Enterprise Transport API C# Edition 3.4.0.L2

DEVELOPERS GUIDE

Document Version: 3.4.0.L2
Date of issue: May 2025
Document ID: ETACSharp340L2UM.250

LSEG DATA & ANALYTICS

Contents

Contents

Contents

1 Introduction

1.1 About this Manual

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

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

This document explains the configuration parameters for the Enterprise Messaging API (simply called the Message API). Message API configuration is specified first via compiled-in configuration values, then via an optional user-provided XML configuration file, and finally via programmatic changes introduced via the software.

Configuration works in the same fashion across all platforms.

1.2 Audience

This manual provides information and examples that aid programmers using the Enterprise Transport API C# Edition. The level of material covered assumes that the reader is a user or a member of the programming staff involved in the design, coding, and test phases for applications which will use the Enterprise Transport API. It is assumed that the reader is familiar with the data types, classes, operational characteristics, and user requirements of real-time data delivery networks, and has experience developing products using the C# programming language in a networked environment.

This manual provides information that aids software developers and local site administrators in understanding Enterprise Transport API configuration parameters. You can obtain further information from the *Enterprise Message C# Edition API Developer's Guide*.

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

1.3 Programming Language

The Enterprise Transport API Components are written to the C, Java, and C# languages. This guide discusses concepts related to the C# Edition. All code samples in this document and all example applications provided with the product are written accordingly.

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
DMM	Domain Message Model
Enterprise Message API	The Enterprise Message API (EMA) 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 (ETA) 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)
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.
RFA	Robust Foundation API
RMTES	A multi-lingual text encoding standard
RSSL	Source Sink Library

Table 1: Acronyms and Abbreviations

ACRONYM / TERM	MEANING
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.
SOA	Service Oriented Architecture
SSL	Sink Source Library
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 References

- Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*
- *API Concepts Guide*
- Enterprise Transport API C# Edition *Configuration Guide*
- Enterprise Transport API C# Edition *Developers Guide*
- Enterprise Transport API *ANSI Library Reference Manuals*
- Enterprise Transport API C# Edition*Value Added Components Developers Guide*
- Enterprise Transport API C# Edition *Performance Tools 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:

- Typographic
- Diagrams

1.7.1 Typographic

This document uses the following types of conventions:

- C# Structuresclasses, 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:

```
// decode contents into the filter list structure
            CodecReturnCode ret = fieldList.Decode(dIter, null);
            if (ret != CodecReturnCode.SUCCESS)
            {
                Console.WriteLine("DecodeFieldList() failed with return code: " + ret);
                return ret;
            }
consumer = new(new OmmConsumerConfig().OperationModel(OperationModelMode.USER_DISPATCH)
                                                    .Host("localhost:14002").UserName("user"));
```

```
consumer.RegisterClient(new RequestMsg().ServiceName("DIRECT_FEED").Name("IBM.N"),
       new AppClient(), 0);
var endTime = System.DateTime.Now + TimeSpan.FromMilliseconds(60000);
while(System.DateTime.Now < endTime)
{
    consumer.Dispatch(10);      // calls to OnRefreshMsg(), OnUpdateMsg(), or
       OnStatusMsg() execute on this thread
}
```

1.7.2 Diagrams

Diagrams that depict the interaction between components on a network use the following notation:

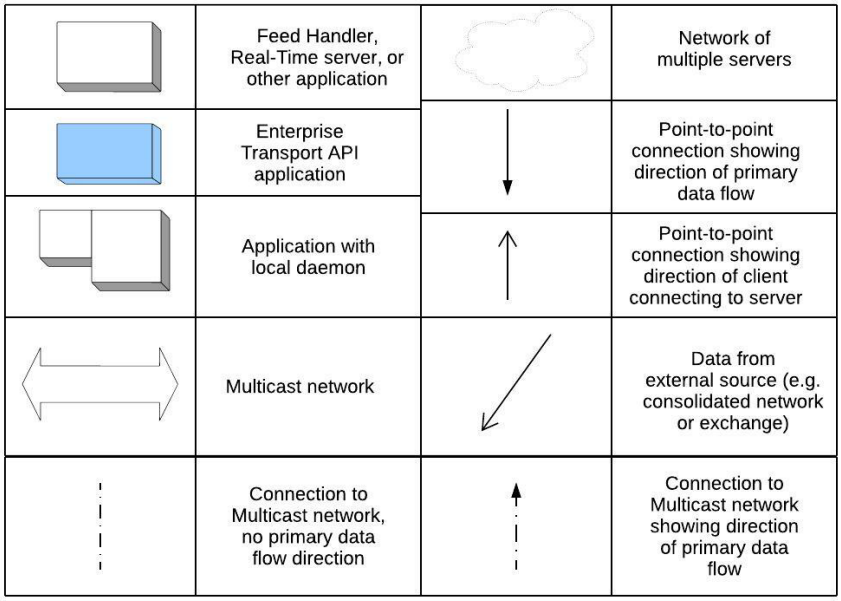


Figure 1. Network Diagram Notation

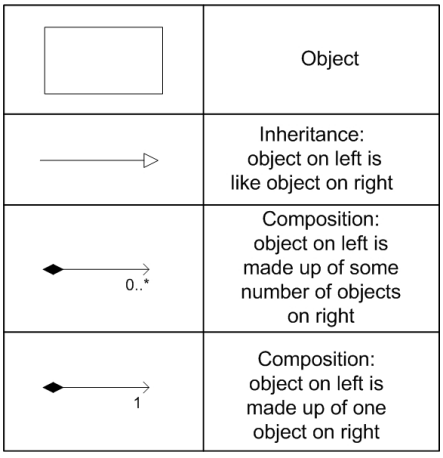


Figure 2. UML Diagram Notation

2 Product Description

2.1 What is the Transport API?

The Enterprise Transport API is a high performance, low latency, foundation of the LSEG Real-Time SDK. It consists of several layers of APIs which are built over lower layers to offer ease of use and introduce feature sets: The Transport library, Value Add Reactor library, and Watchlist features. The transport layer (also known as RSSL and interchangeably also referred to as the Enterprise Transport API), offers multiple transports, buffer management, compression, fragmentation, and packing over each transport. Additionally, the Enterprise Transport API offers encoders and decoders that implement the Open Message Model. Applications can be written to RSSL, the ValueAdd/Reactor layer, or use Watchlist features. Applications written to the Enterprise Transport API 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.

The Enterprise Transport API is currently used by products such as the LSEG Real-Time Advanced Distribution Server, Advanced Data Hub, Robust Foundation API, Data Feed Direct, and LSEG Real-Time. Due to its well-integrated and common usage across these products, the Enterprise Transport API allows clients to write applications for use with LSEG Real-Time Distribution Systems to achieve the highest performance and throughput with the lowest possible latency.

The Enterprise Transport API supports all constructs available as part of the Open Message Model. It also complements the Robust Foundation API and the Enterprise Message API by allowing users to choose the type of functionality and layer (Session or Transport) at which they want to access the LSEG Real-Time Distribution System. With the addition of the Enterprise Transport API, customers have a choice between a feature-loaded session-level API (i.e., the Robust Foundation API or Enterprise Message API) and a high-performance transport-level API (i.e., the Enterprise Transport API, Enterprise Transport API with Reactor, or Enterprise Transport API with Reactor and Watchlist).

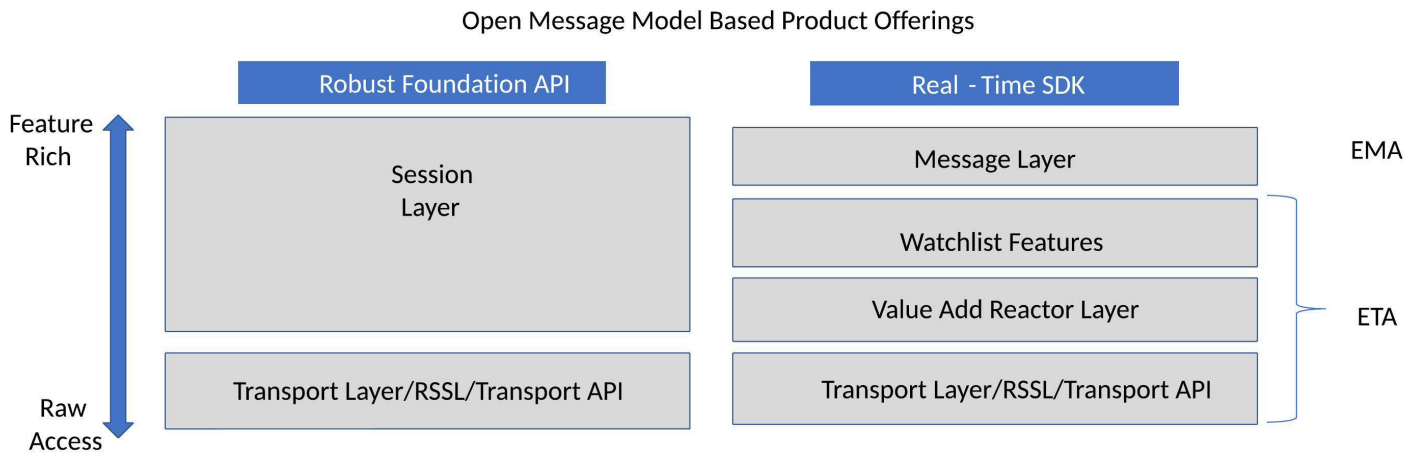


Figure 3. Open Message Model-Based Product Offerings

The Enterprise Transport API is a low-level API that provides application developers with the most flexible development environment and is the foundation on which all LSEG Open Message Model-based components are built. By utilizing an API at the transport level, a client can write to the same API as the LSEG Real-Time Advanced Distribution Server / LSEG Real-Time Advanced Distribution Hub and achieve the same levels of performance.

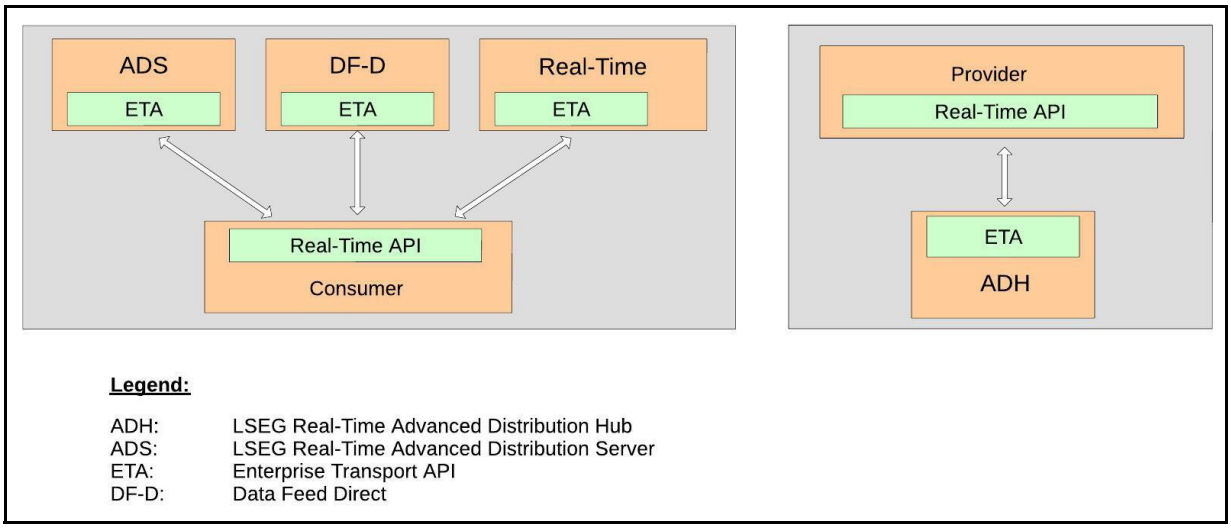


Figure 4. Transport API: Core Diagram

2.2 Transport API Features

The Enterprise Transport API is:

- Available as a C-based, C#-based, and Java-based APIs.
- 64-bit.
- Thread-safe and thread-aware.
- Capable of handling:
  - Any and all Open Message Model primitives and containers.
  - All Domain Models, including those defined by LSEG as well as other user-defined models.
- A reliable, transport-level API which includes Open Message Model encoders/decoders.

Additionally, the Enterprise Transport API provides an ANSI Page parser to encode/decode ANSI sequences and a Data Access Control System library to allow for generating DACS Locks.

2.2.1 General Capabilities

The Enterprise Transport API provides general capabilities independent of the type of application. The Enterprise Transport API:

- Supports fully connected or unified network topologies as well as segmented topologies.
- Supports multiple network session types, including TCP, HTTP, and multicast-based networks.
- Can internally fragment and reassemble large messages.
- Can pack multiple, small messages into the same network buffer.
- Can perform data compression and decompression internally.
- Can choose its locking model based on need. Locking can be enabled globally, within a connection, or disabled entirely, thus allowing clients to develop single-threaded, multi-threaded, thread-safe, or thread-aware solutions.
- Has full control over the number of message buffers and can dynamically increase or decrease this quantity during runtime.
- Does not have external configuration, log file, or message file dependencies: everything is programmatically supplied, where the user can define any external configuration or logging according to their needs.
- Allows users to write messages at different priority levels, allowing higher priority messages to be sent before lower priority messages.

