary domains (depending on Enterprise Message API's specific provider role). To complete the set of provider functionalities, the **OmmProvider** class provides the **dispatch()** method. Depending on its design and configuration, an application might need to call this method to dispatch received messages. The provider configuration class (i.e., **OmmNiProviderConfig** or **OmmIProviderConfig**) class configures both the reactor and **OmmProvider**.

Enterprise Message API sends incoming messages to the application using the **OmmProviderClient** callback mechanism. To receive and process messages, the application needs to implement a class that inherits from the **OmmProviderClient** class. By default, **OmmProviderClient** callback methods are executed in Enterprise Message API's thread of control. However, you can use either the **OmmNiProviderConfig::operationModel()** or **OmmIProviderConfig::operationModel()** interface to execute callback methods on the application's thread, in which case the application must also call the **OmmProvider::dispatch()** method to dispatch received messages.

An Enterprise Message API provider must always have at least one thread, which is implemented by the VA Reactor and runs the internal, VA Reactor logic. For details on this thread, refer to the *Transport API C++ Edition Value Added Component Developers Guide*. Additionally, you can configure Enterprise Message API to create a second internal thread over which to dispatch received messages:

- For non-interactive providers, set the **OmmNiProviderConfig** operation model to **OmmNiProviderConfig::ApiDispatchEnum**. If the operation model is set to **OmmNiProviderConfig::UserDispatchEnum**, Enterprise Message API will not run a second thread.
- For interactive providers, set the **OmmIProviderConfig** operation model to **OmmIProviderConfig::ApiDispatchEnum**. If the operation model is set to **OmmIProviderConfig::UserDispatchEnum**, Enterprise Message API will not run a second thread.

Without running a second thread, the application is responsible for calling the **OmmProvider::dispatch()** method to dispatch all received messages.

The Enterprise Message API provider includes an internal, hard-coded, and configurable initial source directory refresh message. The application can either use the internal hard-coded source directory, configure its own internal one via the `EmaConfig.xml` file, or programmatically create one and/or disable the internal one. To disable the internal source directory message:

- When running Enterprise Message API as a non-interactive provider: the application must set `OmmNiProviderConfig::UserControlEnum` through the `OmmNiProviderConfig::adminControlDirectory()` method.
- When running Enterprise Message API as an interactive provider: the application must set `OmmIProviderConfig::UserControlEnum` through the `OmmIProviderConfig::adminControlDirectory()` method. Additionally, you can configure the ability to disable internal dictionary responses by setting `OmmIProviderConfig::UserControlEnum` through the `OmmIProviderConfig::adminControlDictionary()` method.

NOTE: If the user control is enabled, the application is responsible for sending the response messages.

An Enterprise Message API provider also supports the programmatic configuration of a source directory refresh of dictionary information, which overrides any configuration in `EmaConfig.xml`. To programmatically configure a source directory refresh:

- When running Enterprise Message API as a non-interactive provider: the application must set `OmmNiProviderConfig::ApiControlEnum` through the `OmmNiProviderConfig::adminControlDictionary()` method. An Enterprise Message API non-interactive provider does not support programmatically configuring dictionary information.
- When running Enterprise Message API as an interactive provider: the application must set `OmmIProviderConfig::ApiControlEnum` through the `OmmIProviderConfig::adminControlDictionary()` method. Additionally, you can programmatically configure dictionary information, which overrides any dictionary information defined from `EmaConfig.xml`. To programmatically configure dictionary information, set `OmmIProviderConfig::ApiControlEnum` through the `OmmIProviderConfig::adminControlDictionary()` method.

2.3.3 Enterprise Message API Codec Architecture

The Enterprise Message API Codec uses the Enterprise Transport API decoding and encoding functions to read and populate OMM containers and messages. Each OMM container and message is represented by a respective Enterprise Message API interface class, which provides relevant methods for setting information on, and accessing information from, these containers and messages. All classes representing OMM containers, messages, and primitives inherit from the common parent class of `Data`. Through such inheritance, classes provide the same basic, common, and easy to use functionality that applications might expect from them (e.g., printing contained data using `toString()`).

2.3.4 Enterprise Message API Operational Models

Enterprise Message API allows applications to control when additional information is read from the wire.

There are two operational models to control that:

- API Dispatch (default)
- User Dispatch

Enterprise Message API does not maintain an internal queuing mechanism. All messages provided on callbacks are either read from the network or generated by Enterprise Message API in response to network connection events for recovery/notification purposes.

NOTE: As an example, the API may read up to 100 updates messages at once from the network and convey one message at a time via `onUpdateMsg` callback in the order they were received.

The following table provides information on each operation model.

OPERATION MODEL	DESCRIPTION
API Dispatch	With API Dispatch, the Enterprise Message API creates a separate API dispatching thread to handle dispatching from the network. The maximum number of messages read from the network is specified by the MaxDispatchCountApiThread parameter. Once messages are read, the API delivers them one at a time via the appropriate callback. API dispatch can be further tuned by setting the DispatchTimeoutApiThread parameter (in microseconds). When set to -1 , the API dispatch thread becomes active when notified of incoming content. Otherwise, the API dispatch remains inactive for the duration of DispatchTimeoutApiThread before checking for a notification. Applications expecting to receive content sparsely may choose to set a non-negative value for DispatchTimeoutApiThread .
User Dispatch	With User Dispatch, the application must invoke a dispatch call to process incoming content. It is up to the application to balance dispatching with any other required operations. In this case, the MaxDispatchCountUserThread parameter specifies the maximum number of messages that the API dispatches in a single call to dispatch() with a callback per message.

Table 2: Enterprise Message API Operational Models

For every **OMMConsumer** or **OMMProvider** instance created by the application, the corresponding API dispatch thread (API Dispatch), or a call to dispatch from each instance (User Dispatch), will handle dispatching messages sequentially via callback on the associated AppClients.

For details on configuring the operation model for each application role—such as OMMConsumer, OMMIProvider, or OMMNIProvider—refer to the *Enterprise Message API Configuration Guide*. The guide also explains operation model-specific configuration parameters that can be adjusted to optimize performance.

2.3.5 Enterprise Message API Error Handling

The Enterprise Message API interfaces are designed to throw exceptions when encountering error scenarios: **OmmConsumer** and **OmmProvider** classes throw an **OmmException**. For details on exception types, refer to Section 10.3. For which exceptions are thrown by each class, refer to the Reference Manual.

The Enterprise Message API application must catch exceptions and take appropriate actions. Notably, exceptions related to decoding or encoding content must be caught with appropriate try and catch blocks in callbacks or wherever decoding or encoding takes place. Decoding methods to retrieve content may result in exceptions if the content is not present within the message. It is best practice to check for a field's existence prior to accessing content.

While the **OmmConsumer** and **OmmProvider** classes throw an **OmmException** to report an error condition, the **OmmConsumerErrorClient** class provides an alternate reporting mechanism via callbacks. To use the alternate error reporting, pass the **OmmConsumerErrorClient** on the constructor of the **OmmConsumer** class, which switches the error reporting from exception throwing to callbacks. Similarly, passing in **OmmProviderErrorClient** on the constructor of the **OmmProvider** class switches reporting error to the callbacks. Refer to Section 10.2.2 for sample code to set error callbacks such as **onInvalidHandle**, **onSystemError**, etc. When using API dispatch, the recommendation is to use callbacks for receiving errors by passing in **ErrorClient**. In addition to its error reporting mechanisms, Enterprise Message API provides a logger mechanism which is useful in monitoring Enterprise Message API behavior and debugging any issues that might arise.

2.4 Tunnel Streams

By leveraging the Transport API Value Added Reactor, the Enterprise Message API allows users to create and use special tunnel streams. A tunnel stream is a private stream that has additional behaviors associated with it, such as end-to-end line of sight for authentication and reliable delivery. Because tunnel streams are founded on the private streams concept, these are established between consumer and provider endpoints and then pass through intermediate components, such as LSEG Real-Time Distribution System or the LSEG Real-Time Edge Device.

The user creating the tunnel stream sets any additional behaviors to enforce, which Enterprise Message API sends to the provider application end point. The provider endpoint acknowledges the creation of the stream as well as the behaviors it will enforce on the stream. Once this is accomplished, negotiated behaviors are enforced on the content exchanged via the tunnel stream.

The tunnel stream allows for multiple substreams to exist, where substreams flow and coexist within the confines of a specific tunnel stream. In the following diagram, imagine the tunnel stream as the orange cylinder that connects the consumer application and the Provider application. Notice that this passes directly through any intermediate components. The tunnel stream has end-to-end line of sight so the Provider and Consumer are effectively talking to each other directly, although they are traversing multiple devices in the system. Each of the black lines flowing through the cylinder represent a different substream, where each substream is 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. The substreams established over a tunnel stream are not managed by tunnel stream handling in the APIs: this must be handled by the application. Example: if application chooses to do posting over tunnel stream, the API allows for easily constructing a post and sending it over tunnel streams but does not manage receipt of ACK messages.

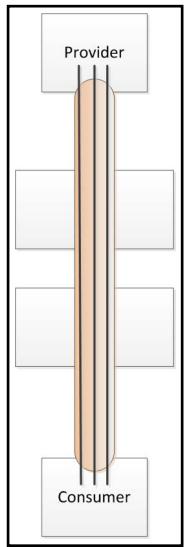


Figure 1. Tunnel Stream

3 OMM Containers and Messages

3.1 Overview

Enterprise Message API supports a full set of OMM containers, messages, and primitives (e.g. **FieldList**, **Map**, **RefreshMsg**, **Int**). For simplicity, Enterprise Message API uses:

- The “set / add” type of functionality to populate OMM containers, messages, and primitives
 - Set functionality is used to specify variables that occur once in an OMM container or message.
 - Add functionality is used to populate entries in OMM containers.
 - Set and add type methods return a reference to the modified object (for fluid interface usage).
- The “get” type of functionality to read and extract data from OMM containers, messages, and primitives. Enterprise Message API uses a simple iterative approach to extract entries from OMM containers, one at a time. Applications iterate over every OMM container type in the same way.

While iterating, an application can apply a filtering mechanism. For example, while iterating over a **FieldList**, the application can specify a field ID or field name in which it is interested; the Enterprise Message API skips entries without matching identification. Individual container entries are extracted during iteration. Depending on the container type, the entry may contain:

- Its own identity (e.g., field id)
- An action to be applied to the received data (e.g., add action)
- Permission information associated with the received data
- An entry’s load and its **data** type.

The Enterprise Message API has two different ways of extracting an entry’s load:

- Use ease-of-use interfaces to return references to contained objects (with reference type being based on the load’s data type)
- Use the **getLoad** interface to return a reference to the base **Data** class. The **getLoad** interface enables more advanced applications to use the down-cast operation (if desired).

For details on ease of use interfaces and the down-cast operation, refer to Section 3.3.

To provide compile time-type safety on the set-type interfaces, Enterprise Message API provides the following, deeper inheritance structure:

- All classes representing primitive / intrinsic data types inherit from the **Data** class (e.g. **OmmInt**, **OmmBuffer**, **OmmRmtes**, etc.).
- **OmmArray** class inherits from the **Data** class. The **OmmArray** is treated as a primitive instead of a container, because it represents a set of primitives.
- **OmmError** class inherits from the **Data** class. **OmmError** class is not an OMM data type.
- All classes representing OMM containers (except **OmmArray**) inherit from the **ComplexType** class, which in turn inherits from the **Data** class (e.g., **OmmXml**, **OmmOpaque**, **Map**, **Series**, or **Vector**).
- All classes representing OMM messages inherit from the **Msg** class, which in turn inherits from the **ComplexType** class (e.g., **RefreshMsg**, **GenericMsg**, or **PostMsg**).

3.2 Classes

3.2.1 DataType Class

The **DataType** class provides the set of enumeration values that represent each and every supported OMM data type, including all OMM containers, messages, and primitives. Each class representing OMM data identifies itself with an appropriate **DataType** enumeration value (e.g., **DataType::FieldListEnum**, **DataType::RefreshMsgEnum**). You can use the **Data::getDataType()** method to learn the data type of a given object.

The **DataType** class list of enumeration values contains two special enumeration values, which can only be received when reading or extracting information from OMM containers or messages:

- **DataType::ErrorEnum**, which indicates an error condition was detected. For more details, refer to Section 3.2.5.
- **DataType::NoDataEnum**, which signifies a lack of data on the summary of a container, message payload, or attribute.

3.2.2 DataCode Class

The **DataCode** class provides two enumeration values that indicate the data's state:

- The **DataCode::NoCodeEnum** indicates that the received data is valid and application may use it.
- The **DataCode::BlankEnum** indicates that the data is not present and application needs to blank the respective data fields.

3.2.3 Data Class

The **Data** class is a parent abstract class from which all OMM containers, messages, and primitives inherit. **Data** provides interfaces common across all its children, which in turn enables down-casting operations. The **Data** class and all classes that inherit from it are optimized for efficiency and built so that data can be easily accessed. Though all primitive data types are represented by classes that inherit from the **Data** class, the ease-of-use interfaces do not return such references: all primitive data types are returned by their intrinsic representation.



WARNING! The **Data** class and all classes that inherit from it are designed as temporary and short-lived objects. For this reason, do not use them as storage or caching devices.

The Enterprise Message API does not support immediately retrieving data from freshly created OMM containers or messages. The following code snippet demonstrates this restriction:

```
FieldList fieldList;

fieldList.addAscii( 1, "ascii" ).addInt( 10, 20 ).complete();

while ( fieldList.forth() )
{
    const FieldEntry& fieldEntry = fieldList.getEntry();

    ...
}
```

3.2.4 Msg Class

The **Msg** class is a parent class for all the message classes. It defines all the interfaces that are common across all message classes.

3.2.5 OmmError Class

The **OmmError** class is a special purpose class. It is a read only class implemented in the Enterprise Message API to notify applications about errors detected while processing received data. This class enables applications to learn what error condition was detected. Additionally it provides the **getAsHex()** method to obtain binary data associated with the detected error condition. The sole purpose of this class is to aid in debugging efforts.

The following code snippet presents usage of the **OmmError** class while processing **ElementList**.

```
void decode( const ElementList& elementList )
{
    while ( elementList.forth() )
    {
        const ElementEntry& elementEntry = elementList.getEntry();

        if ( elementEntry.getCode() == Data::BlankEnum )
            continue;
        else
            switch ( elementEntry.getLoadType() )
            {
                case DataType::RealEnum:
                    cout << elementEntry.getReal().getAsDouble() << endl;
                    break;
                case DataType::ErrorEnum:
                    cout << elementEntry.getError().getErrorCode() << "(" <<
                        elementEntry.getError().getErrorCodeAsString() << ")" << endl;
                    break;
            }
    }
}
```

3.2.6 TunnelStreamRequest and ClassOfService Classes

The **TunnelStreamRequest** class specifies request information for use in establishing a tunnel stream. A tunnel stream is a private stream that provides additional functionalities such as user authentication, end-to-end flow control and reliable delivery. You can configure these features on a per-tunnel stream basis. The **ClassOfService** class specifies these features and some other related parameters. The identity of the tunnel stream is specified on the **TunnelStreamRequest** class.