2.2.2 Consumer Applications

You can use the Enterprise Transport API to create consumer-based applications that can:

- Make streaming and snapshot-based subscription requests to the LSEG Real-Time Advanced Distribution Server.
- Send batch, views, and symbol list requests to the LSEG Real-Time Advanced Distribution Server.
- Support pause and resume on active data streams with the LSEG Real-Time Advanced Distribution Server.
- Send post messages to the LSEG Real-Time Advanced Distribution Server (for consumer-based publishing and contributions).
- Send and receive generic messages with LSEG Real-Time Advanced Distribution Server.
- Establish a private stream.
- Transparently use HTTP to communicate with an LSEG Real-Time Advanced Distribution Server by tunneling through the Internet.

2.2.3 Provider Applications: Interactive

You can use the Enterprise Transport API to create interactive providers that can:

- Receive requests and respond to streaming and snapshot-based Requests from LSEG Real-Time Advanced Distribution Hub.
- Receive and respond to batch, views, and symbol list requests from LSEG Real-Time Advanced Distribution Hub.
- Receive and respond to requests for a Private Stream from the LSEG Real-Time Advanced Distribution Hub.
- Receive requests for pause and resume on active data streams.
- Receive and acknowledge post messages (used receiving consumer- based Publishing and Contributions) from LSEG Real-Time Advanced Distribution Hub.
- Send and receive Generic Messages with LSEG Real-Time Advanced Distribution Hub.

Additionally, you can use the Enterprise Transport API to create server-based applications that can accept multiple connections from LSEG Real-Time Advanced Distribution Hub, or allow multiple LSEG Real-Time Advanced Distribution Hubs to connect to a provider.

2.2.4 Provider Applications: Non-Interactive

Using the Enterprise Transport API, you can write non-interactive applications that start up and begin publishing data to LSEG Real-Time Advanced Distribution Hub or broadcast-style server applications. This includes both TCP and UDP multicast-based non-interactive provider applications.

2.3 Performance and Feature Comparison

Though LSEG Real-Time Distribution System's core infrastructure can achieve great performance numbers, such performance can suffer from bottlenecks caused by using the rich features offered in certain APIs (i.e., the Robust Foundation API) when developing high-performance applications. By writing to the Enterprise Transport API, a client can leverage the full throughput and low latency of the core infrastructure while by-passing the full set of Robust Foundation API's or Enterprise Message API's features. For a comparison of API capabilities and features, refer to Section 2.4.

As illustrated in Figure 4, core infrastructure components (as well as their performance test tools, such as **rmdstestclient** and **sink_driven_src**) are all written to the Transport API. A Enterprise Transport API-based application's maximum achievable performance (latency, throughput, etc) is determined by the infrastructure component to which is connects. Thus, to know performance metrics, you should look at the performance numbers for the associated infrastructure component. For example:

- If a Enterprise Transport API consumer application talks to the LSEG Real-Time Advanced Distribution Server and you want to know the maximum throughput and latency of the consumer, look at the performance numbers for the LSEG Real-Time Advanced Distribution Server configuration you use.
- If a Enterprise Transport API provider application talks to an LSEG Real-Time Advanced Distribution Hub and you want to know the maximum throughput and latency of the Enterprise Transport API provider, look at the performance numbers for the LSEG Real-Time Advanced Distribution Hub configuration you use.

TIP: The Enterprise Transport API now ships with API performance tools and additional documentation to which you can refer which you can use to arrive at more-specific results for your environment.

When referring to LSEG Real-Time Distribution System infrastructure documentation, look for Enterprise Transport API or RSSL performance numbers (the Enterprise Transport API is sometimes referred to in other LSEG documentation as the Source Sink Library), which will give the throughput and latency of the Enterprise Transport API and the associated core infrastructure component.

The following table compares existing API products and their performance. Key factors are latency, throughput, memory, and thread safety. Results may vary depending on whether you use of watch lists and memory queues and according to your hardware and operating system. Typically, when measuring performance on the same hardware and operating system, these comparisons remain consistent.

API	THREAD SAFETY	THROUGHPUT	LATENCY	MEMORY FOOTPRINT
Transport API	Safe and Aware	Very High	Lowest	Lowest
Message API	Safe and Aware	High	Low	Medium
Robust Foundation API	Safe and Aware	High	Low	Medium (watch list, allows optional queues)
System Foundation Classes C++	None	Medium	High	Medium – High (watch list, cache)

Table 2: API Performance Comparison

2.4 Functionality: Which API to Choose?

To make an informed decision on which API to use, you should balance consider both performance and functionality. For performance comparisons, refer to Section 2.3.

The Robust Foundation API uses information provided from the Enterprise Transport API and creates specific implementations of capabilities. Though these capabilities are not implemented in the Enterprise Transport API, Enterprise Transport API-based applications can use the information provided by the Enterprise Transport API to implement the same functionality (i.e., as provided by the Robust Foundation API). Additionally, Enterprise Transport API Value Added Components offer fully-supported reference implementations for much of this functionality.

The Enterprise Transport API Reactor is an open source component that functions within the Enterprise Transport API.

The following table lists API capabilities using the following legend:

- X: Supported in current version, natively implemented
- X*: Supported only in the C / C++ version of the software
- X**: Supported in current version, leverages lower-level capability
- Any X that is in blue: Supported only in C/C++ and Java version of the software.
- X+: Supports V2 authentication in C# and both V1 and V2 in C/C++ and Java
- Future: Planned for a future release
- Legacy: A legacy functionality

CAPABILITY TYPE	CAPABILITY	ENTERPRISE TRANSPORT API 3.X	ENTERPRISE TRANSPORT REACTOR	ENTERPRISE MESSAGE API 3.X	THE ROBUST FOUNDATION API 8.X
Transport	Compression via Open Message Model	X	X**	X**	X
	HTTP Tunneling (Rssl Wire Format)	X	X**	X**	X
	TCP/IP: Rssl Wire Format	X	X**	X**	X
	Reliable Multicast: Rssl Wire Format	X	X**	X**	X
	Sequenced Multicast	X			
	Websocket	X	X	X**	
	Unidirectional Shared Memory	X			
Application Type	Consumer	X	X	X**	X
	Provider: Interactive	X	X	X**	X
	Provider: Non-Interactive	X	X	X**	X

Table 3: Capabilities by API

CAPABILITY TYPE	CAPABILITY	ENTERPRISE TRANSPORT API 3.X	ENTERPRISE TRANSPORT REACTOR	ENTERPRISE MESSAGE API 3.X	THE ROBUST FOUNDATION API 8.X
General	Batch Request	X	X	X	X
	Batch Re-issue and Close	X	X		X
	Generic Messages	X	X	X	X
	Pause/Resume	X	X	X	X
	Posting	X	X	X	X
	Snapshot Requests	X	X	X	X
	Streaming Requests	X	X	X	X
	Private Streams	X	X	X	X
	Qualified Streams	X	X	X	
	Views	X	X	X	X
Domain Models	Custom Data Model Support	X	X	X	X
	Domain Model: Dictionary	X	X	X	X
	Domain Model: Enhanced Symbol List	X	X	X**	X
	Domain Model: Login	X	X	X	X
	Domain Model: Market Price	X	X	X	X
	Domain Model: MarketByOrder	X	X	X	X
	Domain Model: MarketByPrice	X	X	X	X
	Domain Model: Market Maker	X	X	X	X
	Domain Model: Source Directory	X	X	X	X
	Domain Model: Symbol List	X	X	X	X
	Domain Model: Yield Curve	X	X	X	X
Encoders/Decoders	AnsiPage	X	X**	X**	Legacy
	DACS Lock	X	X**	X**	X
	Open Message Model	X	X	X**	X
	RMTES	X	X	X**	X

Table 3: Capabilities by API(Continued)

CAPABILITY TYPE	CAPABILITY	ENTERPRISE TRANSPORT API 3.X	ENTERPRISE TRANSPORT REACTOR	ENTERPRISE MESSAGE API 3.X	THE ROBUST FOUNDATION API 8.X
Layer Specific	Config: file-based			X	X
	Config: programmatic	X	X	X	X
	Group fanout to items		X	X**	X
	Load balancing: API-based				X
	Logging: file-based			X	X
	Logging: programmatic	X	X		X
	Quality of Service Matching		X	X**	X
	Network Pings: automatic		X	X**	X
	Recovery: connection		X	X**	X
	Preferred Host in ConnectionList		X*	X*	
	Recovery: items		X	X**	X
	Request routing			X	X
	Round trip time	X	X	X	
	Session management		X+	X+	
	Service Groups				X
	Single Open: API-based		X	X**	X
	Warm Standby: API-based (must enable Watchlist)		X	X	X
	Warm Standby with Preferred Group		X*	X*	
	Watchlist		X	X**	X
	Controlled fragmentation and assembly of large messages	X	X**	X**	
	Controlled locking/threading model	X			

Table 3: Capabilities by API(Continued)

CAPABILITY TYPE	CAPABILITY	ENTERPRISE TRANSPORT API 3.X	ENTERPRISE TRANSPORT REACTOR	ENTERPRISE MESSAGE API 3.X	THE ROBUST FOUNDATION API 8.X
	Controlled dynamic message buffers with ability to programmatically modify during runtime	X	X**		
	Controlled message packing	X	X**	X**	
	Messages can be written at different priority levels	X	X**	X**	

Table 3: Capabilities by API(Continued)

3 Consumers and Providers

3.1 Overview

For those familiar with previous API products or concepts from LSEG Real-Time Distribution System, we map how the Enterprise Transport API implements the same functionality.

At a very high level, the LSEG Real-Time Distribution System system facilitates controlled and managed interactions between many different service ***providers*** and ***consumers***. Thus, LSEG Real-Time Distribution System is a real-time, streaming Service Oriented Architecture (SOA) used extensively as middleware integrating financial-service applications. While providers implement services and expose a certain set of capabilities (e.g. content, workflow, etc.), consumers use the capabilities offered by providers for a specific purpose (e.g., trading screen applications, black-box algorithmic trading applications, etc.). In some cases, a single application can function as both a consumer and a provider (e.g., a computation engine, value-add server, etc.).