3.3 Working with OMM Containers

Enterprise Message API supports the following OMM containers: **ElementList**, **FieldList**, **FilterList**, **Map**, **Series**, and **Vector**.

Each of these classes provides set type interfaces for container header information (e.g., dictionary id, element list number, and the add-type interfaces for adding entries). You must set the container header and optional summary before adding the first entry.

Though it is treated as an OMM primitive, the **OmmArray** acts like a container and therefore provides add-type interfaces for adding primitive entries.

NOTE: OMM Container classes do perform some validation of their usage. If a usage error is detected, an appropriate **OmmException** will be thrown.

3.3.1 Example: Populating a FieldList Class

The following example illustrates how to populate a **FieldList** class with fluid interfaces.

```
try {
    FieldList fieldList;

    fieldList.info( 1, 1 )
        .addUInt( 1, 64 )
        .addReal( 6, 11, OmmReal::ExponentNeg2Enum )
        .addDate( 16, 1999, 11, 7 )
        .addTime( 18, 02, 03, 04, 005 )
        .complete();
} catch ( const OmmException & excp ) {
    cout << excp << endl;
}
```

3.3.2 Example: Populating a Map Class Relying on the FieldList Memory Buffer

The following code snippet illustrates how to populate a **Map** class with summary data and a single entry containing a **FieldList**. In this example, the **FieldList** class uses its own memory buffer to store content while it is populated. This buffer later gets copied to the buffer owned by the **Map** class. This container population model applies to all OMM containers that might contain other containers, primitives, or messages.

```
try {
    FieldList fieldList;

    fieldList.addUInt( 1, 64 )
        .addReal( 6, 11, OmmReal::ExponentNeg2Enum )
        .addDate( 16, 1999, 11, 7 )
        .addTime( 18, 02, 03, 04, 005 )
        .complete();

    Map map;
    map.summary( fieldList ).addKeyAscii( "entry_1", MapEntry::AddEnum, fieldList
        ).complete();
} catch ( const OmmException& excp ) {
    cout << excp << endl;
}
```

3.3.3 Example: Populating a Map Class Relying on the Map Class Buffer

The following example illustrates how to populate a **Map** class with a single entry containing a **FieldList**. In this case, the **FieldList** class uses the memory buffer owned by the **Map** class to store its own content while it is populated, therefore avoiding the internal buffer copy described in Section 3.3.2. This container population model applies to iterable containers only (e.g., OmmArray, ElementList, FieldList, FilterList, Map, Series, and Vector).

```
try {
    FieldList fieldList;

    Map map;
    fieldList.addUInt( 1, 64 )
        .addReal( 6, 11, OmmReal::ExponentNeg2Enum )
        .addDate( 16, 1999, 11, 7 )
        .addTime( 18, 02, 03, 04, 005 )
        .complete();

    map.addKeyAscii( "entry_1", MapEntry::AddEnum, fieldList );

    map.complete();
} catch ( const OmmException& excp ) {
    cout << excp << endl;
}
```

3.3.4 Example: Extracting Information from a FieldList Class

In the following example illustrates how to use the `FieldList::forth()` method to extract information from the `FieldList` class by iterating over the class. The following code extracts information about all entries.

```
void decode( const FieldList& fieldList )
{
    if ( fieldList.hasInfo() )
    {
        Int16 dictionaryId = fieldList.getInfoDictionaryId();
        Int16 fieldListNum = fieldList.getInfoFieldListNum();
    }

    while ( fieldList.forth() )
    {
        const FieldEntry& fieldEntry = fieldList.getEntry();

        if ( fieldEntry.getCode() == Data::BlankEnum )
            continue;

        switch ( fieldEntry.getLoadType() )
        {
            case DataType::AsciiEnum :
                const EmaString& value = fieldEntry.getAscii();
                break;
            case DataType::IntEnum :
                Int64 value = fieldEntry.getInt();
                break;
        }
    }
}
```

3.3.5 Example: Application Filtering on the FieldList Class

In the following code snippet application filters or extracts select information from FieldList class. The FieldList::forth(Int16) method is used to iterate over the FieldList class. In this case only entries with field id of 22 will be extracted; all the other ones will be skipped.

```
void decode( const FieldList& fieldList )
{
    while ( fieldList.forth( 22 ) )
    {
        const FieldEntry& fieldEntry = fieldList.getEntry();

        if ( fieldEntry.getCode() == Data::BlankEnum )
            continue;

        switch ( fieldEntry.getLoadType() )
        {
            case DataType::AsciiEnum :
                const EmaString& value = fieldEntry.getAscii();
                break;
            case DataType::IntEnum :
                Int64 value = fieldEntry.getInt();
                break;
        }
    }
}
```

3.3.6 Example: Extracting FieldList information using a Downcast Operation

The following example illustrates how to extract information from a **FieldList** object using the down-cast operation.

```
void AppClient::decodeFieldList( const FieldList& fl )
{
    if ( fl.hasInfo() )
        cout << "FieldListNum: " << fl.getInfoFieldListNum() << " DictionaryId: " << fl
fl.getInfoDictionaryId() << endl;

    while ( fl.forth() )
    {
        cout << "Load" << endl;
        decode( fl.getEntry().getLoad() );
    }
}

void AppClient::decode( const Data& data )
{
    if ( data.getCode() == Data::BlankEnum )
        cout << "Blank data" << endl;
    else
        switch ( data.getDataType() )
```

```

{
    case DataType::RefreshMsgEnum :
        decodeRefreshMsg( static_cast<const RefreshMsg&>( data ) );
        break;
    case DataType::UpdateMsgEnum :
        decodeUpdateMsg( static_cast<const UpdateMsg&>( data ) );
        break;
    case DataType::FieldListEnum :
        decodeFieldList( static_cast<const FieldList&>( data ) );
        break;
    case DataType::MapEnum :
        decodeMap( static_cast<const Map&>( data ) );
        break;
    case DataType::NoDataEnum :
        cout << "NoData" << endl;
        break;
    case DataType::TimeEnum :
        cout << "OmmTime: " << static_cast<const OmmTime&>( data ).toString() << endl;
        break;
    case DataType::DateEnum :
        cout << "OmmDate: " << static_cast<const OmmDate&>( data ).toString() << endl;
        break;
    case DataType::RealEnum :
        cout << "OmmReal::getAsDouble: " << static_cast<const OmmReal&>( data
            ).getAsDouble() << endl;
        break;
    case DataType::IntEnum :
        cout << "OmmInt: " << static_cast<const OmmInt&>( data ).getInt() << endl;
        break;
    case DataType::UIntEnum :
        cout << "OmmUInt: " << static_cast<const OmmUInt&>( data ).getUInt() << endl;
        break;
    case DataType::EnumEnum :
        cout << "OmmEnum: " << static_cast<const OmmEnum&>( data ).getEnum() << endl;
        break;
    case DataType::AsciiEnum :
        cout << "OmmAscii: " << static_cast<const OmmAscii&>( data ).toString() << endl;
        break;
    case DataType::ErrorEnum :
        cout << "Decoding error: " << static_cast<const OmmError&>( data
            ).getErrorCodeAsString() << endl;
        break;
    default :
        break;
}
}

```

3.4 Working with OMM Messages

Enterprise Message API supports the following OMM messages: **RefreshMsg**, **UpdateMsg**, **StatusMsg**, **AckMsg**, **PostMsg** and **GenericMsg**. As appropriate, each of these classes provide set and get type interfaces for the message header, permission, key, attribute, and payload information.

3.4.1 Example: Populating the GenericMsg with an ElementList Payload

The following example illustrates how to populate a **GenericMsg** with a payload consisting of an **ElementList**.

```
try {
    GenericMsg genMsg;

    genMsg.domainType( 200 ).name( "TR.N" ).serviceId( 234 ).payload( ElementList().addAscii(
        "entry_1", "value_1" ).complete() );
} catch ( const OmmException& excp ) {
    cout << excp << endl;
}
```

3.4.2 Example: Extracting Information from the GenericMsg Class

The following example illustrates how to extract information from the **GenericMsg** class.

```
void decode( const GenericMsg& genMsg )
{
    if ( genMsg.hasName() )
        cout << endl << "Name: " << genMsg.getName();

    if ( genMsg.hasHeader() )
        const EmaBuffer& header = genMsg.getHeader();

    switch ( genMsg.getPayload().getDataType() )
    {
        case DataType::FieldListEnum :
            decode( genMsg.getPayload().getFieldList() );
            break
    }
}
```

3.4.3 Example: Working with the `TunnelStreamRequest` Class

The following code snippet demonstrates using the `TunnelStreamRequest` class in the consumer application to open a tunnel stream.

```
CosAuthentication cosAuthentication;
cosAuthentication.type( CosAuthentication::OmmLoginEnum );

CosDataIntegrity cosDataIntegrity;
cosDataIntegrity.type( CosDataIntegrity::ReliableEnum );

CosFlowControl cosFlowControl;
cosFlowControl.type( CosFlowControl::BidirectionalEnum ).recvWindowSize( 1200
    ).sendWindowSize( 1200 );

ClassOfService cos;
cos.authentication( cosAuthentication ).dataIntegrity( cosDataIntegrity ).flowControl(
    cosFlowControl );

TunnelStreamRequest tsr;
tsr.classOfService( cos ).domainType( MMT_SYSTEM ).name( "TUNNEL" ).serviceName( "DIRECT_FEED" );
```

4 Consumer Classes

4.1 OmmConsumer Class

The **OmmConsumer** class is the main consumer application interface to the Enterprise Message API. This class encapsulates watchlist functionality and transport level connectivity. It provides all the interfaces a consumer-type application needs to open, close, and modify items, as well as submit messages to the connected server (both **PostMsg** and **GenericMsg**). The **OmmConsumer** class provides configurable admin domain message processing (i.e., login, directory, and dictionary requests).

4.1.1 Connecting to a Server and Opening Items

Applications observe the following steps to connect to a server and open items:

- **(Optional)** Specify a configuration using the **EmaConfig.xml** file.
This step is optional because the Enterprise Message API provides a default configuration which is usually sufficient in simple application cases.
- Create **OmmConsumerConfig** object (for details, refer to Section 4.3).
If an **EmaConfig.xml** file is not used, then at a minimum, applications might need to modify the default host address and port.
- Implement an application callback client class that inherits from the **OmmConsumerClient** class (for details, refer to Section 4.2).
An application needs to override the default implementation of callback methods and provide its own business logic. Not all methods need to be overridden; only methods required for the application's business logic.
- **(Optional)** Implement an application error client class that inherits from the **OmmConsumerErrorClient** class (for details, refer to Section 10.2).
The application needs to override default error call back methods to be effectively notified about error conditions.
- Create an **OmmConsumer** object and pass the **OmmConsumerConfig** object (and if needed, also pass in the application error client object), and optionally register for Login events by passing in an application callback client class.
- Open items of interest using the **OmmConsumer::registerClient()** method.
- Process received messages.
- **(Optional)** Submit **PostMsg** and **GenericMsg** messages and modify / close items using appropriate **OmmConsumer** class methods.
- Exit.

4.1.2 Opening Items Immediately After OmmConsumer Object Instantiation

To allow applications to open items immediately after creating the **OmmConsumer** object, the Enterprise Message API performs the following steps when creating and initializing the **OmmConsumer** object:

- Create an internal item watchlist.
- Establish connectivity to a configured server / host.
- Log into the server and obtain source directory information.
- Obtain dictionaries (if configured to do so).

4.1.3 Destroying the OmmConsumer Object

Destroying an **OmmConsumer** object causes the application to log out and disconnect from the connected server, at which time all items are closed.

4.1.4 Example: Working with the OmmConsumer Class

The following example illustrates the simplest application managing the OmmConsumer Class.

```
try {
    AppClient client;
    OmmConsumer consumer( OmmConsumerConfig().host( "localhost:14002" ).username( "user" ) );
    consumer.registerClient( ReqMsg().serviceName( "DIRECT_FEED" ).name( "IBM.N" ), client );
    sleep( 60000 );
} catch ( const OmmException& excp ) {
    cout << excp << endl;
}
```

4.1.5 Working with Items

The Enterprise Message API assigns all opened items or instruments a unique numeric identifier (e.g. **UInt64**), called a handle, which is returned by the **OmmConsumer::registerClient()** call. A handle is valid as long as its associated item stays open. Holding onto these handles is important only to applications that want to modify or close particular items, or use the items' streams for sending **PostMsg** or **GenericMsg** messages to the connected server. Applications that just open and watch several items until they exit do not need to store item handles.

While opening an item, on the call to the **OmmConsumer::registerClient()** method, an application can pass an item closure or an application-assigned numeric value. The Enterprise Message API will maintain the association of the item to its closure as long as the item stays open.

Respective closures and handles are returned to the application in an **OmmConsumerEvent** object on each item callback method.

NOTE: Remove the handle when its item stream becomes invalid (due to a closed status or snapshot request). Since handles can be reused by **OmmConsumer::registerClient()** call, this prevents accidentally closing an active stream that shares the same handle.

4.1.6 Example: Working with Items

The following example illustrates using the item handle while modifying an item's priority and posting modified content.

```
void AppClient::onRefreshMsg( const RefreshMsg& refreshMsg, const OmmConsumerEvent& event )
{
    cout << "Received refresh message for item handle = " << event.getHandle() << endl;
    cout << refreshMsg << endl;
}

try {
    AppClient client;
    OmmConsumer consumer( OmmConsumerConfig().host( "localhost:14002" ).username( "user" ) );

    Int64 closure = 1;
    UInt64 itemHandle = consumer.registerClient( ReqMsg().serviceName( "DIRECT_FEED" ).name(
        "IBM.N" ), client, (void*)closure );

    consumer.reissue( ReqMsg().serviceName( "DIRECT_FEED" ).name( "IBM.N" ).priority( 2, 2 ),
                      itemHandle );

    consumer.submit( PostMsg().payload( FieldList().addInt( 1, 100 ).complete() ), itemHandle
                    );
    sleep( 60000 );
} catch ( const OmmException& excp ) {
    cout << excp << endl;
}
```

In the code snippet above, when submitting a message, specifically **PostMsg**, **RefreshMsg**, **StatusMsg**, or **UpdateMsg**, application may specify original publisher information using the Visible Publisher Identifier (VPI) feature. This is set using **publisherId** by setting **UserId** and **UserAddress**. For an explanation of the VPI feature, refer to the *Enterprise Transport API Developers Guide* for an explanation of VPI feature. For more usage information, refer to the Reference Manual.

4.1.7 Working with Tunnel Streams

Enterprise Message API assigns all tunnel streams a unique numeric identifier (e.g., UInt64), called a parent handle, which is returned by the call: **OmmConsumer::registerClient(TunnelStreamRequest,...)**. A parent handle is valid only as long as its associated tunnel stream is open. You can use parent handles to open substreams (as illustrated in Section 4.1.8).

When opening a tunnel stream, on the call to the **OmmConsumer::registerClient(TunnelStreamRequest,...)** method, an application can pass a tunnel stream closure or an application-assigned numeric value. The Enterprise Message API will maintain the association of the tunnel stream to its closure as long as the tunnel stream stays open. Respective closures and parent handles are returned to the application in an **OmmConsumerEvent** object on each tunnel stream callback method.

For more details on a **TunnelStreamRequest** and how to create it, refer to Section 3.2.6 and Section 3.4.3.

4.1.8 Example: Working with Tunnel Streams

The following example illustrates the use of a parent handle (as returned by `OmmConsumer::registerClient(TunnelStreamRequest,...)`) to open a substream from the `OmmConsumerClient::onStatusMsg()` callback.

```

void onStatusMsg(const StatusMsg& statusMsg, const OmmConsumerEvent& event)
{
    if (event.getHandle() == _tunnelStreamHandle &&
        statusMsg.hasState() &&
        statusMsg.getState().getStreamState() == OmmState::OpenEnum )
    {
        // open substream with parent handle returned when opening tunnel stream below
        _pOmmConsumer->registerClient( ReqMsg().name( "TUNNEL_IBM" ).serviceId( 1 ), *this,
                                         (void*)1, _tunnelStreamHandle );
    }
}

int main()
{
    try {
        AppClient client;
        OmmConsumer consumer( OmmConsumerConfig().username( "user" ) );
        client.setOmmConsumer( consumer );
        CosAuthentication cosAuthentication;
        cosAuthentication.type( CosAuthentication::OmmLoginEnum );
        CosDataIntegrity cosDataIntegrity;
        cosDataIntegrity.type( CosDataIntegrity::ReliableEnum );
        CosFlowControl cosFlowControl;
        cosFlowControl.type( CosFlowControl::BidirectionalEnum ).recvWindowSize( 1200
            ).sendWindowSize( 1200 );
        ClassOfService cos;
        cos.authentication( cosAuthentication ).dataIntegrity( cosDataIntegrity
            ).flowControl( cosFlowControl );
        TunnelStreamRequest tsr;
        tsr.classOfService( cos ).domainType( MMT_SYSTEM ).name( "TUNNEL" ).serviceName(
            "DIRECT_FEED" );
        /* open tunnel stream and save tunnel stream parent handle to be used for opening
        substreams in onStatusMsg() callback above */
        _tunnelStreamHandle = consumer.registerClient( tsr, client );

        sleep( 60000 ); // API calls onRefreshMsg(), onUpdateMsg(), or onStatusMsg()
    } catch ( const OmmException& excp ) {
        cout << excp << endl;
    }
}

```

4.2 OmmConsumerClient Class

4.2.1 OmmConsumerClient Description

The **OmmConsumerClient** class provides a callback mechanism through which applications receive OMM messages on items for which they subscribe. The **OmmConsumerClient** is a parent class that implements empty, default callback methods. Applications must implement their own class (inheriting from **OmmConsumerClient**), and override the methods they are interested in processing. Applications can implement many specialized client-type classes; each according to their business needs and design. Instances of client-type classes are associated with individual items while applications register item interests.

The **OmmConsumerClient** class provides default implementation for the processing of **RefreshMsg**, **UpdateMsg**, **StatusMsg**, **AckMsg** and **GenericMsg** messages. These messages are processed by their respectively named methods: **onRefreshMsg()**, **onUpdateMsg()**, **onStatusMsg()**, **onAckMsg()**, and **onGenericMsg()**. Applications only need to override methods for messages they want to process.

4.2.2 Example: OmmConsumerClient

The following example illustrates an application client-type class, depicting **onRefreshMsg()** method implementation.

```
class AppClient : public refinitiv::ema::access::OmmConsumerClient
{
protected :

    void onRefreshMsg( const refinitiv::ema::access::RefreshMsg&, const
                      refinitiv::ema::access::OmmConsumerEvent& );

    void onUpdateMsg( const refinitiv::ema::access::UpdateMsg&, const
                      refinitiv::ema::access::OmmConsumerEvent& );

    void onStatusMsg( const refinitiv::ema::access::StatusMsg&, const
                      refinitiv::ema::access::OmmConsumerEvent& );
};

void AppClient::onRefreshMsg( const RefreshMsg& refreshMsg, const OmmConsumerEvent& )
{
    if ( refreshMsg.hasMsgKey() )
        cout << endl << "Item Name: " << refreshMsg.getName() << endl << "Service Name: " <<
        refreshMsg.getServiceName();

    cout << endl << "Item State: " << refreshMsg.getState().toString() << endl;

    if ( DataType::NoDataEnum != refreshMsg.getPayload().getDataType() )
        decode( refreshMsg.getPayload().getData() );
}
```

4.3 OmmConsumerConfig Class

4.3.1 OmmConsumerConfig Description

You can use the `OmmConsumerConfig` class to customize the functionality of the `OmmConsumer` class. The default behavior of `OmmConsumer` is hard coded in the `OmmConsumerConfig` class. You can configure `OmmConsumer` in any of the following ways:

- Using the `EmaConfig.xml` file
- Using interface methods on the `OmmConsumerConfig` class
- Passing OMM-formatted configuration data through the `OmmConsumerConfig::config(const Data&)` method.

For more details on using the `OmmConsumerConfig` class and associated configuration parameters, refer to the *Enterprise Message API Configuration Guide*.

4.3.2 Unencrypted Connections

The Enterprise Message API supports unencrypted connections via a `ChannelType` of `RSSL_SOCKET` (on Linux or Windows), `RSSL_WEBSOCKET`, and `RSSL_HTTP` (on Windows only). You set `ChannelType` inside of a `ChannelGroup`. For detailed information on `ChannelGroup` and its `ChannelTypes`, refer to the *Enterprise Message API C++ Configuration Guide*.

4.3.3 Encrypted Connections

The Enterprise Message API supports encrypted TCP connections for both `Consumer` and `NiProvider` via a `ChannelType` of `RSSL_ENCRYPTED` (i.e., `ChannelType::RSSL_ENCRYPTED`).

4.3.3.1 Implementing Protocols and Encryption Behavior

The Enterprise Message API's implementation of TLS protocol and encryption depends on a number of factors including:

- The operating system you use (which in turn determines the types of protocols the Enterprise Message API can use):
 - On Linux, the Enterprise Message API uses only OpenSSL.
 - On Windows, the Enterprise Message API can use either WinINet or OpenSSL.
- The type of protocol you use (as specified by `EncryptedProtocolType`):
 - WinINet (specified by `EncryptedProtocolType::RSSL_HTTP`), or
 - OpenSSL (specified by `EncryptedProtocolType::RSSL_SOCKET` or `EncryptedProtocolType::RSSL_WEBSOCKET`).

The Enterprise Message API supports the following OpenSSL protocol versions:

- OpenSSL 1.0
- OpenSSL 1.1
- OpenSSL 3.X

By default, Enterprise Message API first attempts to load OpenSSL 3 and if it cannot, Enterprise Message API then tries OpenSSL 1.1 and then OpenSSL 1.0.

For details on the specific libraries loaded by the Enterprise Message API, refer to Section 4.3.3.2.

For OpenSSL connections, you can set the specific TLS encryption protocol you want to use in the `SecurityProtocol` flag (for details on setting `SecurityProtocol` flags, refer to the *Enterprise Message API C++ Configuration Guide*). Currently, TLS 1.2 and TLS 1.3 are accepted.

4.3.3.2 OpenSSL Libraries

The libraries that the Enterprise Message API uses to implement OpenSSL encryption depends on the machine's operating system and version of OpenSSL in use:

- On Linux:
 - If using OpenSSL 3.X, the Enterprise Message API uses **libssl.so.3** and **libcrypto.so.3**.
 - If using OpenSSL 1.1, the Enterprise Message API uses **libssl.so.1.1** and **libcrypto.so.1.1**.
 - If using OpenSSL 1.0, the Enterprise Message API uses **libssl.so.10** and **libcrypto.so.10**.
- On Windows:
 - If using OpenSSL 3.X, the Enterprise Message API uses **libssl-3-x64.dll** and **libcrypto-3-x64.dll**.
 - If using OpenSSL 1.1, the Enterprise Message API uses **libssl-1_1-x64.dll** and **libcrypto-1_1-x64.dll**.
 - If using OpenSSL 1.0, the Enterprise Message API uses **ssleay32.dll** and **libeay32.dll**.

If you want the Enterprise Message API to load a specific version, you can specify **libssl** and **libcrypto** libraries using **libsslName** and **libcryptoName** (for details on setting these channel parameters, refer to the *Enterprise Message API C++ Configuration Guide*).

NOTE: The RTSDK package does not include OpenSSL libraries. You can obtain compiled OpenSSL libraries from the appropriate OS vendor.

4.3.3.3 Certificate Authority

If you use an OpenSSL Certificate Authority store, you can specify the authority store's location using **openSSLCASStore**. For details on this parameter and the Enterprise Message API's default behavior, refer to the parameter's description in the *Enterprise Message API C++ Configuration Guide*.

4.3.4 HTTP Proxy Connections

The Enterprise Message API supports HTTP proxy tunneling for **ChannelType::RSSL_SOCKET**, **ChannelType::RSSL_HTTP**, and all **ChannelType::RSSL_ENCRYPTED** connection types.

On Windows, WinINet provides legacy HTTP connection type functionality, and you must configure the proxy through the Internet Explorer configuration. You can override WinINet's proxy configuration by using **tunnelingProxyHostName()** and **tunnellingProxyPort()**.

For **RSSL_SOCKET** connection types (standard or encrypted), **libcurl** manages the proxy connection. As with OpenSSL, you can specify a particular **libcurl** library using **libcurlName**. By default:

- On Linux, the Enterprise Message API loads **libcurl.so**
- On Windows, the Enterprise Message API loads **libcurl.dll**

For **libcurl** connections, you can provide additional proxy authentication credentials with the following functions:

- **proxyUserName()**: set the proxy user name.
- **proxyPasswd()**: set the password for proxy authentication.
- **proxyDomain()**: set the domain for proxy authentication.

5 Provider Classes

5.1 OmmProvider Class

The **OmmProvider** class is the main provider application interface to the Enterprise Message API. It encapsulates transport-level connectivity. This class provides all the interfaces a provider-type application needs to submit item messages (i.e., refresh, update, status, generic) and handle login, directory, and dictionary domains (based on whether the provider is interactive or non-interactive). It also provides configurable admin message processing for these domains.

5.1.1 Connecting to ADH and Submitting Items

In the following process, the value for **ProviderType** is dependent on the type of provider with which you are dealing:

- For non-interactive providers, **ProviderType** is **NiProvider**.
- For interactive providers, **ProviderType** is **IProvider**.

► To establish a connection and submit items:

1. (Optional) Specify a configuration using the **EmaConfig.xml** file.

This is optional because the Enterprise Message API provides a default configuration that is usually sufficient in simple application cases.

2. Create the appropriate **OmmProviderTypeConfig** object (for details, refer to Section 5.3):

- For a non-interactive provider, create an **OmmNiProviderConfig** object.
- For an interactive provider, create an **OmmIProviderConfig** object.

3. (Optional) Change the Enterprise Message API configuration using methods on the **OmmProviderTypeConfig** class.

If **EmaConfig.xml** file is not used, then at a minimum:

- Non-interactive provider applications might need to modify both the default host address and port.
- Interactive provider applications might need to modify the default port.

4. (Conditional) Implement an application callback client class that inherits from the **OmmProviderClient** class (for details, refer to Section 5.2).

An application might need to override the default callback implementation and provide its own business logic. Not all methods need to be overridden: only those that require the application's business logic.

- For non-interactive providers, this step is optional as the application may choose not to open login or dictionary items. In such cases, the provider application will not receive return messages.
- For interactive providers, this step is required, as at a minimum, the application needs to handle all inbound login domain and item request messages.

5. (Optional) Implement an application error client class that inherits from the **OmmProviderErrorClient** class (for details, refer to Section 5.2).

To be effectively notified about error conditions, the application needs to override any default, error callback methods.

6. Create an **OmmProvider** object and pass the **OmmProviderTypeConfig** object (and if needed, also pass in the application error client object), and optionally in **NiProvider** only, register for Login events by passing in an application callback client class.

7. (Optional) For non-interactive providers, open login and dictionary items using the **OmmProvider::registerClient()** method.

8. Process received messages.

9. Create, populate, and submit item messages (refresh, update, status).

- For non-interactive providers, the application needs to associate each item with a handle that uniquely identifies the item.
- For interactive providers, the application needs to use the handle from the **OMMProviderEvent**.

10. (Optional) Submit **GenericMsg** messages using the appropriate **OmmProvider** class methods.

11. Exit.

5.1.2 Interactive Providers: Post OmmProvider Object Instantiation

Before an interactive provider can submit items, the application must accept a login request and send the login response (Enterprise Message API only accepts connections). After login, the consumer requests the source directory, which Enterprise Message API submits.

When creating and initializing the **OmmProvider** object, the Enterprise Message API does the following so applications can submit items:

1. Accept the connection request from a consumer.
2. Accept the login.
3. Submit source directory information.

5.1.3 Non-Interactive Providers: Post OmmProvider Object Instantiation

After creating an **OmmProvider** object, the Enterprise Message API performs the following steps when creating and initializing the **OmmProvider** object so that applications can begin submitting items:

1. Establish connectivity to a configured server or host.
2. Log in to Advanced Distribution Hub and submit source directory information.

5.1.4 Non-Interactive Providers: Encrypted Connections and HTTP Proxy Tunneling

Non-interactive providers support both encrypted and HTTP proxy tunneling connections. Configuration details are identical to that of the Consumer when setting up these types of connections.

- For details on using an encrypted connection, refer to Section 4.3.3.
- For details on using an HTTP proxy tunneling connection, refer to Section 4.3.4.

5.1.5 Destroying the OmmProvider Object

For non-interactive providers, destroying an **OmmProvider** object causes the application to log out and disconnect from the connected Advanced Distribution Hub, at which time all items are closed.

For interactive providers, destroying an **OmmProvider** object causes Enterprise Message API to close all consumer connections.