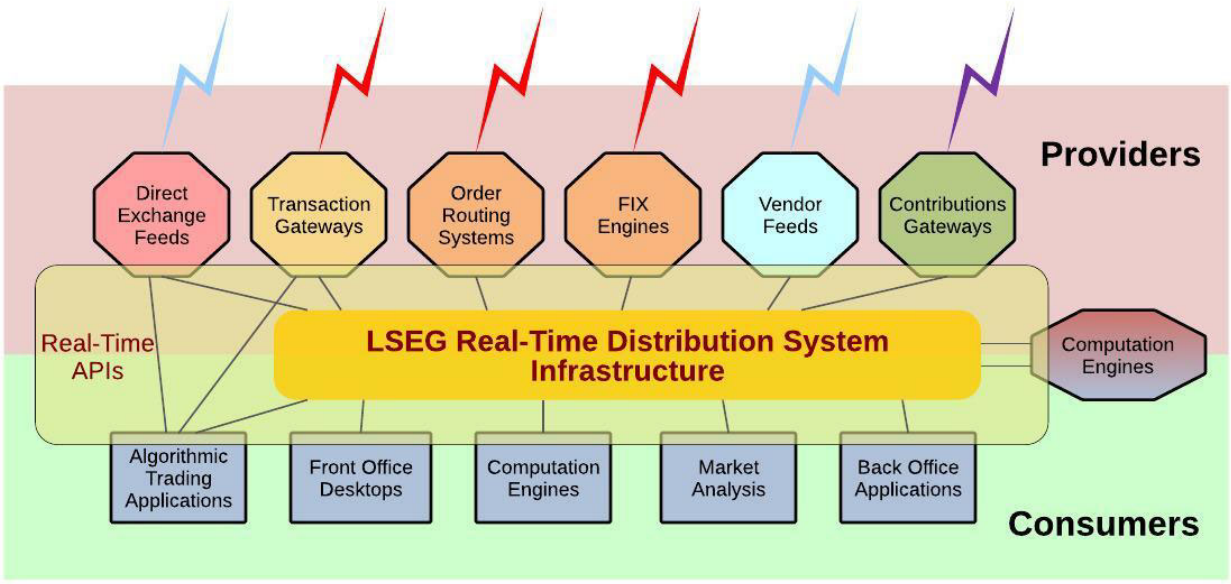


Figure 5. LSEG Real-Time Distribution System Infrastructure

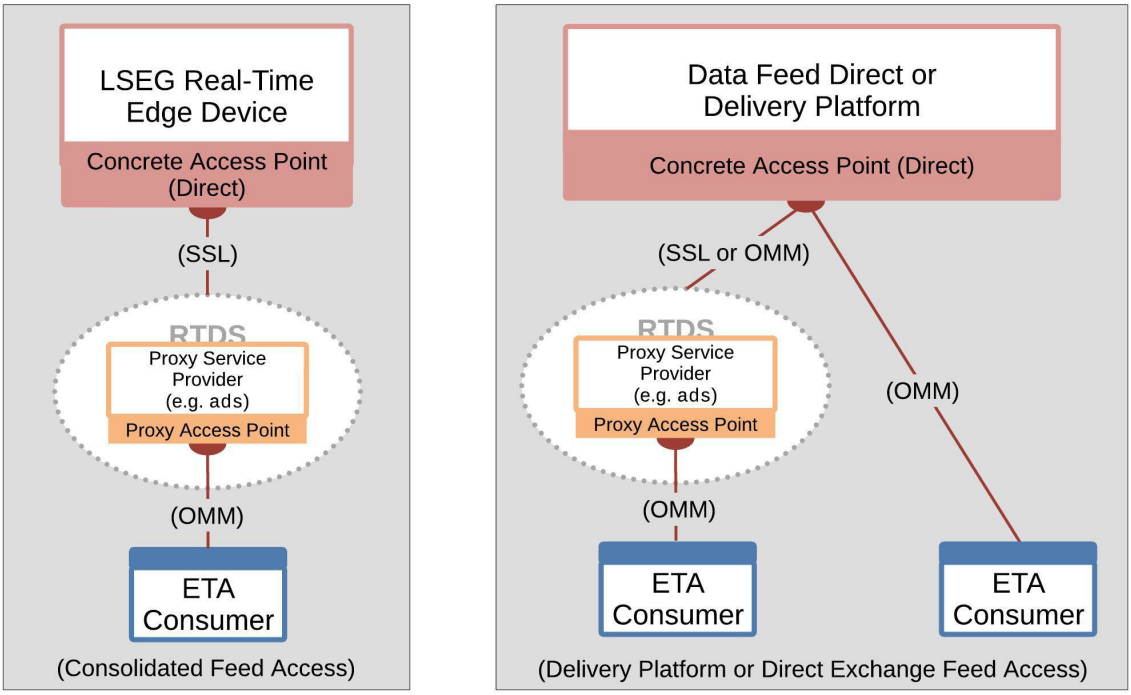
To access needed capabilities, consumers always interact with a provider, either directly and/or via LSEG Real-Time Distribution System. Consumer applications that want the lowest possible latency can communicate directly via LSEG Real-Time (or LSEG Real-Time Distribution System) APIs with the appropriate service providers. However, you can implement more complex deployments (i.e., integrating multiple providers, managing local content, automated resiliency, scalability, control, and protection) by placing the LSEG Real-Time Distribution System infrastructure between provider and consumer applications.

NOTE: Enterprise Message API C# Edition supports only consumers in current release.

3.2 Consumers

Consumers make use of capabilities offered by providers through access points. To interact with a provider, the consumer must attach to a consumer access point. Access points manifest themselves in two different forms:

- A ***concrete access point***. A concrete access point is implemented by the service-provider application if it supports direct connections from consumers. The right-side diagram in the following figure illustrates a Enterprise Transport API consumer connecting to LSEG Real-Time via a direct access point.
- A ***proxy access point***. A proxy access point is point-to-point based and implemented by an LSEG Real-Time Distribution System Infrastructure component (i.e., an LSEG Real-Time Advanced Distribution Server). The following figure also illustrates a Enterprise Transport API consumer connecting to the provider by first passing through a proxy access point.



Legend:
ETA: Enterprise Transport API
OMM: Open Message Model
RTDS: LSEG Real-Time Distribution System
SSL: Sink Source Library

Figure 6. Enterprise Transport API as a Consumer

Examples of consumers include:

- An application that subscribes to data via LSEG Real-Time Distribution System or LSEG Real-Time.
- An application that posts data to LSEG Real-Time Distribution System or LSEG Real-Time (e.g., contributions/inserts or local publication into a cache).
- An application that communicates via generic messages with LSEG Real-Time Distribution System or LSEG Real-Time.
- An application that does any of the above via a private stream.

3.2.1 Subscriptions: Request/Response

After a consumer successfully logs into a provider (i.e., LSEG Real-Time Advanced Distribution Server or LSEG Real-Time) and obtains a list of available sources, the consumer can then subscribe and receive data for various services. A consumer subscribes to a service or service ID that in turn maps to a service name in the Source Directory. Any service or service ID provides a set of items to its clients.

- If a consumer's request does not specify interest in future changes (i.e., after receiving a full response), the request is a classic ***snapshot request***. The data stream is considered closed after a full response of data (possibly delivered in multiple parts) is sent to the consumer. This is typical behavior when a user sends a non-streaming request. Because the response contains all current information, the stream is considered complete as soon as the data is sent.
- If a consumer's request specifies interest in receiving future changes (i.e., after receiving a full response), the request is considered to be a ***streaming request***. After such a request, the provider sends the consumer an initial set of data and then sends additional changes or "updates" to the data as they occur. The data stream is considered open until either the consumer or provider closes it. A consumer typically sends a streaming request when a user subscribes for an item and wants to receive every change to that item for the life of the stream.

Specialized cases of request / response include:

- Batches
- Views
- Symbol Lists
- Server Symbol Lists

3.2.2 Batches

A consumer can request multiple items using a single, client-based, request called a ***batch*** request. After the Transport API consumer sends an optimized batch request to the LSEG Real-Time Advanced Distribution Server, the LSEG Real-Time Advanced Distribution Server responds by sending the items as if they were opened individually so the items can be managed individually.

Figure 7 illustrates a Transport API consumer issuing a batch request for "TRI, "GE", and "INTC.O" and the resulting LSEG Real-Time Advanced Distribution Server responses.

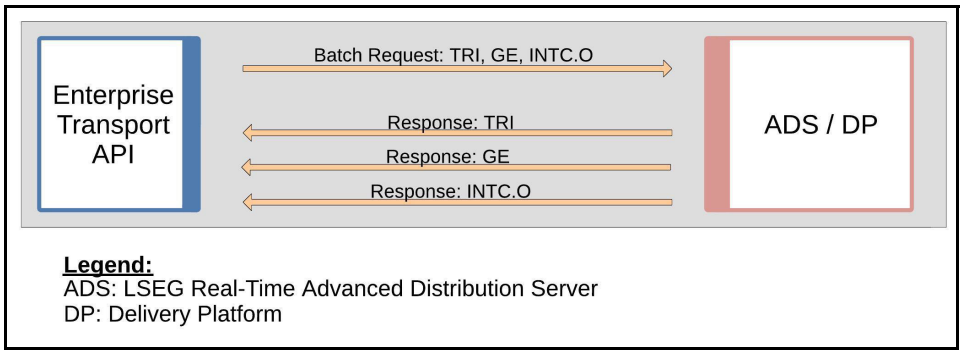


Figure 7. Batch Request

3.2.3 Views

The system reduces the amount of data that flows across the network by filtering out content in which the user is not interested. To improve performance and maximize bandwidth, you can configure the LSEG Real-Time Distribution System to filter out certain fields to downstream users. When filtering, all consumer applications see the same subset of fields for a given item.

Another way of controlling filtering is to configure the consumer application to use ***Views***. Using a view, a consumer requests a subset of fields with a single, client-based request (refer to Figure 8). The API then requests (from the LSEG Real-Time Advanced Distribution Server / LSEG Real-Time) only the fields of interest. When the API receives the requested fields, it sends the subset back to the consumer. This is also called consumer-side (or request-side) filtering.

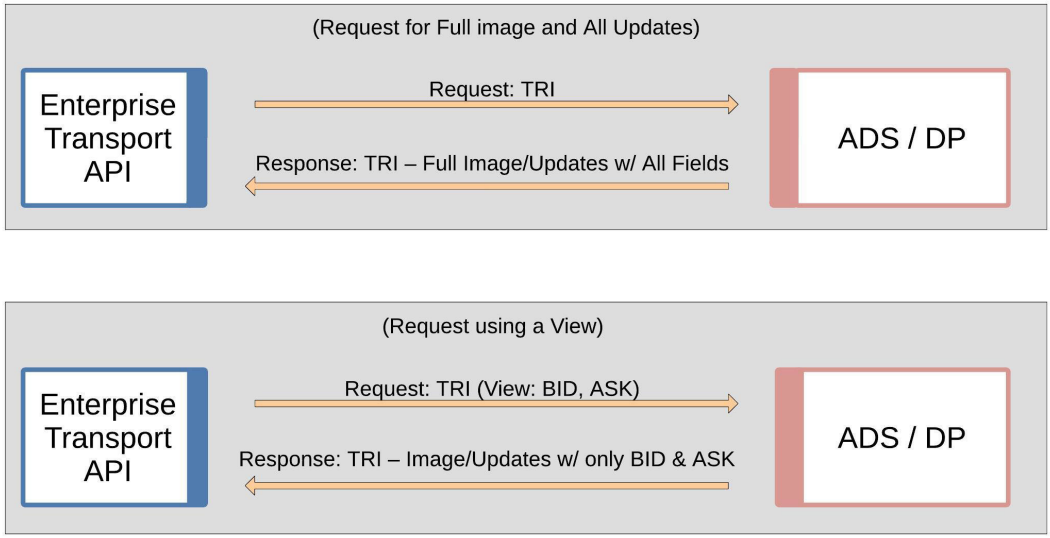


Figure 8. View Request Diagram

Views were designed to provide the same filtering functionality as the System Foundation Classes (based on its own internal cache) while optimizing network traffic.

Views, in conjunction with server-side filtering, can be a powerful tool for bandwidth optimization on a network. Users can combine a view with a batch request to send a single request to open multiple items using the same view.

3.2.4 Pause and Resume

The ***Pause/Resume*** feature optimizes network bandwidth. You can use Pause/Resume to reduce the amount of data flowing across the network for a single item or for many items that might already be openly streaming data to a client.

To pause/resume data, the client first sends a request to pause an item to the LSEG Real-Time Advanced Distribution Server. The LSEG Real-Time Advanced Distribution Server receives the pause request and stops sending new data to the client for that item, though the item remains open and in the LSEG Real-Time Advanced Distribution Server cache. The LSEG Real-Time Advanced Distribution Server continues to receive messages from the upstream device (or feed) and continues to update the item in its cache (but because of the client's pause request, does not send the new data to the client). When the client wants to start receiving messages for the item again, the client sends a resume to the LSEG Real-Time Advanced Distribution Server, which then responds by sending an aggregated update or a refresh (a current image) to the client. After the LSEG Real-Time Advanced Distribution Server resumes sending data, the LSEG Real-Time Advanced Distribution Server sends all subsequent messages.

By using the Pause/Resume feature a client can avoid issuing multiple open/close requests which can disrupt the LSEG Real-Time Advanced Distribution Server and prolong recovery times. There are two main use-case scenarios for this feature:

- Clients with intensive back-end processing
- Clients that display a lot of data

3.2.4.1 Pause / Resume Use Case 1: Back-end Processing

In this use-case, a client application performs heavy back-end processing and has too many items open, such that the client is at the threshold for lowering the downstream update rate. The client now needs to run a specialized report, or do some other back-end processing. Such an increase in workload on the client application will negatively impact its downstream message traffic. The client does not want to back up its messages from the LSEG Real-Time Advanced Distribution Server and risk having LSEG Real-Time Advanced Distribution Server abruptly cut its connection, nor does the client want to close its own connection (or close all the items on the LSEG Real-Time Advanced Distribution Server) which would require the client to re-open all items after finishing its back-end processing.

In this case, the client application:

- Sends a single PAUSE message to the LSEG Real-Time Advanced Distribution Server to pause all the items it has open.
- Performs all needed back-end processing.
- Sends a Resume request to resume all the items it had paused.

After receiving the Resume request, the LSEG Real-Time Advanced Distribution Server sends a refresh (i.e., current image), to the client for all paused items and then continues to send any subsequent messages.

3.2.4.2 Pause / Resume Use Case 2: Display Applications

The second use case assumes the application displays a lot of data. In this scenario, the user has two windows open. One window has item "TRI" open and is updating (Window 1). The other has "INTC.O" open and is updating (Window 2). On his screen, the user moves Window 1 to cover Window 2 and the user can no longer see the contents of Window 2. In this case, the user might not need updates for "INTC.O" because the contents are obstructed from view. In this case, the client application can:

- Pause "INTC.O" as long as Window 2 is covered and out of view.
- Resume the stream for "INTC.O" when Window 2 moves back into view.

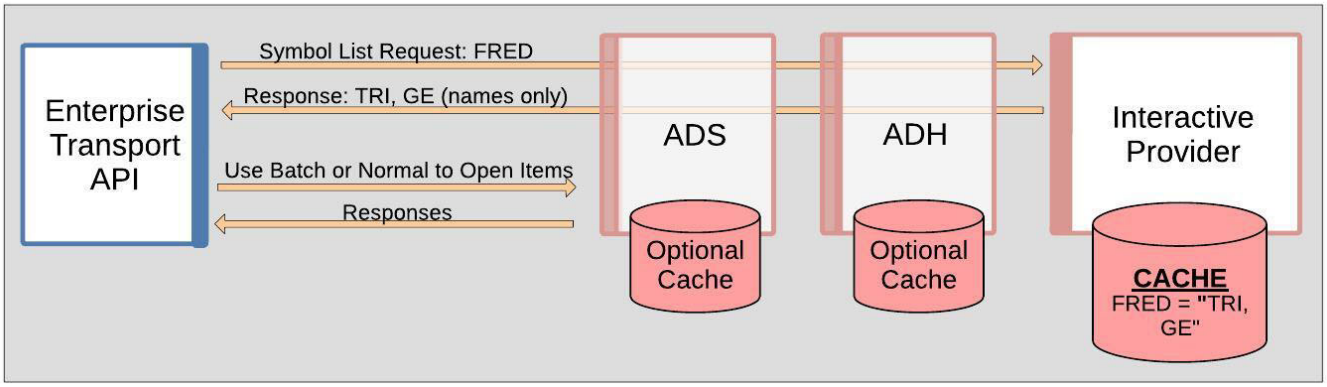
When Window 2 is again visible, the LSEG Real-Time Advanced Distribution Server sends a refresh, or current image, to the client for the item "INTC.O" and then continues to send any subsequent messages.