5.1.6 Non-Interactive Example: Working with the OmmProvider Class

The following example illustrates the simplest application managing the `OmmProvider` class.

```

try
{
    OmmProvider provider( OmmNiProviderConfig().host( "localhost:14003" ).username
        ( "user" ) );
    UInt64 itemHandle = 5;

    provider.submit( RefreshMsg() .serviceName( "NI_PUB" ) .name( "IBM.N" )
        .state( OmmState::OpenEnum, OmmState::OkEnum, OmmState::NoneEnum, "UnSolicited
            Refresh Completed" )
        .payload( FieldList()
            .addReal( 22, 3990, OmmReal::ExponentNeg2Enum )
            .addReal( 25, 3994, OmmReal::ExponentNeg2Enum )
            .addReal( 30, 9, OmmReal::Exponent0Enum )
            .addReal( 31, 19, OmmReal::Exponent0Enum )
            .complete() )
        .complete(), itemHandle );

    sleep( 1000 );

    for ( Int32 i = 0; i < 60; i++ )
    {
        provider.submit( UpdateMsg() .serviceName( "NI_PUB" ) .name( "IBM.N" )
            .payload( FieldList()
                .addReal( 22, 3391 + i, OmmReal::ExponentNeg2Enum )
                .addReal( 30, 10 + i, OmmReal::Exponent0Enum )
                .complete() ), itemHandle );
        sleep( 1000 );
    }
}
catch ( const OmmException& excp )
{
    cout << excp << endl;
}
return 0;
}

```

5.1.7 Interactive Provider Example: Working with the OmmProvider Class

The following example illustrates the simplest interactive application managing the **OmmProvider** class.

```

try
{
    AppClient appClient;

    OmmProvider provider( OmmIProviderConfig().port( "14002" ), appClient );

    while ( itemHandle == 0 ) sleep(1000);

    for ( Int32 i = 0; i < 60; i++ )
    {
        provider.submit( UpdateMsg().domainType( MMT_MARKET_BY_ORDER ).payload( Map()
            .addKeyAscii( OrderNr, MapEntry::UpdateEnum, FieldList()
                .addRealFromDouble( 3427, 7.76 + i * 0.1, OmmReal::ExponentNeg2Enum )
                .addRealFromDouble( 3429, 9600 )
                .addEnum( 3428, 2 )
                .addRmtes( 212, EmaBuffer( "Market Maker", 12 ) )
                .complete() )
            .complete() ), itemHandle );

        sleep( 1000 );
    }
}

catch ( const OmmException& excp )
{
    cout << excp << endl;
}

return 0;

```

5.1.8 Interactive Provider Example: Handling Post Message

The following example illustrates how to have **OmmProvider** send an **AckMsg** in response to a **PostMsg**. For more information on support of post messages by a provider, refer to the *Transport API C Edition Developers Guide*.

```

void AppClient::onPostMsg( const PostMsg& postMsg, const OmmProviderEvent& event )
{
    if (postMsg.getSolicitAck())
    {
        AckMsg ackMsg;
        ackMsg.domainType(postMsg.getDomainType());
        ackMsg.ackId(postMsg.getPostId());
        if (postMsg.hasSeqNum())
        {

```

```
        ackMsg.setSeqNum(postMsg.getSeqNum());
    }
    event.getProvider().submit(ackMsg, event.getHandle());
}
}
```

5.1.9 Interactive Provider Example: Handling RTT Responses from Consumer

The following example implements a provider's callback for Generic messages. The example illustrates how the provider can identify and process consumer responses to RTT requests.

```

void AppClient::onGenericMsg(const GenericMsg& genericMsg, const OmmProviderEvent& event)
{
    if (genericMsg.getDomainType() == MMT_LOGIN && event.getHandle() == loginHandle &&
        genericMsg.getPayload().getDataType() == DataType::ElementListEnum)
    {
        cout << "Received login RTT message from Consumer " << event.getHandle() << endl;
        TimeValue currTicks = GetTime::getTicks();
        const ElementList& elementList = genericMsg.getPayload().getElementList();
        while (elementList.forth())
        {
            const ElementEntry& elementEntry = elementList.getEntry();
            if (elementEntry.getName() == ENAME_RTT_TICKS && elementEntry.getLoadType() ==
                DataType::UIntEnum) // "Ticks"
            {
                cout << "\tRTT Tick value is " << elementEntry.getUInt() << "us." << endl;
                lastLatency = (UInt64)((double)currTicks - (double)elementEntry.getUInt()) /
                    GetTime::ticksPerMicro();
                cout << "\tLast RTT message latency is " << lastLatency << "us." << endl;
            }
            else if (elementEntry.getName() == ENAME_RTT_TCP_RETRANS && elementEntry.getLoadType() ==
                DataType::UIntEnum) // "TcpRetrans"
            {
                cout << "\tConsumer side TCP retransmissions: " << elementEntry.getUInt() << endl;
            }
        }
    }
}

```

5.1.10 Working with Items

The application assigns unique numeric identifiers, called handles (e.g., UInt64) to all open items it is providing. Application must pass this identifier along with an item message on the call to **submit()**. The handles are used to manage item stream ids. To reassign a handle to a different item, application must first close the item previously associated with the given handle.

5.1.11 Packing with Providers

Provider applications can use the **PackedMsg** object to send multiple messages packed together in a single packet. Applications can designate the bounds of the **PackedMsg** by setting its limit for messages packed, the byte limit of data it can send, and then pack messages before sending them together.

The following sections provide packing examples for an Interactive Provider and a Non-interactive Provider.

5.1.11.1 Interactive Provider Packing Example

The following example illustrates an Interactive Provider application setting up a basic **PackedMsg** object and packing messages together before submitting **PackedMsg**.

```
void sendPackedMessagesExample(OmmProvider provider, UInt64 clientHandle, UInt64 itemHandle)
{
    FieldList fieldList; // Field list used for message payload

    PackedMsg packedMsg(provider);
    packedMsg.initBuffer(clientHandle); // Initialize buffer using client handle and default size of
                                         // 6000. See reference manual for other uses of initBuffer().

    for (int i = 0; i < 10; i++) // Send 10 packed messages every second (in case of packed buffer
        // sufficient, if not can be send more then one packed buffer per second)
    {
        for (int j = 0; j < 10; j++) // Pack 10 messages
        {
            fieldList.clear();
            fieldList.addReal(22, 3991 + j, OmmReal::ExponentNeg2Enum);
            fieldList.addReal(30, 10 + j, OmmReal::Exponent0Enum);
            fieldList.complete();

            UpdateMsg msg;

            msg.payload(fieldList);
            try
            {
                packedMsg.addMsg(msg, itemHandle); // Add message with its item handle
            }
            catch (const OmmInvalidUsageException& excp)
            {
                //The API was unable to add the current message into the packed buffer.
                //If messages have been successfully added to the packed buffer, submit them,
                //get a new packed buffer, and add the current message into that new buffer.
                if (excp.getErrorCode() == OmmInvalidUsageException::BufferTooSmallEnum)
                {
                    if (packedMsg.packedMsgCount() > 0) // Packed message has some data
                    {
                        // Submit the messages we've already packed, get a new packed buffer,
                        //and add the current message.
                        provider.submit(packedMsg); //Submit packed message on OmmProvider
                        packedMsg.initBuffer(clientHandle); // Re-initialize buffer for next set of
                    }
                }
            }
        }
    }
}
```

```

        packed messages.
        packedMsg.addMsg(msg, itemHandle); // Add missed message with its item handle
    }
    else
    {
        //Packed buffer too small to add even first message.
        //Consider initializing the buffer to a higher value than the default 6000 bytes
        // if needed.
        //See initBuffer() methods for more details.
    }
}
else
{
    // Handle other exceptions from addMsg() here
}
}

if (packedMsg.packedMsgCount() > 0)
{
    provider.submit(packedMsg); //Submit packed message on OmmProvider
    packedMsg.initBuffer(clientHandle); // Re-initialize buffer for next set of packed
    messages.
}
else
{
    // Nothing to submit because packed message is empty.
}
sleep(1000);
}
}
}

```

5.1.11.2 Non-interactive Provider Packing Example

The following example illustrates a Non-interactive Provider application setting up a basic **PackedMsg** object and packing messages together before submitting the **PackedMsg**.

```

void sendPackedMessagesExample(OmmProvider provider, UInt64 itemHandle)
{
    FieldList fieldList; // Field list used for message payload

    PackedMsg packedMsg(provider);
    packedMsg.initBuffer(); // Initialize buffer with default size of 6000. See reference manual for
    // other uses of initBuffer().

    for (int i = 0; i < 10; i++) // Send 10 packed messages every second
    {
        for (int j = 0; j < 10; j++) // Pack 10 messages

```

```

{
    fieldList.clear();
    fieldList.addReal(22, 3991 + j, OmmReal::ExponentNeg2Enum);
    fieldList.addReal(30, 10 + j, OmmReal::Exponent0Enum);
    fieldList.complete();

    UpdateMsg msg;

    msg.payload(fieldList);
    try
    {
        packedMsg.addMsg(msg, itemHandle); // Add message with its item handle
    }
    catch (const OmmInvalidUsageException& excp)
    {
        //The API was unable to add the current message into the packed buffer.
        //If messages have been successfully added to the packed buffer, submit them, get a new
        //packed buffer, and add the current message into that new buffer.
        if (excp.getErrorCode() == OmmInvalidUsageException::BufferTooSmallEnum)
        {
            if (packedMsg.packedMsgCount() > 0) // Packed message has some data.
            {
                // Submit the messages we've already packed, get a new packed buffer, and add
                // the current message.
                provider.submit(packedMsg); //Submit packed message on OmmProvider.
                packedMsg.initBuffer(); // Re-initialize buffer for next set of packed
                //messages.
                packedMsg.addMsg(msg, itemHandle); // Add missed message with its item handle
            }
            else
            {
                //Packed buffer too small to add even first message.
                //Consider initializing the buffer to a higher value than the default 6000 bytes
                //if needed.
                //See initBuffer() methods for more details.
            }
        }
        else
        {
            // Handle other exceptions from addMsg() here.
        }
    }
}

if (packedMsg.packedMsgCount() > 0)
{
    provider.submit(packedMsg); // Submit packed message on OmmProvider.
    packedMsg.initBuffer(); // Re-initialize buffer for next set of packed messages.
}
else

```

```

    {
        // Nothing to submit because packed message is empty.
    }
    sleep(1000);
}
}

```

5.2 OmmProviderClient Class

5.2.1 OmmProviderClient Description

The **OmmProviderClient** class provides a callback mechanism through which applications receive OMM messages on items for which they subscribe. The **OmmProviderClient** is a parent class that implements empty, default callback methods. Applications must implement their own class (inheriting from **OmmProviderClient**), and override the methods they are interested in processing. Applications can implement many specialized client-type classes; each according to their business needs and design. Instances of client-type classes are associated with individual items while applications register item interests. The **OmmProviderClient** class provides default implementation for the processing of **RefreshMsg**, **StatusMsg**, and **GenericMsg** messages. These messages are processed by their respectively named methods: **onRefreshMsg()**, **onStatusMsg()**, **onGenericMsg()**, **onRequest()**¹, **onReIssue()**¹, **onClose()**¹, and **onPost()**¹. Applications only need to override methods for messages they want to process.

5.2.2 Non-Interactive Example: OmmProviderClient

The following example illustrates an application client-type class, depicting **onRefreshMsg()** method implementation.

```

class AppClient : public refinitiv::ema::access::OmmProviderClient
{
protected :
    void onRefreshMsg( const refinitiv::ema::access::RefreshMsg&, const
                      refinitiv::ema::access::OmmProviderEvent& );
    void onStatusMsg( const refinitiv::ema::access::StatusMsg&, const
                      refinitiv::ema::access::OmmProviderEvent& );
    bool _bConnectionUp;
};

void AppClient::onRefreshMsg( const RefreshMsg& refreshMsg, const OmmProviderEvent&
                             ommEvent )
{
    cout << endl << "Handle: " << ommEvent.getHandle() << " Closure: " <<
        ommEvent.getClosure() << endl;
    cout << refreshMsg << endl;

    if ( refreshMsg.getState().getStreamState() == OmmState::OpenEnum )
    {
        if ( refreshMsg.getState().getDataState() == OmmState::OkEnum )
            _bConnectionUp = true;
    }
}

```

1. Interactive Provider Only

```

        else
            _bConnectionUp = false;
    }
else
    _bConnectionUp = false;
}

```

5.2.3 Interactive Example: OmmProviderClient

The following example illustrates an application client-type class, depicting `onRefreshMsg()` method implementation.

```

void AppClient::processLoginRequest( const ReqMsg& reqMsg, const OmmProviderEvent& event )
{
    event.getProvider().submit(RefreshMsg().domainType(MMT_LOGIN).name(reqMsg.getName()).
        nameType(USER_NAME).complete().solicited( true ).state( OmmState::OpenEnum, OmmState::OkEnum, OmmState::NoneEnum,
        "Login accepted" ),event.getHandle() );
}

void AppClient::processMarketByOrderRequest( const ReqMsg& reqMsg, const OmmProviderEvent&
    event )
{
    if ( itemHandle != 0 )
    {
        processInvalidItemRequest(reqMsg, event);
        return;
    }

    event.getProvider().submit(RefreshMsg().domainType(MMT_MARKET_BY_ORDER).
        name(reqMsg.getName()).serviceName(reqMsg.getServiceName()).solicited(true)

        .summaryData( FieldList().addEnum( 15, 840 ).addEnum( 53, 1 ).addEnum( 3423, 1 ) .
        addEnum( 1709, 2 ).complete() )
        .addKeyAscii( OrderNr, MapEntry::AddEnum, FieldList()
            .addRealFromDouble( 3427, 7.76, OmmReal::ExponentNeg2Enum )
            .addRealFromDouble( 3429, 9600 )
            .addEnum( 3428, 2 )
            .addRmtes( 212, EmaBuffer( "Market Maker", 12 ) )
            .complete() )
        .complete() )
        .complete(), event.getHandle() );

    itemHandle = event.getHandle();
}

```

```

void AppClient::processInvalidItemRequest( const ReqMsg& reqMsg, const OmmProviderEvent&
                                         event )
{
    event.getProvider().submit( StatusMsg().name( reqMsg.getName() ).serviceName(
        reqMsg.getServiceName() )
        .domainType( reqMsg.getDomainType() )
        .state( OmmState::ClosedEnum, OmmState::SuspectEnum, OmmState::NotFoundEnum,
                "Item not found" ),
        event.getHandle() );
}

void AppClient::onReqMsg( const ReqMsg& reqMsg, const OmmProviderEvent& event )
{
    switch ( reqMsg.getDomainType() )
    {
        case MMT_LOGIN:
            processLoginRequest( reqMsg, event );
            break;
        case MMT_MARKET_BY_ORDER:
            processMarketByOrderRequest( reqMsg, event );
            break;
        default:
            processInvalidItemRequest( reqMsg, event );
            break;
    }
}

```

5.3 OmmNiProviderConfig and OmmIProviderConfig Classes

In the following, the value for **ProviderType** is dependent on the type of provider with which you are dealing, thus:

- For non-interactive providers, **ProviderType** is **NiProvider**. Config class is **OmmNiProviderConfig**.
- For interactive providers, **ProviderType** is **IProvider**. Config class is **OmmIProviderConfig**.