3.2.5 Symbol Lists

If a consumer wants to open multiple items but doesn't know their names, the consumer can first issue a request using a ***Symbol List***. However, the consumer can issue such a request only if a provider exists that can resolve the symbol list name into a set of item names.

This replaces the functionality for clients that previously used Criteria-Based Requests (CBR) with the Source Sink Library 4.5 API.

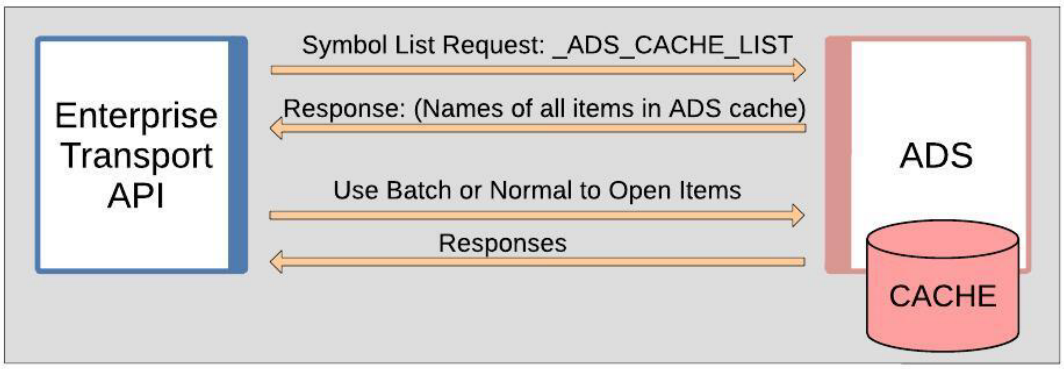
The following diagram illustrates issuing a basic symbol list request. In this diagram, the consumer issues the request using a particular key name (**`FRED`**). The request flows through the platform to a provider capable of resolving the symbol list name (the interactive provider with **`FRED`** in its cache). The provider sends back all names that map to **`FRED`** (**`TRI`** and **`GE`**). After receiving the response, the client can then choose whether to open items; individually or by making a batch request for multiple items. A subsequent request is resolved by the first cache that contains the data (listed in the diagram as optional caches).



Legend:
ADH: LSEG Real-Time Advanced Distribution Hub
ADS: LSEG Real-Time Advanced Distribution Server

Figure 9. Symbol List: Basic Scenario

The following diagram illustrates how a consumer can access all items in the LSEG Real-Time Advanced Distribution Server cache, effectively dumping the cache to the Open Message Model client. In this scenario, the client requests the symbol list **`_ADS_CACHE_LIST`**. The LSEG Real-Time Advanced Distribution Server receives the request and responds with the names of all items in its cache. The client can then choose to open items individually, or make a batch request to open multiple items. The LSEG Real-Time Advanced Distribution Server provides an additional symbol list (**`_SERVER_LIST`**) for obtaining lists of items stored in specific LSEG Real-Time Advanced Distribution Hub instances. For details on this symbol list, refer to the *LSEG Real-Time Advanced Distribution Server* and *LSEG Real-Time Advanced Distribution Hub System Administration Manuals*.



Legend:
ADS: LSEG Real-Time Advanced Distribution Server

Figure 10. Symbol List: Accessing the Entire LSEG Real-Time Advanced Distribution Server Cache

3.2.5.1 Requesting Symbol List Data Streams

For consumer applications using the Transport API reactor value-add component on certain APIs: if the consumer watchlist is enabled, an application can indicate in its request that it wants streams for the items in the symbol list to be opened on its behalf. The reactor will internally process responses on the symbol list stream and open requests as new items appear in the list. The responses to these item requests will be provided to the application using negative **`streamId`** values.

The reactor supports this method with the LSEG Real-Time Advanced Distribution Server or in direct connections with interactive providers. For details on the model for requesting symbol list data streams, see the *Enterprise Transport API LSEG Domain Model Usage Guide* specific to the API that you use.

NOTE: The reactor opens items from the symbol list as market price items, and uses the best available quality of service advertised by the service in the provider's source directory response.

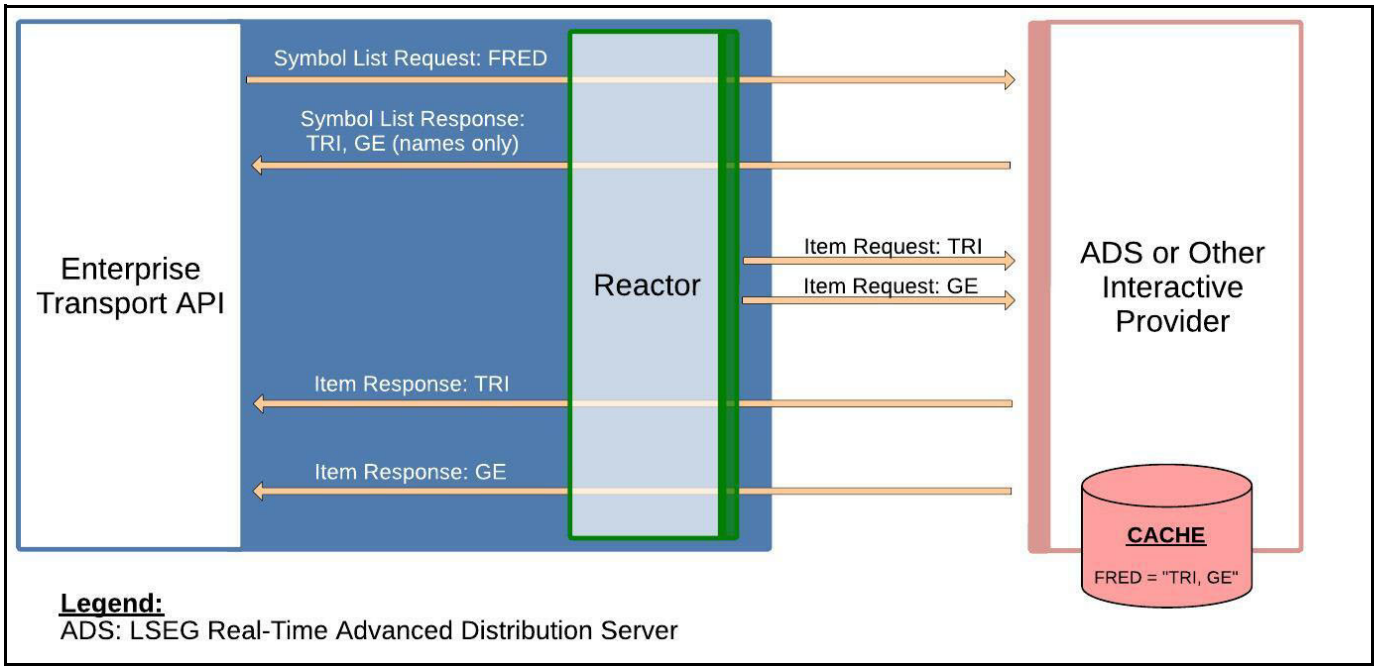


Figure 11. Symbol List: Requesting Symbol List Streams via the Transport API Reactor

3.2.5.2 Server Symbol Lists

Using certain LSEG Real-Time APIs, client Client applications can request a list of all symbols maintained in the cache of all LSEG Real-Time Advanced Distribution Hub servers across the network. Client applications start by first requesting a symbol list item **_SERVER_LIST** which will return a list of all servers and their supported domains. Each entry on that list is a symbol list item name formatted as follows **_CACHE_LIST.*serverId.domain***. Client applications can then spawn individual symbol list requests for servers and domains of interest using the symbol name **_CACHE_LIST.*serverId.domain***. If ***domain*** is not provided, it defaults to **6**.

The symbol list response for **_CACHE_LIST.*serverId.domain*** will include a list of all Level 1 or Level 2 items in the server cache. It will also include opened non-cached items but not items opened on private streams. The symbol list response will provide only item names, not item data.

The streams for **_SERVER_LIST** and **_CACHE_LIST.*serverId.domain*** requests will be kept open and updates will be sent to modify list of servers or list of items in server cache. These streams will be closed if a server is no longer available or it no longer supports a particular domain.

If the LSEG Real-Time Advanced Distribution Hub is configured for source mirroring, a failover will trigger a server id change and will lead to closing of the relevant **_CACHE_LIST.*serverId.domain*** request and updating of the **_SERVER_LIST** to show the new server id after the failover. Clients will need to make a new symbol list request to the new server.

This feature provides the symbol list of all items in the LSEG Real-Time Advanced Distribution Hub cache for both interactive and non-interactive services and is supported for both RSSL (symbol list) and SSL 4.5 (criteria) clients.