You can use the **OmmProviderTypeConfig** class to customize the functionality of the **OmmProvider** class. The default behavior of **OmmProvider** is hard coded in the **OmmProviderTypeConfig** class. You can configure **OmmProvider** in any of the following ways:

- Using the **EmaConfig.xml** file
- Using interface methods on the **OmmProviderTypeConfig** class
- Passing OMM-formatted configuration data through the **OmmProviderTypeConfig::config(const Data&)** method.

For more details on using the **OmmProviderTypeConfig** class and associated configuration parameters, refer to the *Enterprise Message API Configuration Guide*.

6 Consuming Data from the Cloud

6.1 Overview

You can use the Enterprise Message API to consume data from a cloud-based LSEG Real-Time Advanced Distribution Server. The API interacts with cloud-based servers using the following workflows:

- Credential Management (for details, refer to Section 6.3)
- Service Discovery (for details, refer to Section 6.6)
- Consuming Market Data (for details, refer to Section 6.7)
- Login Reissue (for details, refer to Section 6.4.3)

There are two versions of login credentials for the Delivery Platform:

- Version 1 Authentication also known as “V1 auth”, “OAuthPasswordGrant” or “V1 Password Credentials”: Uses the OAuth2.0 Password grant or Refresh Token grant. Requires a Machine Account consisting of username and password; also requires a client ID generated by the **LSEG AppGenerator**. For details, refer to Section 6.4.
- Version 2 Authentication also known as “V2 auth”, “OAuthClientCredentials” or “V2 Client Credentials”: Uses OAuth2.0 Client Credentials grant to obtain an access token. Requires a Service Account consisting of client ID and client Secret. For details, refer to Section 6.5.

The Enterprise Transport API will determine which authentication version to use based on the inputs. By default, for cloud connections the Enterprise Message API connects to a server in the **us-east-1** cloud location.

For further details on Real-Time as it functions in the cloud, refer to the *Real-Time — Optimized: Installation and Configuration for Client Use*. For details on the parameters you use to configure cloud connections, refer to the *EMA C++ Edition Configuration Guide*.

6.2 Encrypted Connections

When connecting to an LSEG Real-Time Advanced Distribution Server in the cloud, you must use a **ChannelType** of **RSSL_ENCRYPTED** (for details on **ChannelType**, refer to the *Enterprise Message API C++ Configuration Guide*).

Encrypted connections to the cloud must use an OpenSSL-based connection type (on both Windows and Linux). WinINet is not supported for cloud connectivity.

6.3 Credential Management

By default, the Enterprise Message API will store all credential information. In order to use secure credential storage, a callback function can be specified by the user. If a callback function is specified, credentials are not stored in API; instead, application is called back whenever credentials are required.

If an **OmmOAuth2ConsumerClient** is specified when creating the **OmmConsumer** object, the API will callback **OmmOAuth2ConsumerClient.onCredentialRenewal** whenever credentials are required. This call back must call **OmmConsumer.renewOAuthCredentials** to provide the updated credentials.

NOTE: **OmmConsumer.renewOAuthCredentials** can only be called during the callback.

6.4 Version 1 Authentication Using oAuth Password and Refresh_Token

6.4.1 Client_ID (AppKey) and Client Secret

To connect to Real-Time - Optimized infrastructure, the Enterprise Message API requires a **Client_ID**, and optionally can include a client secret. **Client_IDs** are generated using **AppGenerator**, which refers to the **Client_ID** as an AppKey. Each user must obtain their unique **Client_ID** using the machine account email sent by LSEG, which includes a link to **AppGenerator**. Keep your **Client_ID** private: do not share **Client_IDs**.

- For further details on generating this ID, refer to the *Real-Time - Optimized: Installation and Configuration for Client Use* document. Each **Client_ID** is unique: do not share it with others.
- For further details on supporting client secret submissions, refer to the.
- For details on how OAuth uses a Client Secret with a Client ID and their relationship, refer to OAuth documentation at: the following URL: <https://www.oauth.com/oauth2-servers/client-registration/client-id-secret/>.

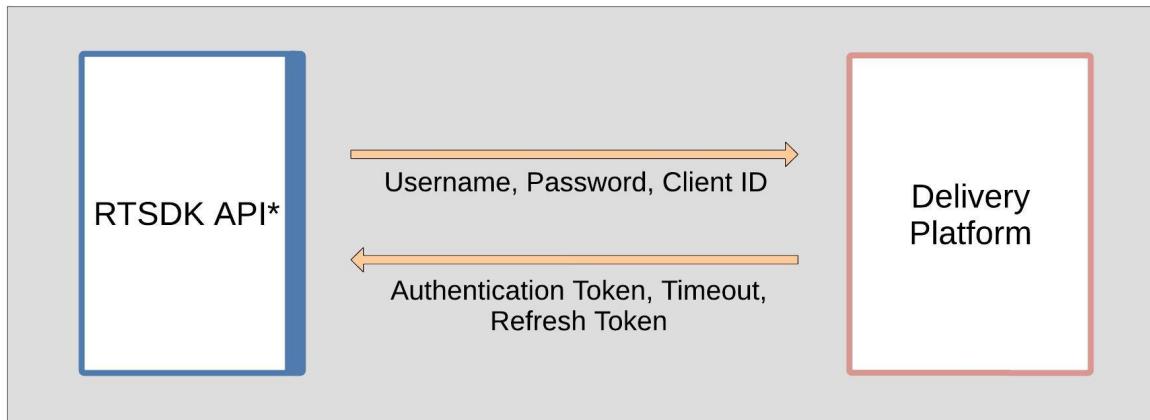
6.4.2 Obtaining Initial Access and Refresh Tokens

To obtain an access token, the RTSDK API sends its username, **Client_ID**, and password in a single message to the Delivery Platform.

 **TIP:** You can also specify **tokenScope** and **clientSecret** in the OMMConsumerConfig.

In response, the Delivery Platform sends an access token, its expiration timeout (by default: 300 seconds), and a refresh token for use in the login reissue process (for details on the expiration timeout and login reissue process, refer to Section 6.4.3). The API must obtain an access token before executing a service discovery or obtaining market data.

The following diagram illustrates the process by which the RTSDK API obtains its tokens:



*: An RTSDK API can be a consuming application written to EMA or ETA in C/C++, Java or C#.

Figure 2. Obtaining an Authentication Token

6.4.3 Refreshing the Access Token and Sending a Login Reissue

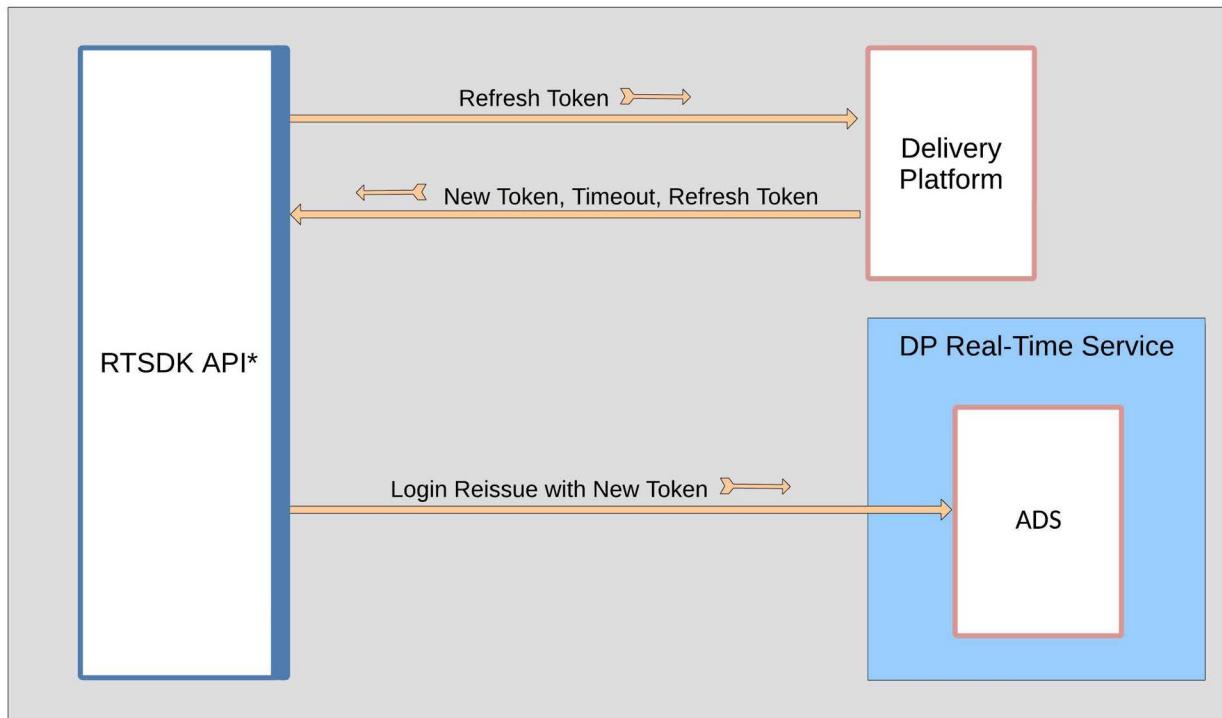
In response to the API's token request, the Delivery Platform sends an access token and a refresh token, both with associated expiration timeouts which set the length of time for which the token is valid. If the LSEG Real-Time Advanced Distribution Server does not receive a new access token before the end of the expiration timeout, the LSEG Real-Time Advanced Distribution Server sends a login close status message and closes the connection.

NOTE: The life cycle of `OmmConsumer` in the Enterprise Message API depends on the state of the login stream because the Enterprise Message API closes the underlying channel whenever the API receives a close status message from LSEG Real-Time Advanced Distribution Server. To recover from this scenario, the application must create another `OmmConsumer` and resubscribe to all applicable items.

To create a seamless experience for API users, the API sends the refresh token to proactively obtain a new access token prior to the published expiration timeout. The Enterprise Message API calculates the time at which it requests a new access token by multiplying the token's published timeout by 4/5 (i.e., **0.8**).

In response to receiving a refresh token, the Delivery Platform sends a new access token with an associated timeout to the API. After receiving the new access token from the Delivery Platform, the API renews its connection by sending a Login Reissue with the new access token to the LSEG Real-Time Advanced Distribution Server. The process of renewing the access token and refreshing the LSEG Real-Time Advanced Distribution Server connection via a Login Reissue continues until the refresh token itself expires (which can take several hours or days). When using a `grant_type` of `refresh_token`, if the value for `expires_in` does not match the `expires_in` received from when the API obtained the `refresh_token` (i.e., when `grant_type` was `password`), this is an indication that the `refresh_token` is about to expire. In this case, the API will obtain a new set of both refresh and access tokens as described in Section 6.4.2.

The login reissue process is illustrated in the following diagram:



*: An RTSDK API can be a consuming application written to EMA or ETA in C/C++, Java or C#
RDP: Delivery Platform
ADS: LSEG Real-Time Advanced Distribution Server

Figure 3. Login Reissue

6.5 Version 2 Authentication Using OAuth Client Credentials

Version 2 authentication is available with two types of accounts:

- OAuth Client Credentials which requires a client ID and client secret.
- OAuth Client Credentials with JWT which requires client ID and private JWK for JWT.

Version 2 will generate an Access Token. Once connected to Real-Time — Optimized RTC, the login session to the LSEG Real-Time Connector (RTC) will remain valid until the consumer disconnects or is disconnected from Real-Time — Optimized. The API will only re-request an Access Token in the following cases:

- When the consumer disconnects and goes into a reconnection state.
- If the **Channel** stays in reconnection long enough to get close to the expiry time of the Access Token.

Due to the above changes, credentials are managed independently per reactor channel. Channels do not share credentials.

6.5.1 Configuring and Managing Version 2 Credentials

The client ID and client secret or private JWK must be set on the **OmmConsumer** object as described in Section 6.10.2.1 of the *Enterprise Transport API C++ Edition Value Added Developers Guide*. The **OmmOAuth2ConsumerClient** will handle the credentials the same way as Version 1, with an **OmmOAuth2ConsumerClient** callback for credentials if the user does not wish for the **OmmOAuth2ConsumerClient** to store them.

6.5.1.1 JWT Credentials Handling

Version 2 OAuth Client Credentials with JWT requires a JWK public/private pair to be generated and registered with LSEG via the Platform Admin UI. The API will use a private JWK to create and sign a JWT request, which will be sent to retrieve an access token. The JWK will be handled by the API the exact same way as a client secret above. For more information about the Platform Admin UI, refer to the Real-Time — Optimized documentation in the LSEG Developers portal.

NOTE: Follow best practices for securely storing and retrieving JWK.

6.5.2 Version 2 OAuth Client Credentials Token Lifespan

Unlike Version 1, Version 2 will only produce a single Access Token, which will be valid for the length of the entire **expires_in** field in the token. This Access Token is used by the API to perform service discovery, and to connect to Real-Time — Optimized.

Once connected, the API does not need to periodically renew a token.

The API will re-request a token on reconnect, and will use that token for all reconnect attempts until a short time prior to expiry. At that time, the API will get a new token for reconnection use.

6.6 Service Discovery

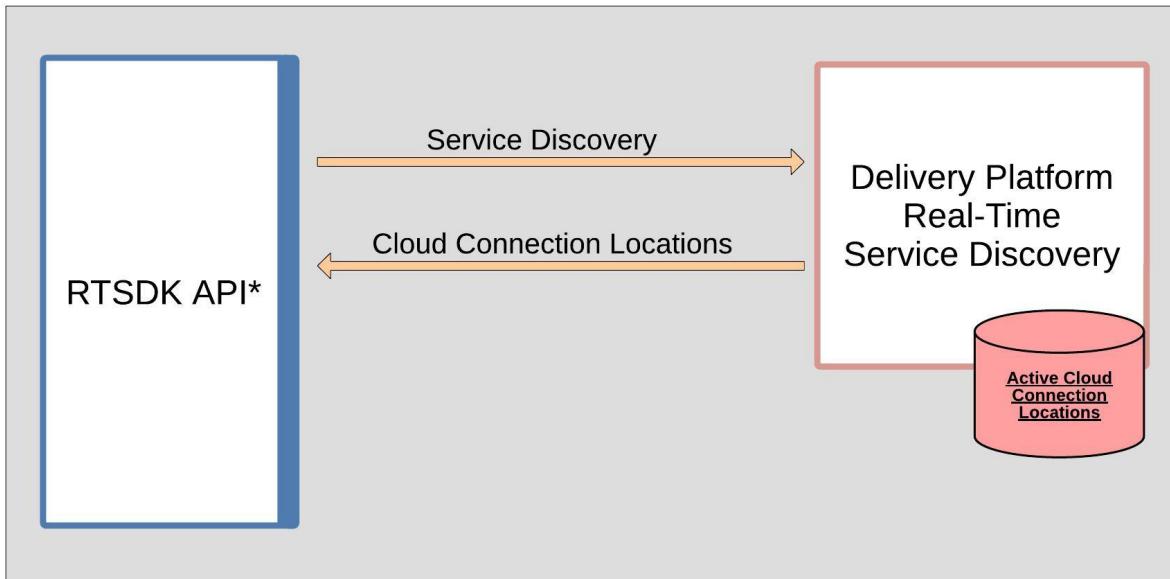
After obtaining a token (for details, refer to Section 6.4.2), the Enterprise Message API can perform a service discovery against the Delivery Platform to obtain connection details for the Real-Time — Optimized. To discover endpoints, application may rely either on file or programmatic configuration. This is accomplished by making a REST query to the Service Discovery service. EMA API may be configured to perform this query and choose an endpoint (host and port) in a specified region. Or, EMA application may interact with a pre-defined service discovery object (see **ServiceEndpointDiscovery**) to customize choosing endpoint(s).