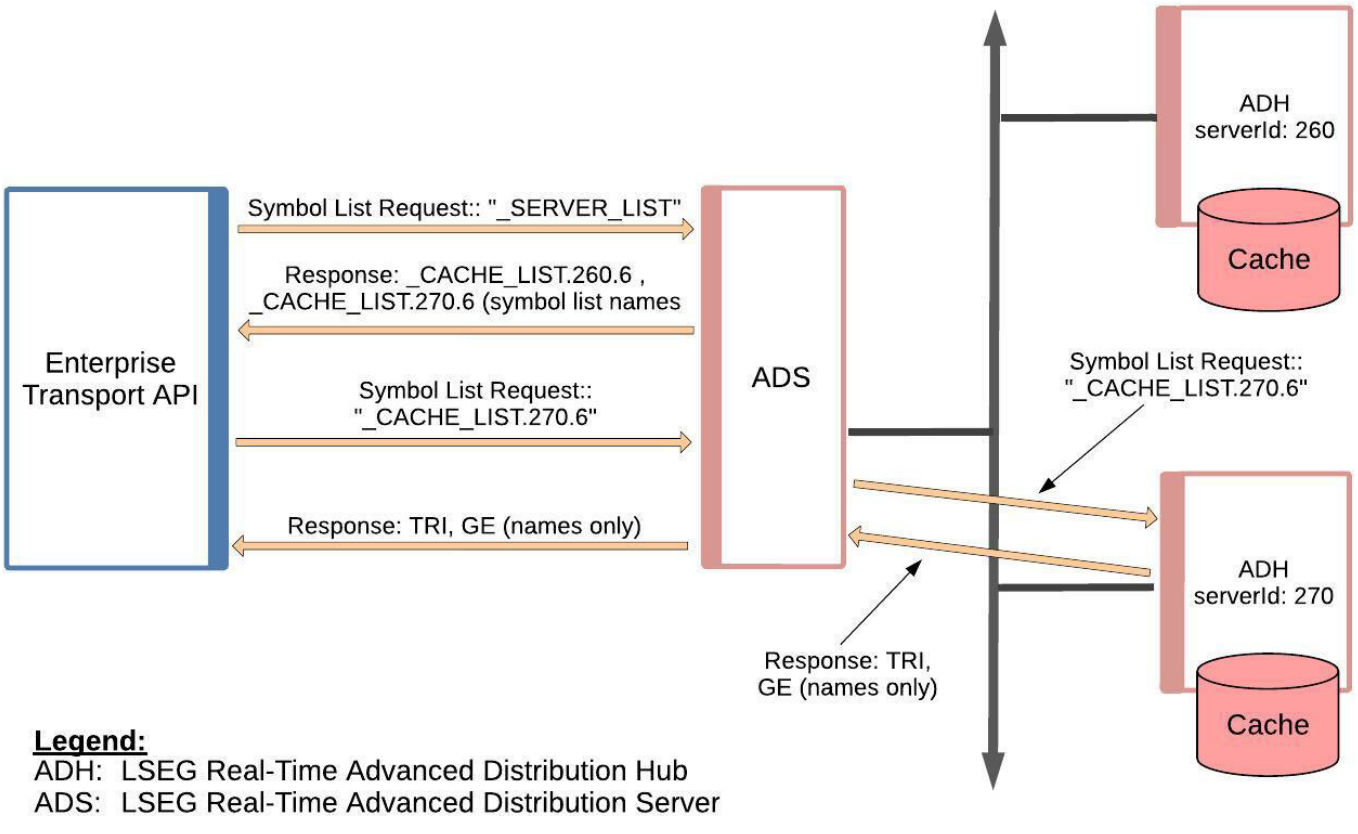


Figure 12. Server Symbol List

3.2.6 Posting

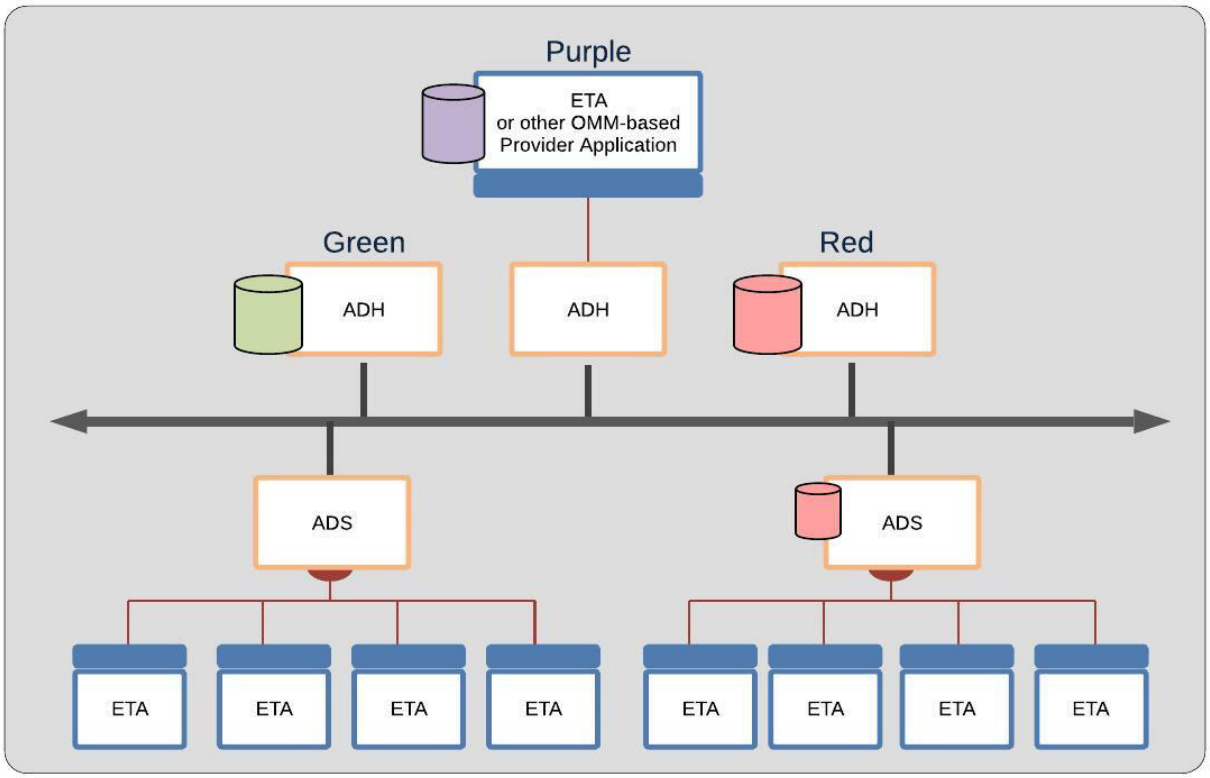
Through posting, API consumers can easily push content into any cache within the LSEG Real-Time Distribution System (i.e., an HTTP POST request). Data contributions/inserts into the ATS or publishing into a cache offer similar capabilities today. When posting, API consumer applications reuse their existing sessions to publish content to any cache(s) residing within the LSEG Real-Time Distribution System (i.e., service provider(s) and/or infrastructure components). When compared to spreadsheets or other applications, posting offers a more efficient form of publishing, because the application does not need to create a separate provider session or manage event streams. The posting capability, unlike unmanaged publishing or inserts, offers optional acknowledgments per posted message. The two types of posting are on-stream and off-stream:

- ***On-Stream Post***: Before sending an on-stream post, the client must first open (request) a data stream for an item. After opening the data stream, the client application can then send a post. The route of the post is determined by the route of the data stream.
- ***Off-Stream Post***: In an off-stream post, the client application can send a post for an item via a Login stream, regardless of whether a data stream first exists. The route of the post is determined by the Core Infrastructure (i.e., LSEG Real-Time Advanced Distribution Server, LSEG Real-Time Advanced Distribution Hub, etc.) configuration.

3.2.6.1 Local Publication

The following diagram illustrates the benefits of posting.

Green and Red services support internal posting and are fully implemented within the LSEG Real-Time Advanced Distribution Hub. In both cases the LSEG Real-Time Advanced Distribution Hub receives posted messages and then distributes these messages to interested consumers. In the right-side segment, the LSEG Real-Time Advanced Distribution Server component has enabled caching (for the Red service). In this case posted messages received from connected applications are cached and distributed to these local applications before being forwarded (re-posted) up into the LSEG Real-Time Advanced Distribution Hub cache. The Enterprise Transport API can even post to provider applications (i.e., the Purple service in this diagram) that support posting.



Legend:
ADH: LSEG Real-Time Advanced Distribution Hub
ADS: LSEG Real-Time Advanced Distribution Server
ETA: Enterprise Transport API

Figure 13. Posting into a Cache

You can use the Enterprise Transport API to post into an LSEG Real-Time Advanced Distribution Hub cache. If a cache exists in the LSEG Real-Time Advanced Distribution Server (the Red service), the LSEG Real-Time Advanced Distribution Server cache is also populated by responses from the LSEG Real-Time Advanced Distribution Hub cache. If you configure LSEG Real-Time Distribution System to allow such behavior, posts can be sent beyond the LSEG Real-Time Advanced Distribution Hub (to the Provider Application in the Purple service). Such posting flexibility is a good solution if one's applications are restricted to a LAN which hosts an LSEG Real-Time Advanced Distribution Server but allows publishing up the network to a cache with items to which other clients subscribe.

3.2.6.2 Contribution/Inserts

Posting also allows Open Message Model-based contributions. Through such posting, clients can contribute data to a device on the head end or to a custom-provider. In the following example, the Enterprise Transport API sends an Open Message Model post to a provider application that supports such functionality.

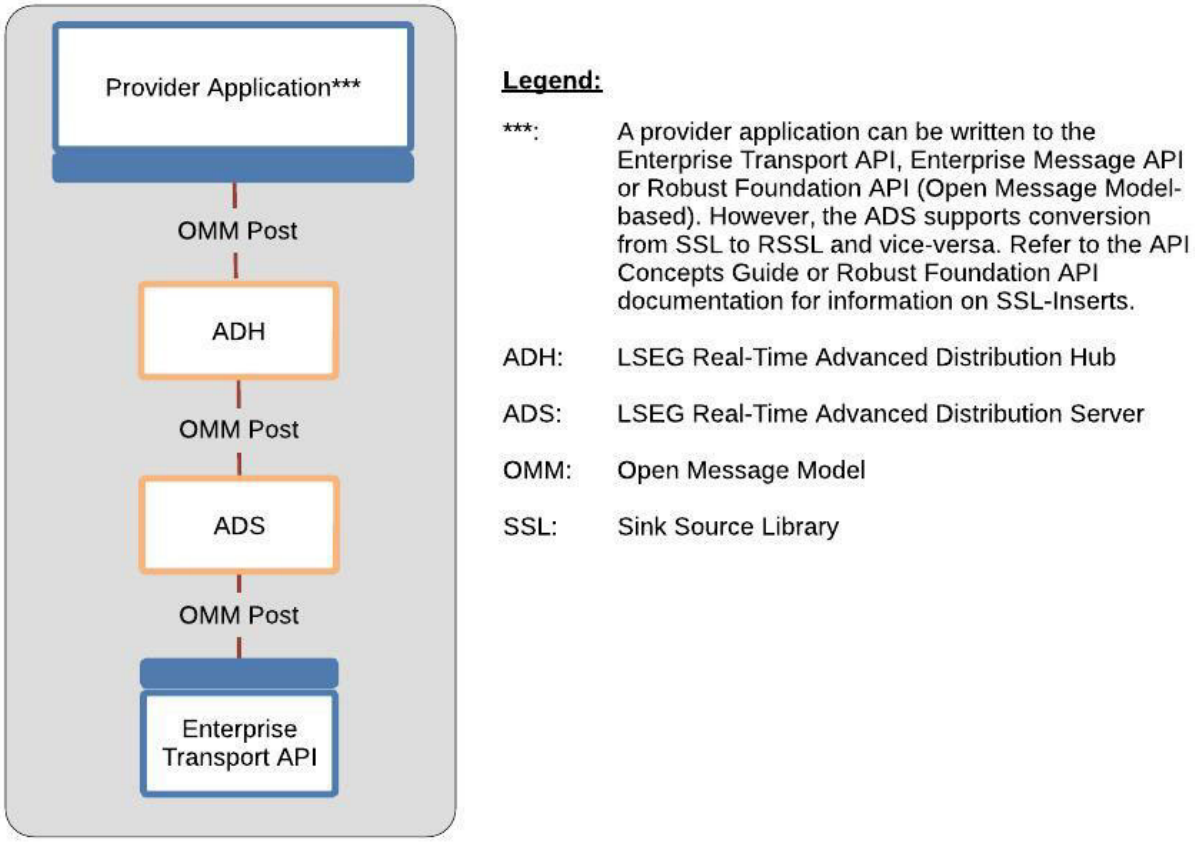


Figure 14. Open Message Model Post with Legacy Inserts

3.2.7 Generic Message

Using a ***Generic Message***, an application can send or receive a bi-directional message. A generic message can contain any Open Message Model primitive type. Whereas the request/response type message flows from LSEG Real-Time Distribution System to a consumer application, a generic message can flow in any direction, and a response is not required or expected. One advantage to using generic messages is its freedom from the traditional request/response data flow.

In a generic message scenario, the consumer sends a generic message to an LSEG Real-Time Advanced Distribution Server, while the LSEG Real-Time Advanced Distribution Server also publishes a generic message to the consumer application. All domains support this type of generic message behavior, not just market data-based domains (such as Market Price, etc). If a generic message is sent to a component that does not understand generic messages, the component ignores the message.

3.2.8 Private Streams

Using a ***Private Stream***, a consumer application can create a virtual private connection with an interactive provider. This virtual private connection can be either a direct connection, through the LSEG Real-Time Distribution System, or via a cascaded set of platforms. The following diagram illustrates these different configurations.

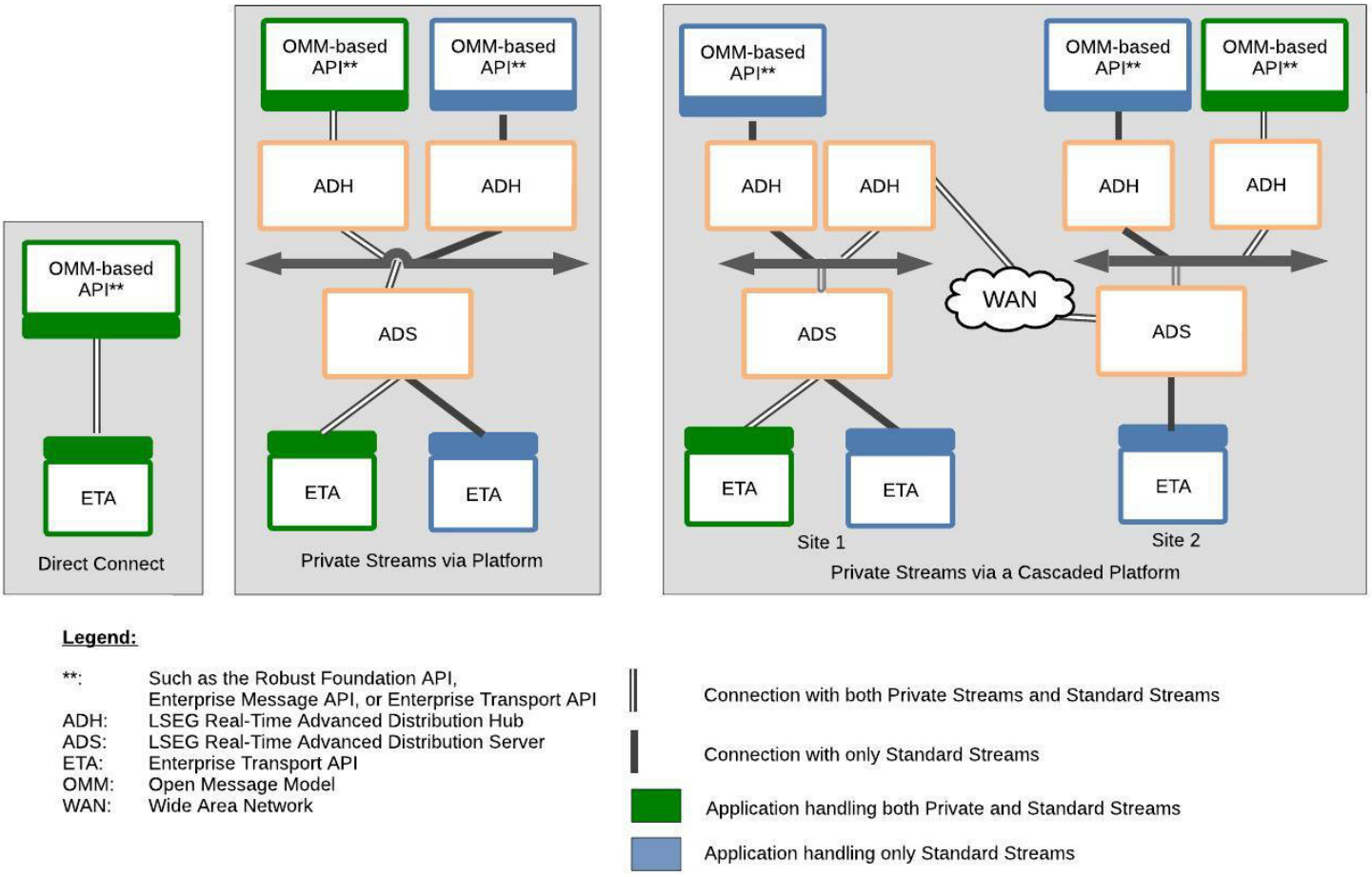


Figure 15. Private Stream Scenarios

A virtual private connection piggy backs on existing, individual point-to-point and multicast connections in the system (Figure 15 illustrates this behavior using a white connector). Messages exchanged via a Private Stream flow between a Consumer and an Interactive Provider using these existing underlying connections. However, unlike a regular stream, the Enterprise Transport API or LSEG Real-Time Distribution System components do not fan out these messages to other consumers or providers.

In Figure 15, each diagram shows a green consumer creating a private stream with a green provider. The private stream, using existing infrastructure and network connections, is illustrated as a white path in each of the diagrams. When established, communications sent on a private stream flow only between the green consumer and the green provider to which it connects. Blue providers and consumers do not see messages sent via the private stream.

Any break in a “virtual connection” causes the provider and consumer to be notified of the loss of connection. In such a scenario, the consumer is responsible for re-establishing the connection and re-requesting any data it might have missed from the provider. All types of requests, functionality, and Domain Models can flow across a private stream, including (but not limited to):

- Streaming Requests
- Snapshot Requests
- Posting
- Generic Messages

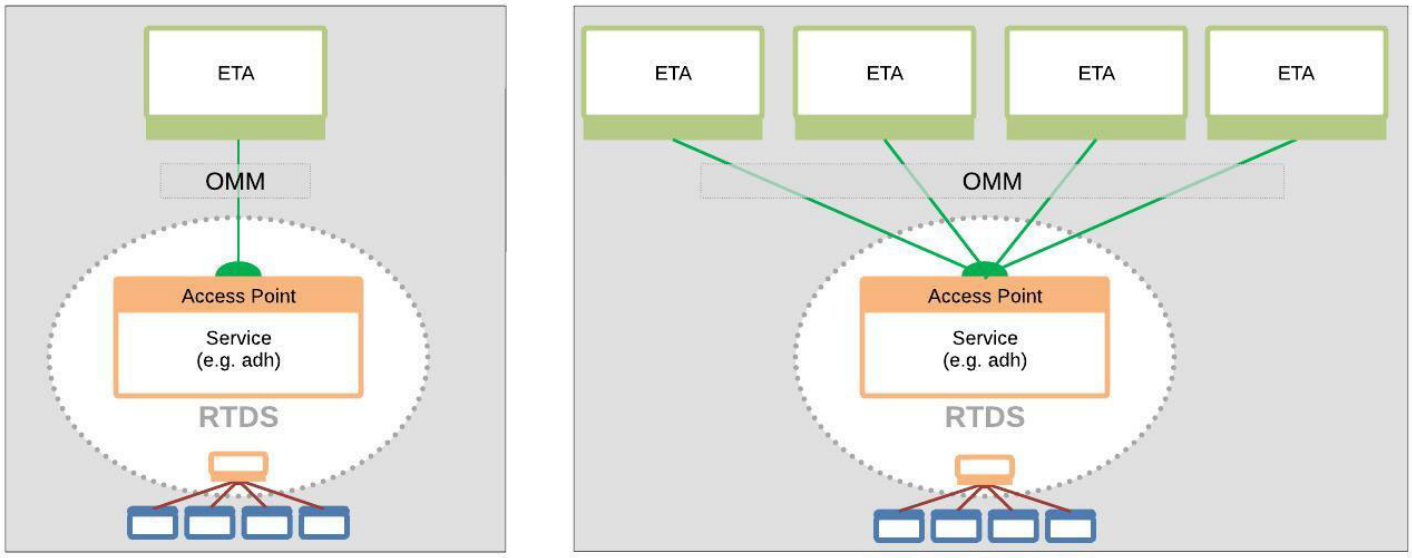
- Batch Requests
- Views
- All LSEG Domain Models & Custom Domain Models

3.3 Providers

Providers make their services available to consumers through LSEG Real-Time Distribution System infrastructure components. Every provider-based application must attach to a provider access point to inter-operate with consumers. All provider access points are considered concrete and are implemented by an LSEG Real-Time Distribution System infrastructure component (like the LSEG Real-Time Advanced Distribution Hub).

Examples of providers include:

- A user who receives a subscription request from LSEG Real-Time Distribution System.
- A user who publishes data into LSEG Real-Time Distribution System, whether in response to a request or using a broadcast-publishing style.
- A user who receives post data from LSEG Real-Time Distribution System. Providers can handle such concepts as receiving requests for contributions/inserts, or receiving publication requests.
- A user who sends and/or receives generic messages with LSEG Real-Time Distribution System.



Legend:
ETA: Enterprise Transport API
OMM: Open Message Model
RTDS: LSEG Real-Time Distribution System

Figure 16. Provider Access Point

3.3.1 Interactive Providers

An ***interactive provider*** is one that communicates with the LSEG Real-Time Distribution System, accepting and managing multiple connections with LSEG Real-Time Distribution System components. The following diagram illustrates this concept.

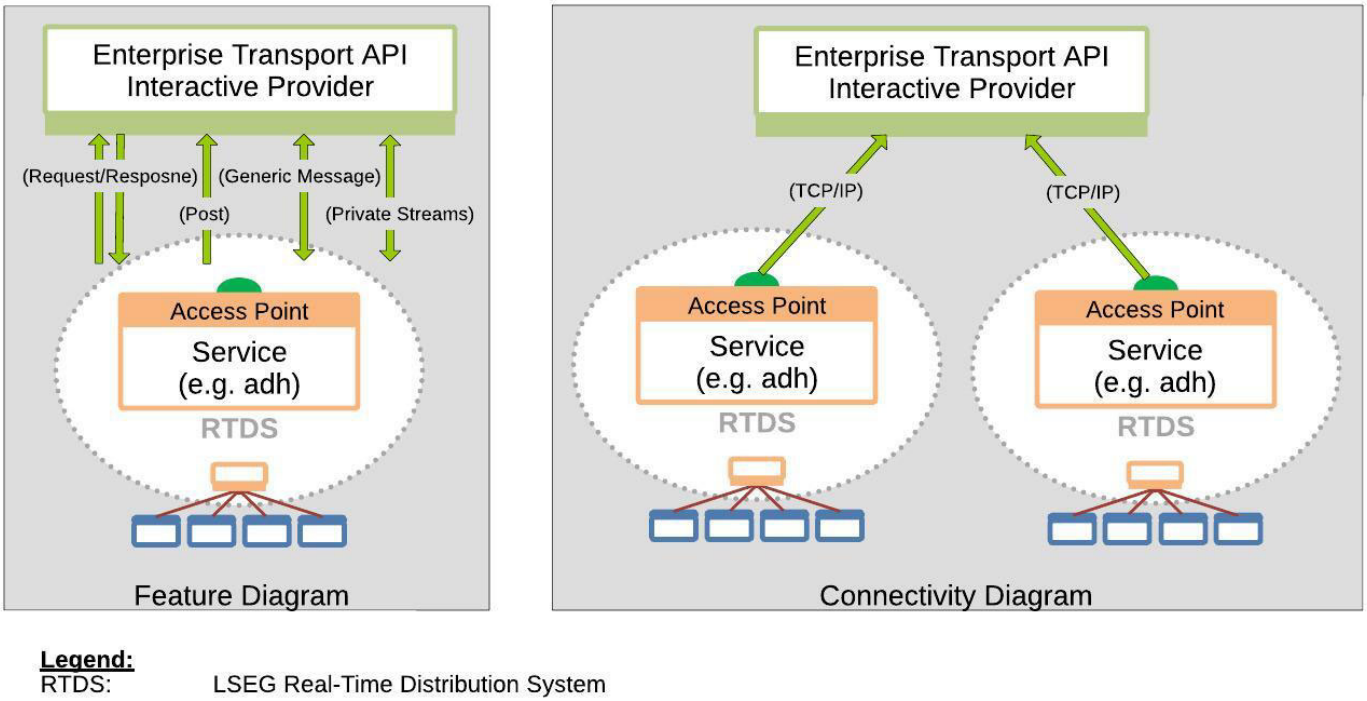


Figure 17. Interactive Providers

An interactive provider receives connection requests from the LSEG Real-Time Distribution System. The Interactive Provider responds to requests for information as to what services, domains, and capabilities it can provide or for which it can receive requests. It may also receive and respond to requests for information about its data dictionary, describing the format of expected data types. After this is completed, its behavior is interactive.

For the LSEG Real-Time Distribution System adopters, the Interactive Provider is similar in concept to the legacy Sink-Driven Server or Managed Server Application. Interactive Providers act like servers in a client-server relationship. A Enterprise Transport API interactive provider can accept and manage connections from multiple LSEG Real-Time Distribution System components.

3.3.1.1 Request /Response

In a standard request/response scenario, the interactive provider receives requests from consumers on LSEG Real-Time Distribution System (e.g., “Provide data for item AAPL”). The consumer then expects the interactive provider to provide a response, status, and possible updates whenever the information changes. If the item cannot be provided by the interactive provider, the consumer expects the provider to reject the request by providing an appropriate response - commonly a status message with state and text information describing the reason. Request and response behavior is supported in all domains, not simply Market-Data-based domains.

Interactive providers can receive any consumer-style request described in the consumer section of this document, including batch requests, views, symbol lists, pause/resume, etc. Provider applications should respond with a negative acknowledgment or response if the interactive application cannot provide the expected response to a request.

3.3.1.2 Posts

The interactive provider can receive post messages via LSEG Real-Time Distribution System. Post messages will state whether an acknowledgment is required. If required, LSEG Real-Time Distribution System will expect the interactive provider to provide a response, in the form of a positive or negative acknowledgment. Post behavior is supported in all domains, not simply Market-Data-based domains. Whenever an interactive provider connects to LSEG Real-Time Distribution System and publishes the supported domains, the provider states whether it supports post messages.

Further discussion on posting can be found in Section 13.9.

3.3.1.3 Generic Messages

Using generic messages, an application can send or receive bi-directional messages. Whereas a request/response type message flows from LSEG Real-Time Distribution System to an interactive provider, generic messages can flow in any direction and do not expect a response. When using generic messages, the application need not conform to the request/response flow. A generic message can contain any Open Message Model data type.

Interactive providers can receive a generic message from and publish a generic message to LSEG Real-Time Distribution System.

Generic message behavior is supported in all domains, not simply Market-Data-based domains. If a generic message is sent to a component (e.g., a legacy application) which does not understand generic messages, the component ignores it.

Additional details on generic messages can be found in Section 12.2.6.

3.3.1.4 Private Streams

In a typical private stream scenario, the interactive provider can receive requests for a private stream. Once established, interactive providers can receive any consumer-style request via a private stream, described in the consumer section of this document, including Batch requests, Views, Symbol Lists, Pause/Resume, Posting, etc. Provider applications should respond with a negative acknowledgment or response if the interactive application cannot provide the expected response to a request.

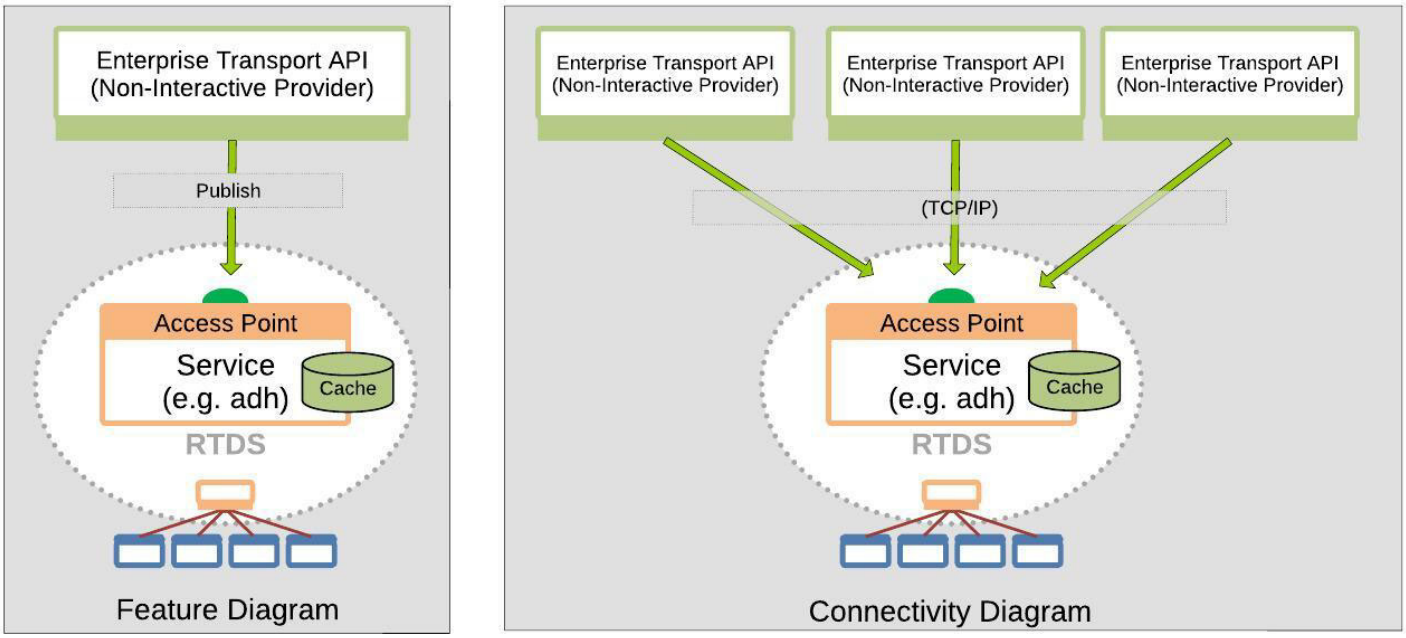
3.3.1.5 Tunnel Streams (Available Only in ETA Reactor and EMA)

An interactive provider can receive requests for a tunnel stream when using the ETA Reactor or EMA. When creating a tunnel stream, the consumer indicates any additional behaviors to enforce, which is exchanged with the provider application end point. The provider end-point acknowledges creation of the stream as well as the behaviors that it will enforce on the stream. After the stream is established, the consumer can exchange any content it wants, though the tunnel stream will enforce behaviors on the transmitted content as negotiated with the provider.