For service discovery performed by API, see Cons113 example. For service discovery performed in application, see Cons450 example.

In response to a service discovery, the Delivery Platform returns transport and data format protocols and a list of hosts and associated ports for the requested service(s) (i.e., an LSEG Real-Time Advanced Distribution Server running in the cloud or endpoint). LSEG provides multiple cloud locations based on region, which is significant in how the Enterprise Message API chooses the IP address and port to use when connecting to the cloud.

From the list sent by the Delivery Platform, the Enterprise Message API identifies a Real-Time — Optimized endpoint with built-in resiliency whose regional location matches the API's location setting in **ChannelGroup** (for details, refer to Section 3.3.2 "Universal Channel Entry

Parameters" of the *Enterprise Message API C++ Edition Configuration Guide*). If you do not specify a location, the Enterprise Message API defaults to the **us-east-1** cloud location. An endpoint with built-in resiliency lists multiple locations in its location field (e.g., **location: [us-east-1a, us-east-1b]**). If multiple endpoints are configured for failover, the Enterprise Message API chooses to connect to the first endpoint listed.

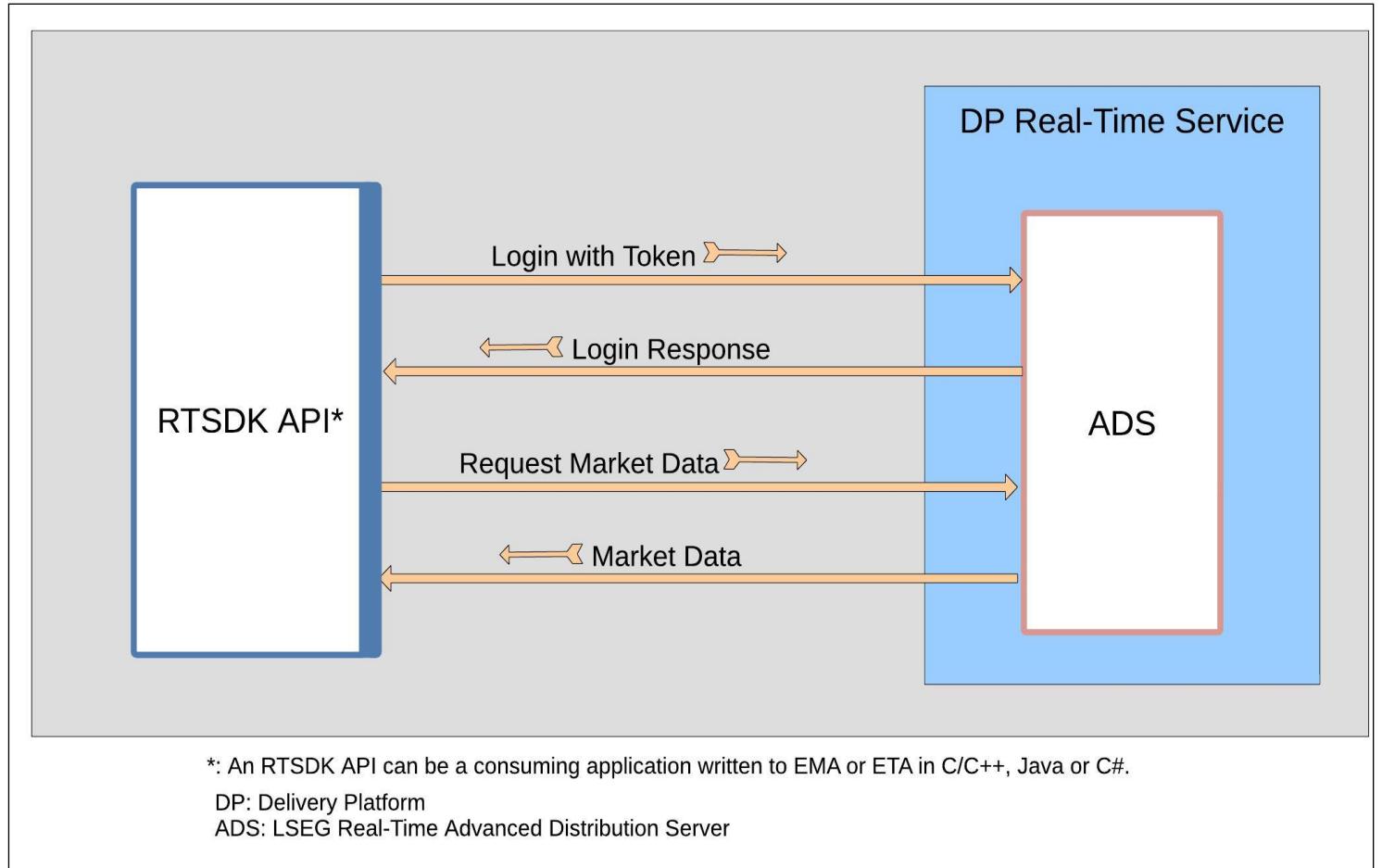


*: An RTSDK API can be a consuming application written to EMA or ETA in C/C++, Java or C#.

Figure 4. Service Discovery

6.7 Consuming Market Data

After obtaining its login token (for details, refer to Section 6.4.2) and running a service discovery (for details, refer to Section 6.6), the API can connect to the LSEG Real-Time Advanced Distribution Server in the cloud and obtain market data. While consuming market data, the API must periodically renew its token via the login reissue workflow (for details, refer to Section 6.4.3).



6.8 HTTP Error Handling for Reactor Token Reissues

The Enterprise Message API supports handling for the following HTTP error codes from the API gateway:

- 300 Errors:
 - Perform URL redirect for 301, 302, 307 and 308 error codes
 - Retry the request to the API gateway for all other error codes
- 400 Errors:
 - For Version 1 authentication, retry with username and password for error codes 400 and 401
 - Stop retry the request for error codes 403, 404, 410, and 451
 - Retry the request to the API gateway for all other error codes
- 500 Errors:
 - Retry the request to the API gateway for all error codes

6.9 Cloud Connection Use Cases

You can connect to the cloud and consume data according to the following use cases:

- Start to finish session management (for details, refer to Section 6.9.1)
- Explicit service discovery option for applications (for details, refer to Section 6.9.2)

6.9.1 Session Management Use Case

In this use case, the Enterprise Message API manages the entire connection from start to finish. To use session management, you need to configure the API to enable session management. To do so, in the ChannelGroup, set the Channel entry parameter **EnableSessionManagement**.

The API exhibits the following behavior for this use case:

1. Obtains a token (according to the details in Section 6.4.2).
2. Queries service discovery (according to the details in Section 6.6).
3. Consumes market data (according to the details in Section 6.7).

Manages login reissues for Version 1 authentication when needed on a cyclical basis (according to the details in).Enterprise Message API's Consumer example (**113_MarketPrice_SessionManagement** example) provides sample source to illustrate session management.

With session management enabled, application may specify a host and port in ChannelGroup parameters. In this case, the Enterprise Message API exhibits the same behavior listed above, but ignores the endpoints it receives from the service discovery and connects to the specified host and port.

6.9.2 Query Service Discovery

Application has the option to do a service discovery, parse the results, and choose an endpoint to pass into API. The API exhibits the following behavior when application does an explicit service discovery:

1. Obtains a token (according to the details in Section 6.4.2).
2. Queries service discovery (according to the details in Section 6.6).

Enterprise Message API's **Consumer** example (**450_MarketPrice_QueryServiceDiscovery**) provides sample source that discovers an endpoint using the service discovery feature and establishes an encrypted connection to consume data.

6.10 Logging of Authentication and Service Discovery Interaction

If needed, you can log interactions with the Delivery Platform. To enable logging, use the parameters `RestEnableLog` and `RestLogFileName` in the EMA configuration file or programmatic configuration in the Consumer Group. If Service Discovery is done from the application, logging may be enabled only via function call configuration. For details on these parameters, refer to the *Enterprise Message API C++ Configuration Guide*.

6.10.1 Logged Request Information

With logging turned on in the fashion mentioned in Section 6.10, the Enterprise Message API writes the following request information in the log:

Request:

- Time stamp
- The Name of the class and method that made the request
- Request method
- URI
- Request headers
- Proxy information (if used)
- Body of request as set of pairs parameter_name: parameter_value

NOTE: If the request contains parameters `password`, `newPassword`, or `client_secret`, the Enterprise Message API uses a placeholder instead of the real value of the respective parameter (thus indicating that the value was present).

6.10.2 Logged Response Information

With logging turned on in the fashion mentioned in Section 6.10, the Enterprise Message API writes the following response information in the log:

Response:

- Time stamp
- The Name of the class and method that received the response
- Response status code
- Response headers
- Body of response in string format

7 Warm Standby Feature

7.1 Overview

The Warm Standby feature, a client-side feature, is implemented at the Value Add Watchlist layer of Enterprise Transport API (ETA) and made available via Enterprise Message API with configuration. This feature works by providing the application the capability to failover from an active to one or more standby server(s) in the event that the primary/active fails. Application must configure the active and standby servers to use this API feature. After the connections are established with the provided servers which form a Warm Standby group, the client-side or consumer sends messages to the standby server connections to change their mode to Standby. Requested items are opened on all servers by the consumer but the active server responds with messages such as refresh, updates, status, etc. to the consumer. Standby servers respond with blank/empty refreshes. When primary fails, consumer notifies the next server in standby list that it is now Active. The new active server responds with refresh as needed resumes updates for all open items. This process of cut-over is transparent to the application.

A server qualifies to be a standby only if it advertises support for Warm Standby, supports similar features over login and offers an identical service (supported domains, quality of service, etc.) as the active server.

Warm Standby not only reduces overall recovery time, but also network traffic by not inducing a “packet storm” with a flurry of re-requests to a standby server. Because the standby server is already aware of items an application has subscribed for, during a failover Enterprise Message API does not need to re-subscribe open items between a provider and consumer.

7.2 Warm Standby Modes

The Enterprise Message API Value Add layer supports two Warm Standby modes:

- Login based Warm Standby
- Service based Warm Standby

The login based Warm Standby uses the connection lost event to switch from a primary server to a standby server from the standby server list. The service based Warm Standby uses the service down event OR connection lost event to switch all subscribe items from a primary service to a standby service.

The service based Warm Standby mode offers better resiliency than the login based mode as it can switch from primary to standby if an upstream service is down but the connection to both servers remains intact. A particular server may be the primary for one service and standby for another service as a result. This ability to failover in the event of service down or channel down events makes the service based Warm Standby the recommended mode.

The following figure illustrates the sequence of events when using the Login Based Warm Standby feature:

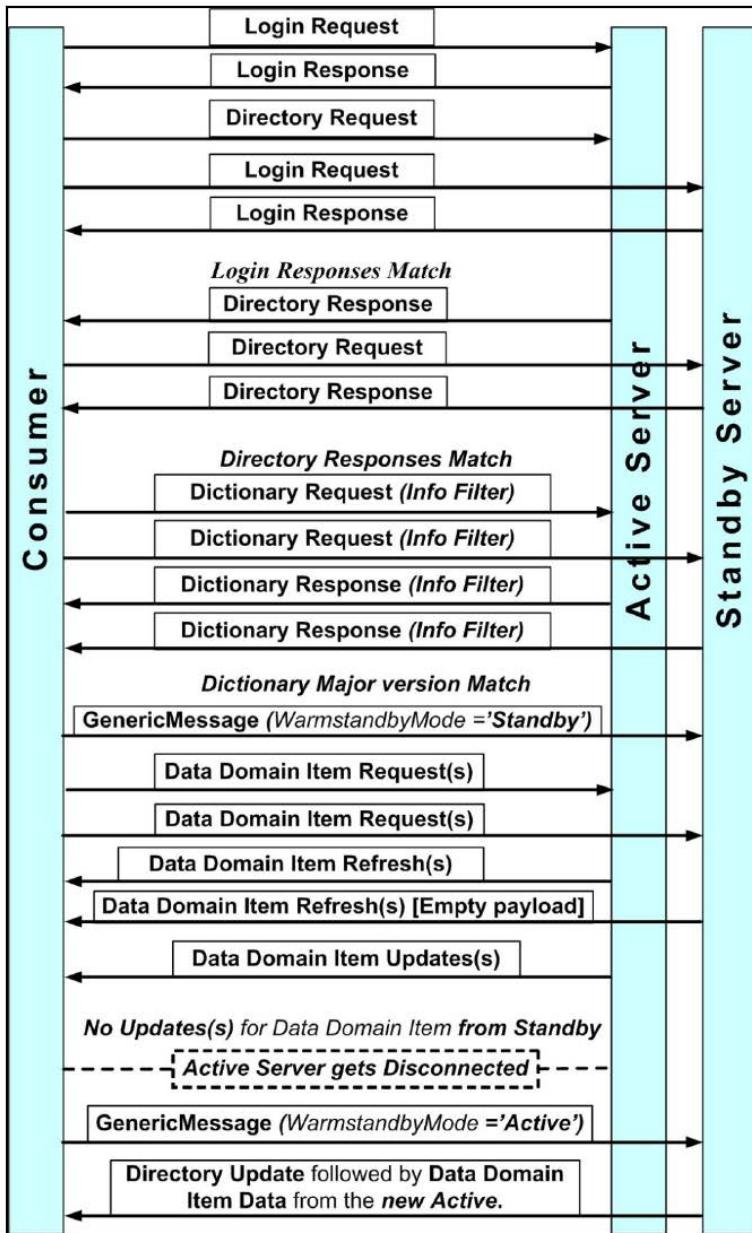


Figure 5. Login Based Warm Standby Order of Events in a Cutover from Active to Standby

The following figure illustrates the sequence of events when using the Service Based Warm Standby feature:

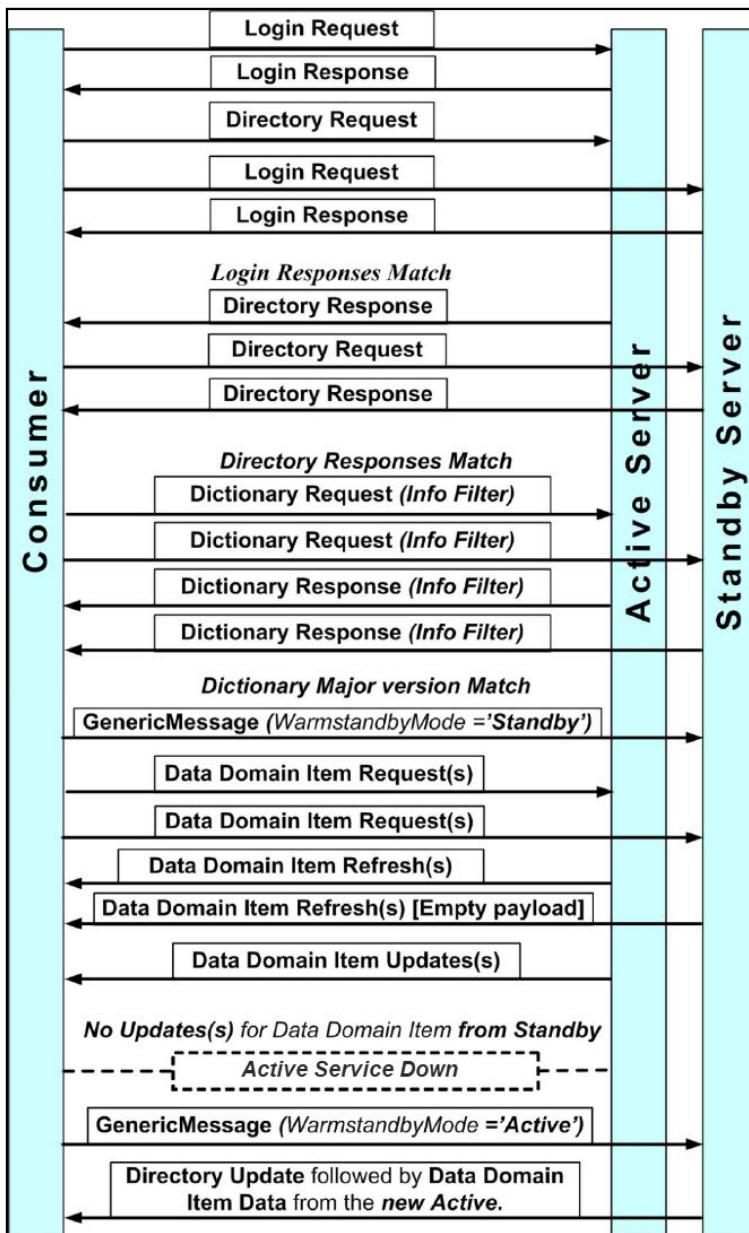


Figure 6. Service Based Warm Standby Order of Events in a Cutover from Active to Standby

7.3 Warm Standby Configuration and Feature Details

For details, refer to the *Enterprise Message API C++ Edition Configuration Guide*.

8 Preferred Host Feature

8.1 Overview

The Preferred Host feature is supported with the ChannelSet and WarmStandby features. This feature allows consumer applications to configure a specific host or warm standby group as “preferred”, and upon either a timer-based (using either a static timer or a cron string) or method-based trigger, cause the API library to perform a single connection *attempt* to the preferred host without cutting any currently active connections. Once a connection has been established to the preferred host, the API library will switch the connections and alert the user. If the connection attempt fails, the library will signal that the operation is complete and will not make any changes to the current connection. Also, with this feature enabled, upon connection recovery, depending on configuration, the library will attempt to connect to the preferred host or warm standby group.

The Preferred Host configuration for an EMA channel can be changed at any time through IOCTL calls, including the ability to disable or enable this feature.

For details on the Preferred Host feature and associated configuration parameters, refer to the *Enterprise Message API Configuration Guide*.

8.2 Preferred Host Reconnection Behavior Changes

When the Preferred Host feature is enabled, the reconnection order is changed to attempt the configured preferred host connection and warm standby group (if enabled) more aggressively by alternating between a configured preferred connection and a non-preferred connection. For more information about specific ordering and differences from non-preferred host reconnection, see section “Reactor Channel Reconnection and Recovery Behaviors” in the *Enterprise Transport API Value Add Developers Guide*.

8.3 Preferred Host Operation Steps

When the Preferred Host feature is enabled and the preferred host operation is triggered, the application continues to receive data from the current connections while the preferred host operation is occurring. The following sections describe possible scenarios.

8.3.1 ChannelSet Behaviors with Preferred Host Options Enabled

1. If the Channel is already on the preferred channel:
 - a. After finishing the preferred host operation, the library will send Login StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, text “Preferred host complete”, and code **OmmState.SocketPHNoFallback**, if the EMA application is registered to receive Login administrative domain messages.
2. If the Channel is *not* on preferred channel in the ChannelSet configuration:
 - a. The library will attempt to establish a connection to the configured preferred channel in the ChannelSet.
 - b. Once that is established, the library will do the following:
 - i. Send a StatusMsg with dataState **OmmState::SuspectEnum**, streamState **StreamState::OpenEnum**, and text “channel down” to all open items.
 - ii. Swap the underlying transport channels, and then send a Login StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, and text “channel up” to the EMA application, if it is registered to receive Login administrative domain messages.
 - iii. After the former non-preferred channel is fully closed by the library, the library will send a StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, text “Preferred host complete”, and code **OmmState.SocketPHComplete** to the application for all open items.

8.3.2 Warm Standby Configuration with Preferred Host Options Enabled

1. If the Channel is already on the preferred warm standby group (in WarmStandby config):
 - a. After finishing preferred host operation, the library will send Login StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, text "Preferred host complete", and code **OmmState.SocketPHNoFallback**, if the EMA application is registered to receive Login administrative domain messages.
2. If using warm standby configuration and not on preferred group and **PHFallbackWithInWSBGroup** is false or disabled:
 - a. The library attempts to establish a connection to the configured preferred warm standby group's starting connection. Once that is established, the following occurs:
 - i. The library closes all currently active standby connections and the starting connection and generates item and source directory (if application requested directory) status messages. Open items receive StatusMsg with dataState **OmmState::SuspectEnum**, streamState **StreamState::OpenEnum**, and text "channel down", indicating that the warm standby group is fully closed and that the library is switching to the preferred warm standby group.
 - ii. The library will then swap the underlying transport channels, and a Login StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, and text "channel up" is sent to EMA application if it is subscribed to Login administrative domain.
 - iii. The library will send a Login StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, and text "Preferred host complete", if the EMA application is registered to receive Login administrative domain messages.
 - iv. Once the library internally receives login and directory responses, it will connect to the configured secondary servers in the warm standby group.
 - b. If the connection attempt to preferred group's starting active server fails, the library will notify the application that the preferred host operation is complete with a Login StatusMsg with dataState **OmmState::OkEnum**, streamState **StreamState::OpenEnum**, text "Preferred host complete", and code **OmmState.SocketPHComplete**. The library stays connected to current connections and any flowing data continues to flow.
3. If using warm standby configuration and not on preferred group and **PHFallbackWithInWSBGroup** is true or enabled:
 - a. The fallback within a warm standby group will occur, and the Channel will not attempt to connect to a different warm standby group.
 - b. The following operations will be done depending on the type of the current warm standby group configuration:
 - For a login-based warm standby group: If the starting server connection is active and not the current active connection for the warm standby group, the library will swap the current active server to the starting server connection. The application may see unsolicited refreshes to re-synchronize the item streams.
 - For a service-based warm standby group (or login and service-based): The library will iterate through the configuration of the warm standby group, starting with the starting server, and going through each connection defined in the secondary server list. For each service name defined in **PerServiceNameSet**, if the service name is in an ACTIVE state on that connection, that connection will become the ACTIVE for that service, and the previous active will become STANDBY. Note that if service names are not defined, the behavior is the same as login-based.

If a service name is defined multiple times in the warm standby group, the first time it is found as ACTIVE on a connection will be used and ignored on subsequent matches.

Any services that are not defined in the configuration will be ignored by this operation, and the current ACTIVE for those services will not be changed.

9 Request Routing

The Request Routing feature, a client-side feature, provides the application the capability to route market data item requests to multiple connections depending on the availability of services of each connection. EMA routes item requests by matching the quality of service and capabilities of the requested concrete service or service list name in order to submit the request messages. EMA manages multiple connections, aggregates login responses and concrete services on behalf of application. The same concrete service name of each connection must have the same quality of service list, item list and support QoS range attributes otherwise the subsequent connection is closed and removed. Application can define a service list which contains a list of concrete service names for requesting market data items using the service list name which requests are routed to the first qualifying service on the service list. For item recovery, EMA recovers items from the concrete services in the consumer session which matches with the requested concrete service name or one of concrete service in the specified service list and the service is ready to accept requests.

For details, see the following sections:

- [Administrative Domains Behaviors](#)
- [Service List](#)
- [Item Request Routing and Recovery](#)
- [Posting Messages](#)
- [Sending Generic Message](#)
- [Session Channel Information from OmmConsumer and OmmConsumerEvent](#)

9.1 Administrative Domains Behaviors

In order to create a consumer session for request routing, multiple connections must be aggregated. Handling administrative domains for aggregating multiple connections has some specific behaviors.

For details, see the following sections:

- [Login Request Timer Handling and Login Response Aggregation](#)
- [Aggregated Login Elements](#)
- [Scenarios for Receiving Aggregated Login Stream](#)
- [Directory Request Timer Handling and Directory Response Aggregation](#)
- [Dictionary Request Timer Handling](#)

9.1.1 Login Request Timer Handling and Login Response Aggregation

The **LoginRequestTimeOut** parameter is configured in ConsumerGroup to control how long would EMA waits to receive login response for all configured session channels during OmmConsumer initialization. The following are scenarios for handling multiple connections for a consumer session.

SCENARIOS FOR LOGIN REQUEST TIMEOUT	LOGIN INITIALIZATION BEHAVIOR
No connections come up or EMA does not receive any login response within timeout.	<ul style="list-style-type: none"> Initialization fails at timeout. Application must re-create the OmmConsumer object. The OmmConsumer object will not be created.
Only some connections are up and sent login response, the login timer fires.	<ul style="list-style-type: none"> Initialization succeeds with the login domain at timeout. <p>NOTE: Initialization succeeds with the login domain at timeout. Channels that did not come up will be closed: if application wants to route to channels that did NOT come up in time, application must re-create OmmConsumer.</p> <ul style="list-style-type: none"> Application is notified of which connections are up/down upon timeout via login stream. Requests are routed in the ordered list of connections in the Consumer's SessionChannelSet configuration.
EMA receives login denied status message for a connection.	Enterprise Message API closes the connection and removes the channel from the session channel list.
All configured connections come up before the login timeout fires.	<ul style="list-style-type: none"> Initialization succeeds with the login domain immediately upon all connections up. Application is notified of which connections are up/down upon all connections up. Requests are routed in the ordered list of connections in the Consumer's SessionChannelSet configuration.

Table 3: Scenarios for Login Request Timeout

9.1.2 Aggregated Login Elements

Enterprise Message API aggregates all login responses from all connections and presents to application as a login response once login domain handling is completed. Login aggregation takes the login element attributes that are present across all of the connections.

The following is the list of aggregated login elements. For each element, all connections must support login aggregation, otherwise it will not be supported.

- ProvidePermissionProfile
- ProvidePermissionExpressions
- SupportBatchRequests
- SupportOptimizedPauseResume
- SupportPauseResume
- SupportOMMPost
- SupportEnhancedSymbolList
- SupportViewRequests
- RoundTripLatency

NOTE: SingleOpen and AllowSuspect attributes are always turned on for the Request Routing feature.

9.1.3 Scenarios for Receiving Aggregated Login Stream

Applications can register to receive an aggregated login stream and channel information via login stream for each session channels in order to receive notification of channel events. The aggregated login message's **OmmState** is used to represent the entire consumer session as described in the following table.

AGGREGATED LOGIN STREAM SCENARIOS	MESSAGE TYPE	STREAM STATE	DATA STATE	DESCRIPTION
EMA tries to establish a connection with all session channels for a consumer session within the timeout.	StatusMsg	Open	Suspect	Application receives channel information for each session channel via login stream event so that application get notified which session channel is up or down or down reconnecting. This should only be sent once for each underlying reactor channel, upon initialization of the reactor channel.
EMA receives a "login accepted" status message from at least one session channel in a consumer session and login timeout fires.	RefreshMsg	Open	Ok	Enterprise Message API is successfully establishing a login stream for a OmmConsumer . Enterprise Message API notified application with aggregated login response.
EMA receives a "login closed" status message from a session channel, but others login stream is still open.	StatusMsg	Open	Ok	There is no change to the aggregated login stream and application receives channel down event for the session channel.
EMA receives a "login closed" status message from every session channel.	StatusMsg	Closed	Suspect	EMA notifies application with a Login Status with OmmState of Closed/Suspect upon detecting that <i>all</i> channels in a SessionChannelSet are down either due to receiving a Login Status Closed/Suspect or channel down event. See the previous row for behavior if only some of the session channels are closed. In this case, OmmConsumer is in terminal state and application should uninitialized it.
EMA receives a channel down event from every session channel.	StatusMsg	Closed	Suspect	EMA notifies application with a Login Status with OmmState of Closed/Suspect indicating channel down upon detecting that <i>all</i> channels in a SessionChannelSet are down either due to receiving a Login Status Closed/Suspect or channel down event. See two rows for behavior if only some of the session channels are closed. In this case, OmmConsumer is in terminal state and application should uninitialized it.

Table 4: Aggregated Login Stream Scenarios

9.1.4 Directory Request Timer Handling and Directory Response Aggregation

The **DirectoryRequestTimeOut** parameter in **ConsumerGroup** is used to control how long would Enterprise Message API waits to receive directory response for all configured session channels during **OmmConsumer** initialization. Initialization fails at timeout when Enterprise Message API does not receive at least one source directory valid response from a channel.

For source directory aggregation, Enterprise Message API generates a unique service ID for each particular service in order to aggregate particular services with the same name from multiple connections. Applications can register with the source directory domain to receive source directory response for aggregated services in order to retrieve source directory information including the generated service ID. Enterprise Message API aggregates and caches for particular services only for INFO and STATE filters for routing user's requests.

For each particular service name, the following element attributes must match.

- List of supported quality of service (QoS)
- Item List name
- Support QoS range

NOTE: If element attributes do not match the service name during OmmConsumer creation, Enterprise Message API removes and closes the subsequent connection based on the order of the session channel list. Afterward, it ignores source directory update with mismatched attributes for the service.

For service's state aggregation of a particular service:

- **ServiceState** is UP (1) unless the service of all connections is in the DOWN (0) state.
- **AcceptingRequests** is UP (1) unless the service of all connections in the DOWN (0) state.