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

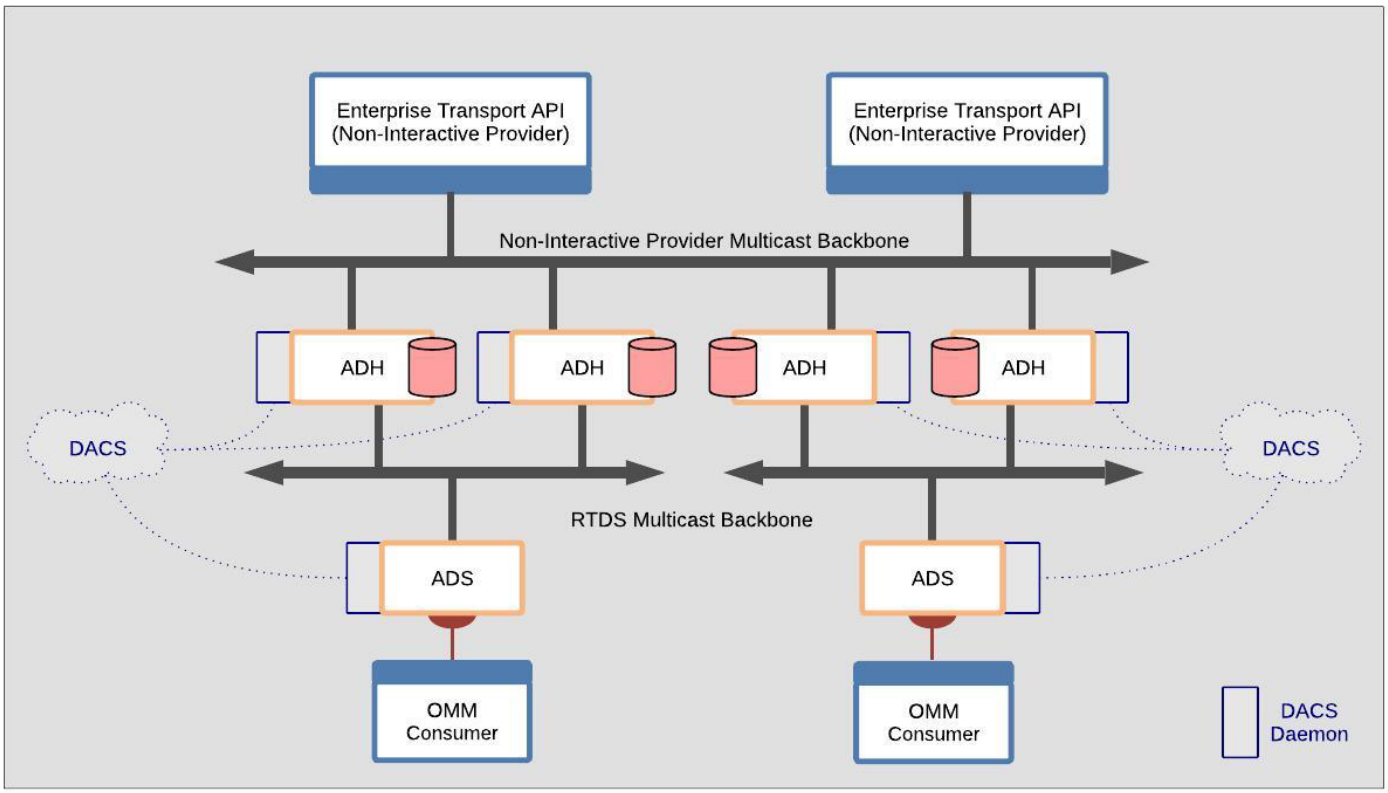
3.3.2 Non-Interactive Providers

A ***non-interactive provider*** writes a provider application that connects to LSEG Real-Time Distribution System and sends a specific set of non-interactive data (services, domains, and capabilities).



Legend:
RTDS: LSEG Real-Time Distribution System

Figure 18. Non-Interactive Provider: Point-To-Point



Legend:

ADH:	LSEG Real-Time Advanced Distribution Hub
ADS:	LSEG Real-Time Advanced Distribution Server
DACS:	Data Access Control System
OMM:	Open Message Model
RTDS:	LSEG Real-Time Distribution System

Figure 19. Non-Interactive Provider: Multicast

After a non-interactive provider connects to LSEG Real-Time Distribution System, the non-interactive provider can start sending information for any supported item and domain. For the LSEG Real-Time Distribution System adopters, the non-interactive provider is similar in concept to what was once called the Src-Driven, or Broadcast Server Application.

Non-interactive providers act like clients in a client-server relationship. Multiple non-interactive providers can connect to the same LSEG Real-Time Distribution System and publish the same items and content. For example, two non-interactive providers can publish the same or different fields for the same item “INTC.O” to the same LSEG Real-Time Distribution System.

Non-interactive provider applications can connect using a point-to-point TCP-based transport as shown in Figure 18, or using a multicast transport as shown in Figure 19.

The main benefit of this scenario is that all publishing traffic flows from top to bottom: the way a system normally expects updating data to flow. In the local publishing scenario, posting is frequently done upstream and must contend with a potential Infrastructure bias in prioritization of upstream versus downstream traffic.

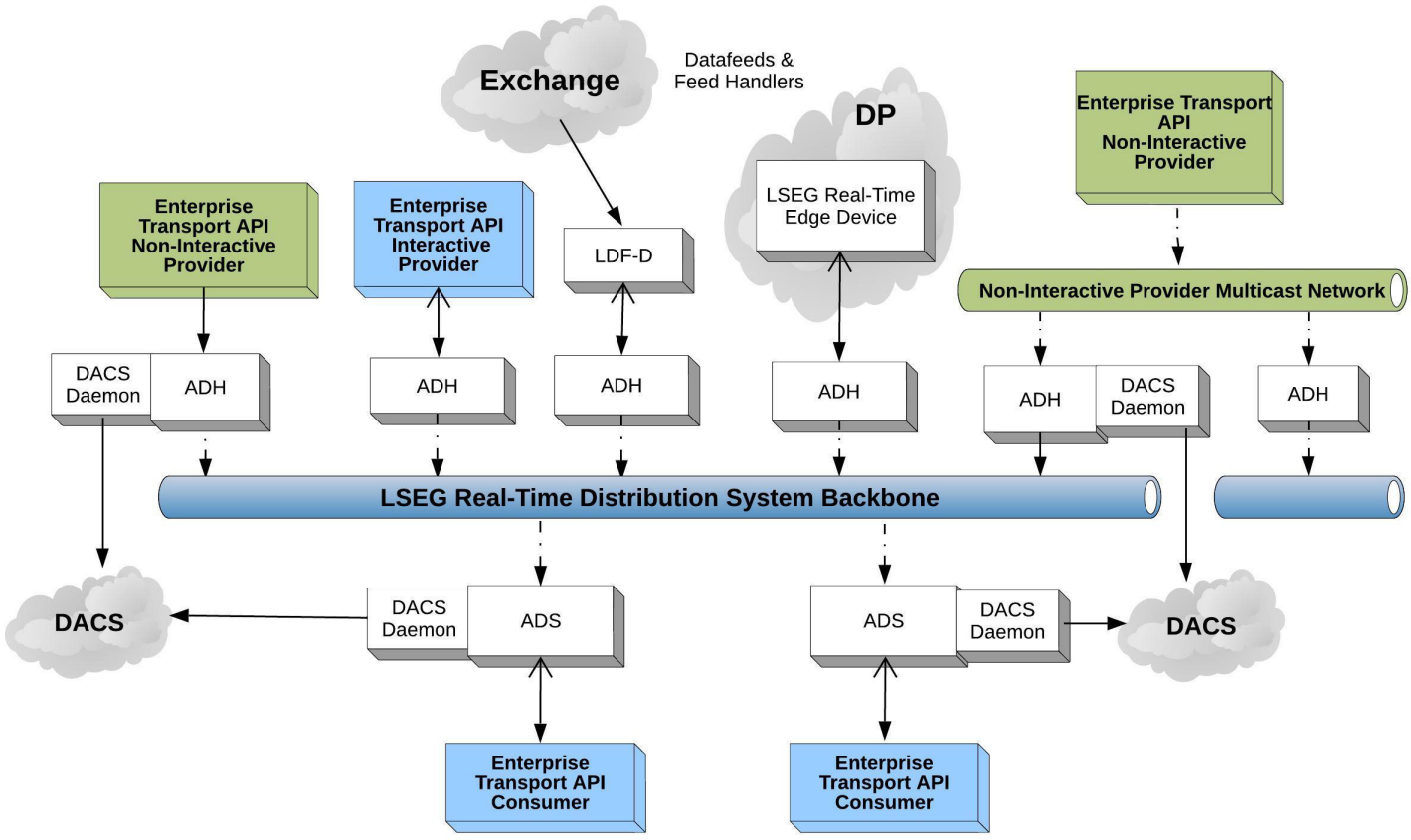
4 System View

4.1 System Architecture Overview

An LSEG Real-Time Distribution System network typically hosts the following:

- Core Infrastructure: LSEG Real-Time Advanced Distribution Server (ADS), LSEG Real-Time Advanced Distribution Hub (ADH), etc.
- Consumer applications that typically request and receive information from the network
- Provider applications that typically write information to the network. Provider applications fall into one of two categories:
  - Interactive provider applications which receive and interpret request messages and reply back with any needed information.
  - Non-interactive provider applications which publish data, regardless of user requests or which applications consume the data.
- Permissioning infrastructure: Data Access Control System
- Devices that interact with the markets: Data Feed Direct and LSEG Real-Time Edge Device

The following figure illustrates a typical deployment of an LSEG Real-Time Distribution System network and some of its possible components. Components that use the Enterprise Transport API could alternatively choose to leverage RFA, depending on user needs and required access levels. The remainder of this chapter briefly describes the components pictured in the diagram and explains how the Enterprise Transport API integrates with each.



Legend:

ADH:	LSEG Real-Time Advanced Distribution Hub
ADS:	LSEG Real-Time Advanced Distribution Server
DACS:	Data Access Control System
LDF-D:	LSEG Data Feed Direct
DP:	Delivery Platform

Figure 20. Typical LSEG Real-Time Distribution System Components

4.2 LSEG Real-Time Advanced Distribution Server

The LSEG Real-Time Advanced Distribution Server provides a consolidated distribution solution for LSEG, value-added, and third-party data for trading-room systems. It distributes information using the same Open Message Model and Rssl Wire Format protocols exposed by the Enterprise Transport API.

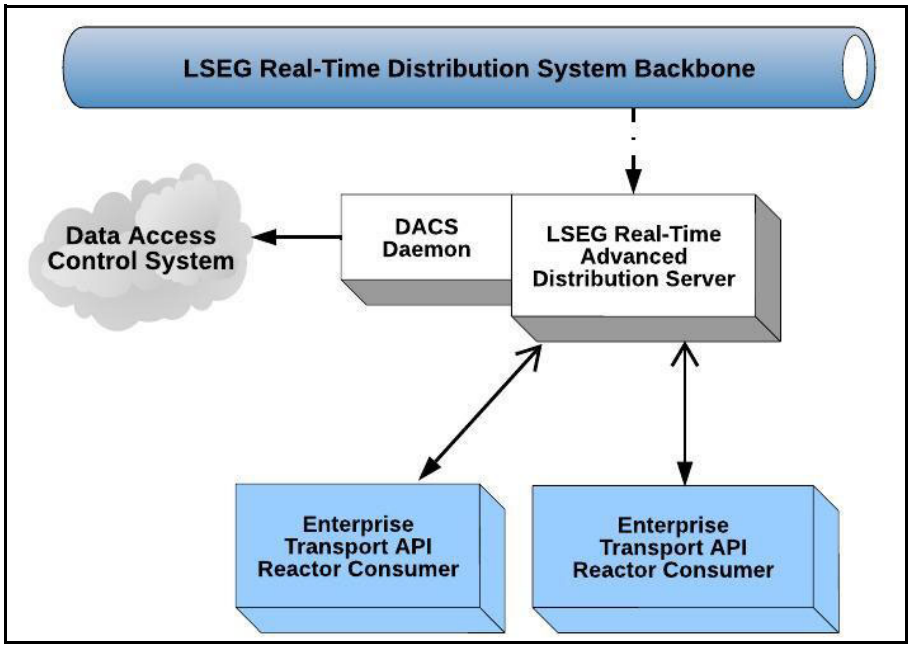


Figure 21. Enterprise Transport API and LSEG Real-Time Advanced Distribution Server

As a distribution device for market data, the LSEG Real-Time Advanced Distribution Server delivers data from the LSEG Real-Time Advanced Distribution Hub. Because the LSEG Real-Time Advanced Distribution Server leverages multiple threads, it can offload the encoding, fan out, and writing of client data. By distributing its tasks in this fashion, LSEG Real-Time Advanced Distribution Server can support a large number of client applications.