9.1.5 Dictionary Request Timer Handling

The **DictionaryRequestTimeOut** parameter in **ConsumerGroup** is used to control how long would Enterprise Message API waits to receive dictionary response for the **ChannelDictionary** type from a channel during **OmmConsumer** initialization. Initialization fails at the timeout when Enterprise Message API does not receive a dictionary response from a session channel which provides the first available service for downloading data dictionary.

NOTE: Enterprise Message API supports single data dictionary version per **OmmConsumer**, so all services of all session channels must support the same version.

9.2 Service List

A Service List is a named grouping of particular service names in order to request items using service list name. The **ServiceList** class is used to create a service list, and applications have to add **ServiceList** instances to an instance of **OmmConsumerConfig** before creating **OmmConsumer**.

NOTE: The concrete services in a service list must have the same quality of service.

The following example illustrates the creation of a service list and adding it to **OmmConsumer**.

```
/* Create a service list which can subscribe data using any concrete services in this list */
ServiceList serviceList("SVG1");

serviceList.concreteServiceList().push_back("DIRECT_FEED");
serviceList.concreteServiceList().push_back("DIRECT_FEED_2");

OmmConsumer consumer( OmmConsumerConfig().addServiceList(serviceList));

/* Request an item using the service list */
consumer.registerClient( ReqMsg().serviceName( "SVG1" ).name( "LSEG.L" ), client );
```

NOTE: When making a request using the service list name, the response messages will utilize the name and ID of the service list name instead of the concrete service name and ID.

9.3 Item Request Routing and Recovery

Applications can set an item request using a concrete service name, service ID, or service list name in a `ReqMsg` instance and register it using the `registerClient()` method of `OmmConsumer`. For specifying service name/ID, Enterprise Message API tries to match the specified concrete service with a session channel according to the ordered list of connections in the `OmmConsumer`. For service list name, Enterprise Message API tries to match between the ordered list of concrete service names and the ordered list of connections in the `OmmConsumer`.

The following diagram illustrates the request matching workflow for a concrete service and a session channel.

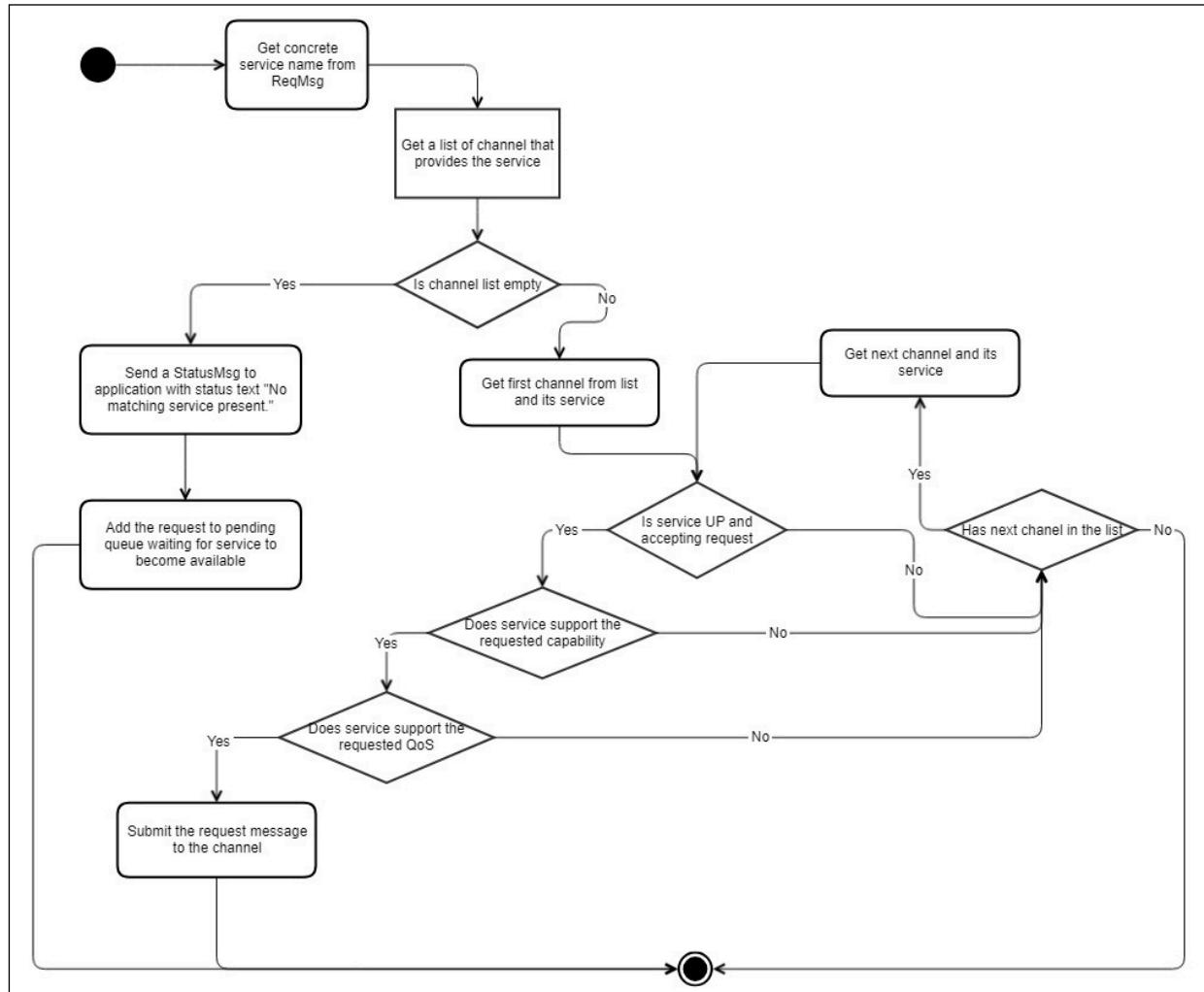


Figure 7. Item Request Matching Workflow

Enterprise Message API conducts automatic item recovery for all streaming item requests initiated by the client, with the exception of private item streams. The situations that lead to an item's data becoming stale include loss of connection, service outages, and the provider transmitting the `CLOSED_RECOVER` stream state to the consumer, among others. The process of item recovery involves recovering items from a stale condition by closing the existing item stream and utilizing the item request matching workflow to identify a session channel for re-requesting the item. Request Routing also works with the WarmStandby feature which has its own fallback mechanism to receive data from a standby server when the active server or service is unavailable. Items that cannot be retrieved due to the unavailability of a service for data requests will be placed in a recovery queue for future retrieval at an appropriate time. The items that are in the `CLOSED` stream state will be recovered in others session channel until there is no other channel, then the closed status message will be sent to application.

NOTE: The `SessionEnhancedItemRecovery` parameter is set to true by default to recovery items to others session channels when the session channel is temporary down due to connection loss. Users can override this behavior by setting the parameter to false to let Enterprise Message API wait until the connection is recovered and request data with the requested service name.

9.4 Posting Messages

In order to perform off-stream posting, the aggregated login refresh for all session channels must support the OMM post feature, otherwise the **OmmInvalidUsageException** is thrown to application. The service name or service ID specified in a **PostMsg** will be translated into the underlying session channel's service ID. The **AckMsg** will be transmitted through the session channel which it was received.

For off-stream posting, Enterprise Message API will distribute the **PostMsg** to all connected session channels that are compatible with the designated service. Enterprise Message API will discard the **PostMsg** from any session channel that does not support the specified service name or ID.

For on-stream posting, Enterprise Message API will submit the **PostMsg** to the session channel of the requested stream that supports the specified service. The **OmmInvalidUsageException** will be raised to the application if the specified service name or ID is not present on the requested stream.

9.5 Sending Generic Message

The application is capable of submitting a **GenericMsg** on both the login and item streams. In this process, the specified service ID within the **GenericMsg** will be converted into the service ID of the underlying session channel. However, if the service ID is unknown, this will not trigger an **OmmInvalidUsageException**, and the unknown service ID will be transmitted as it is.

For sending a **GenericMsg** on login stream, Enterprise Message API will distribute the GenericMsg to all connected channels without regard to the validity of the specified ID.

9.6 Session Channel Information from OmmConsumer and OmmConsumerEvent

Application can get a list of session channels from the **sessionChannelInfo()** method from either **OmmConsumer** or **OmmConsumerEvent** classes. This would help application to get a list of active session channels in order to keep track each session's channel state. The session channel will be removed from the list once it is closed by Enterprise Message API, or the channel is completely down. Users must specify a List of **ChannelInformation** in order to return the current session information per function call. The following is the sample code how to iterator through the list of **ChannelInformation** from the function.

```
EmaVector<ChannelInformation> statusVector;
event.getSessionInformation(statusVector);
// Print out the channel information.
for (UInt32 i = 0; i < statusVector.size(); ++i)
{
    cout << statusVector[i] << endl;
}
```

NOTE: The **sessionChannelInfo()** method returns an empty list if request routing is not enabled.

Moreover, the **ChannelInformation** class can also get a channel name and session channel name using the **channelName()** and **sessionChannelName()** method respectively.

10 Troubleshooting and Debugging

10.1 Enterprise Message API Logger Usage

The Enterprise Message API provides a logging mechanism useful for debugging runtime issues. In the default configuration, Enterprise Message API is set to log significant events encountered during runtime and direct logging output to a file. If needed, you can turn off logging, or direct its output to `stdout`. Additionally, applications can configure the logging level at which the Enterprise Message API logs event (to log every event, only error events, or nothing). For further details on managing and configuring the EMS logging function, refer to the *Enterprise Message API Configuration Guide*.

10.2 Omm Error Client Classes

10.2.1 Error Client Description

Enterprise Message API has two Error Client classes: `OmmConsumerErrorClient` and `OmmProviderErrorClient`. These two classes are an alternate error notification mechanism in the Enterprise Message API, which you can use instead of the default error notification mechanism (i.e., `OmmException`, for details, refer to Section 10.3). Both mechanisms deliver the same information and detect the same error conditions. To use Error Client, applications need to implement their own error client class, override the default implementation of each method, and pass this Error Client class on the constructor to `OmmConsumer` and `OmmProvider`.

10.2.2 Example: Error Client

The following example illustrates an application error client and depicts simple processing of the `onInvalidHandle()` method. In the following example, `ClassName` is either `OmmConsumerErrorClient` (for Enterprise Message API consumer applications) or `OmmProviderErrorClient` (for Enterprise Message API provider applications).

```

class AppErrorClient : public OmmConsumerErrorClient
{
public :
    void onInvalidHandle( UInt64 handle, const EmaString& text );
    void onInaccessibleLogFile( const EmaString& filename, const EmaString& text );
    void onMemoryExhaustion( const EmaString& text );
    void onInvalidUsage( const EmaString& text, Int32 errorCode );
    void onSystemError( Int64 code, void* ptr, const EmaString& text );
    void onJsonConverter( const EmaString& text, Int32 errorCode, const ConsumerSessionInfo&
sessionInfo );
    void onDispatchError( const EmaString& text, Int32 errorCode );
};

void AppErrorclient::onInvalidHandle( UInt64 handle, const EmaString& text )
{
    cout << "InvalidHandle: " << endl << "Handle = " << handle << endl << "text = " << text << endl;
}

...

void AppErrorclient::onDispatchError( const EmaString& text, Int32 errorCode );
{
    cout << "DispatchError: " << endl << "text = " << text << endl << "error = " << errorCode << endl;
}

```

10.3 OmmException Class

If the Enterprise Message API detects an error condition, the Enterprise Message API might throw an exception. All exceptions in the Enterprise Message API inherit from the parent class **OmmException**, which provides functionality and methods common across all **OmmException** types.

 **TIP:** LSEG recommends you use **try** and **catch** blocks during application development and QA to quickly detect and fix any Enterprise Message API usage or application design errors.

The Enterprise Message API supports the following exception types:

- **OmmInaccessibleLogFileException**: Thrown when the Enterprise Message API cannot open a log file for writing.
- **OmmInvalidConfigurationException**: Thrown when the Enterprise Message API detects an unrecoverable configuration error.
- **OmmInvalidHandleException**: Thrown when an invalid / unrecognized item handle is passed in on **OmmConsumer** or **OmmProvider** class methods.
- **OmmInvalidUsageException**: Thrown when the Enterprise Message API detects invalid interface usage.
- **OmmJsonConverterException**: Thrown when the Enterprise Message API fails to perform a RWF/JSON conversion.
- **OmmMemoryExhaustionException**: Thrown when the Enterprise Message API detects an out-of-memory condition.
- **OmmOutOfRangeException**: Thrown when a passed-in parameter lies outside the valid range.
- **OmmSystemException**: Thrown when the Enterprise Message API detects a system exception.
- **OmmUnsupportedDomainTypeException**: Thrown if domain type specified on a message is not supported.

10.4 Creating a DACSLOCK for Publishing Permission Data

Provider applications can create a DACSLocks and publish it to permission data on the LSEG Real-Time Distribution System. A DACSLock controls access to data by users. For further details on the DACSLock API, refer to the *Enterprise Transport API C Edition DACSLock Library*.

The following example code illustrates how to create a DACSLock.

```
#include "dacs_lib.h"

typedef struct {
    char          _operator;
    unsigned short pc_listLen;
    unsigned long  pc_list[256];
} PC_DATA;

PC_DATA pcData;
PRODUCT_CODE_TYPE* pcTypePtr = (PRODUCT_CODE_TYPE *)&pcData;
unsigned char* lockPtr = NULL;
int lockLen = 0;
DACS_ERROR_TYPE dacsError;
unsigned char dacsErrorBuffer[128];

printf("\nGenerates DACS lock \n");
pcData._operator = OR_PRODUCT_CODES;
```

```
pcData.pc_listLen = 1;
pcData.pc_list[0] = 1001;
int serviceId = 261;

if (DACS_GetLock(serviceId, pcTypePtr, &lockPtr, &lockLen, &dacsError) == DACS_FAILURE)
{
    if (DACS_perror(dacsErrorBuffer, sizeof(dacsErrorBuffer), (unsigned char *)"DACS_GetLock() failed
        with error", &dacsError) == DACS_SUCCESS)
    {
        printf("%s\n", dacsErrorBuffer);
    }
    else
    {
        printf("DACS_GetLock() failed\n");
    }
    return;
}
printf("DACS_GetLock() - Success\n");

EmaBuffer permissionData;
permissionData.setFrom((const char*)lockPtr, lockLen);
```

© LSEG 2015 - 2025. All rights reserved.

Republication or redistribution of LSEG Data & Analytics content, including by framing or similar means, is prohibited without the prior written consent of LSEG Data & Analytics. 'LSEG Data & Analytics' and the LSEG Data & Analytics logo are registered trademarks and trademarks of LSEG Data & Analytics.

Any third party names or marks are the trademarks or registered trademarks of the relevant third party.

Document ID: EMAC392L1UM.250
Date of issue: December 2025