The LSEG Real-Time Advanced Distribution Server communicates with its API clients via point-to-point communication.

4.3 LSEG Real-Time Advanced Distribution Hub

The ***LSEG Real-Time Advanced Distribution Hub*** is a networked, data distribution server that runs in the LSEG Real-Time Distribution System. It consumes data from a variety of content providers and reliably fans this data out to multiple LSEG Real-Time Advanced Distribution Servers over a multicast backbone. Enterprise Transport API-based non-interactive or interactive provider applications can publish content directly into an LSEG Real-Time Advanced Distribution Hub, thus distributing data more widely across the network. Non-interactive provider applications can publish content to an LSEG Real-Time Advanced Distribution Hub via TCP or multicast connection types.

The LSEG Real-Time Advanced Distribution Hub leverages multiple threads, both for inbound traffic processing and outbound data fanout. By leveraging multiple threads, the LSEG Real-Time Advanced Distribution Hub can offload the overhead associated with request and response processing, caching, data conflation, and fault tolerance management. By offloading overhead in such a fashion, the LSEG Real-Time Advanced Distribution Hub can support high throughputs.

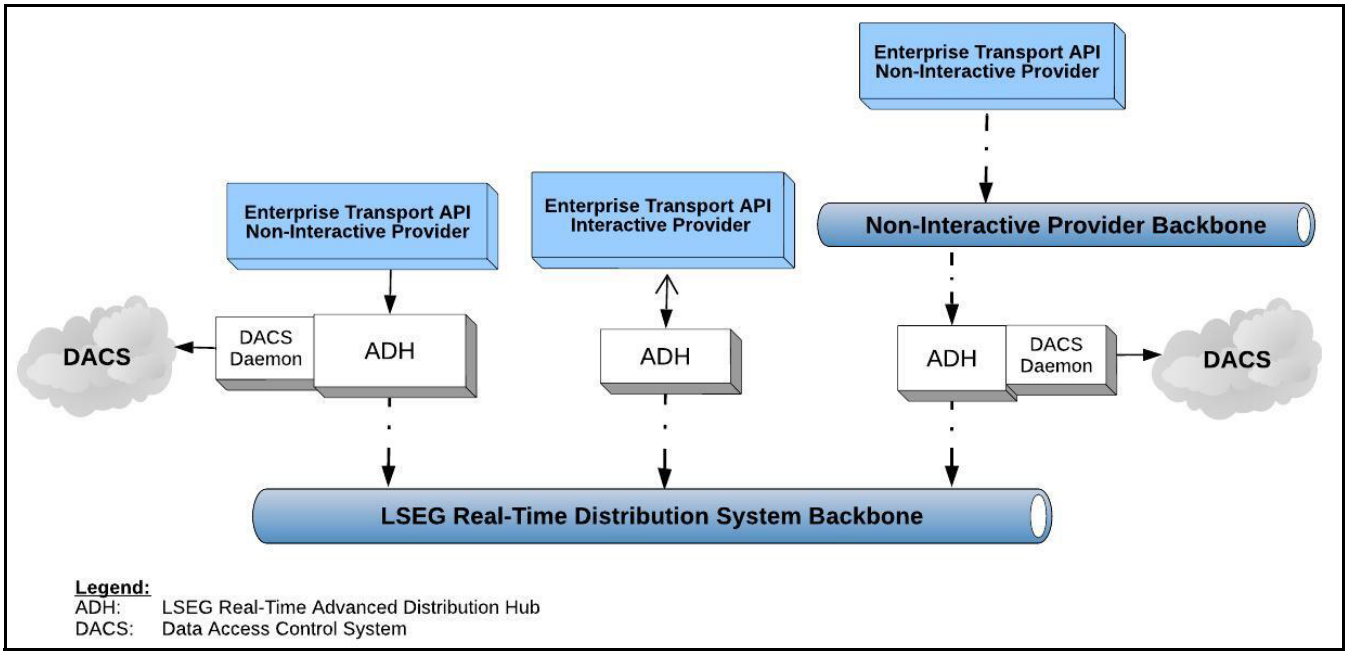


Figure 22. Enterprise Transport API and the LSEG Real-Time Advanced Distribution Hub

4.4 LSEG Real-Time and the Delivery Platform

The ***Delivery Platform*** is an open, global, ultra-high-speed network and hosting environment, which allows users to access and share a variety of content including Real-Time data. The Delivery Platform allows access to information from a wide network of content providers, including exchanges, where all exchange data is normalized using the Open Message Model.

Real-Time content, one of the content sets available via the Delivery Platform, can be obtained by consuming applications written to any Real-Time API or by connecting to on-prem LSEG Real-Time Distribution Systems (i.e., cascaded LSEG Real-Time Advanced Distribution Hub and LSEG Real-Time Advanced Distribution Server). Consumer applications authenticate and can discover endpoints via the Delivery Platform and use that information to connect to Real-Time -- Optimized (LSEG's cloud offering) which ultimately sources data from LSEG Real-Time infrastructure.

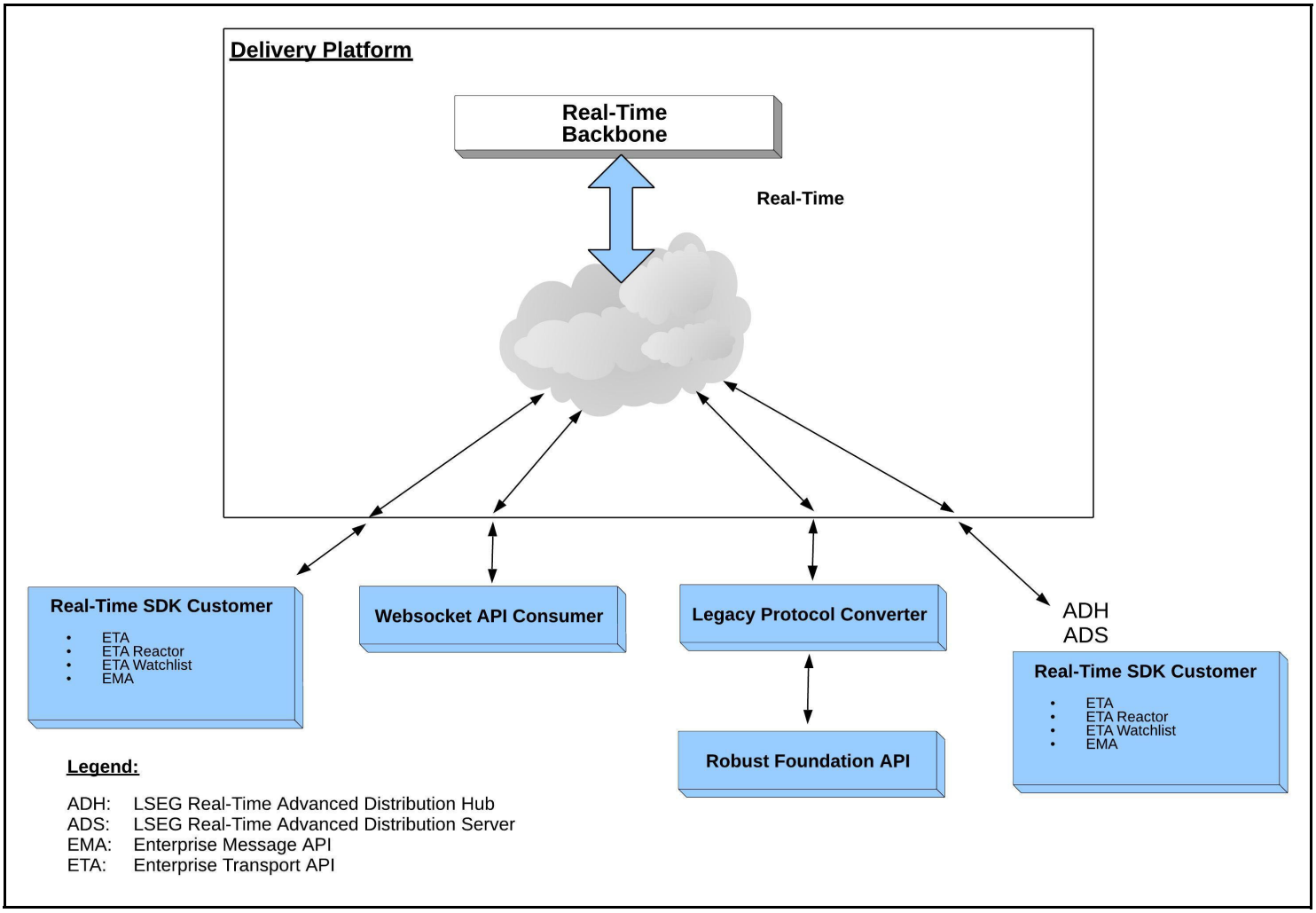


Figure 23. LSEG Real-Tme APIs and Delivery Platform

4.5 Data Feed Direct

Data Feed Direct is a fully managed LSEG exchange feed providing an ultra-low-latency solution for consuming data from specific exchanges. The Data Feed Direct normalizes all exchange data using the Open Message Model.

To access this content, a Enterprise Transport API consumer application can connect directly to the Data Feed Direct or via a cascaded LSEG Real-Time Distribution System architecture.

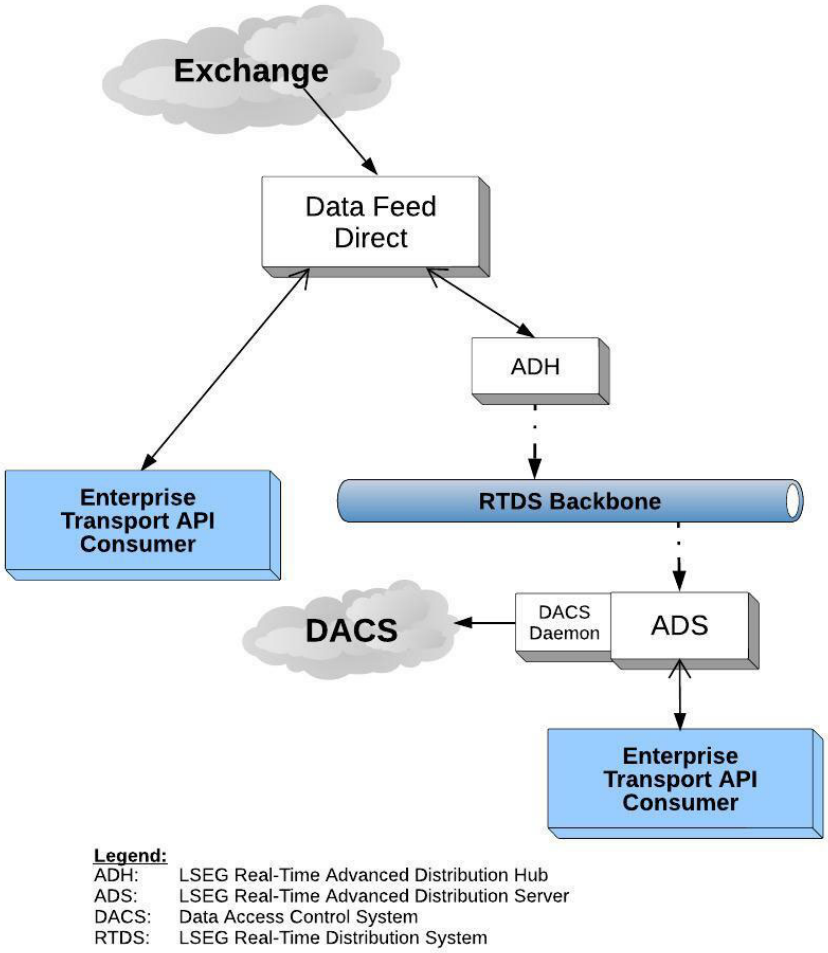


Figure 24. Enterprise Transport API and Data Feed Direct

4.6 Internet Connectivity via HTTP and HTTPS

Consumer and provider applications can use the Enterprise Transport API to establish encrypted connections or HTTPS (on certain platforms) over the public Internet.

- Consumer and non-interactive provider applications can establish connections via HTTP tunneling, socket and websocket connections.
- LSEG Real-Time Advanced Distribution Servers and OMM interactive provider applications can accept incoming Enterprise Transport API connections tunneled via HTTP (such functionality is available across all supported platforms).
- Consumer applications can leverage HTTPS to establish an encrypted tunnel to certain LSEG hosted solutions, performing key and certificate exchange.

For further details, refer to Section 10.15.

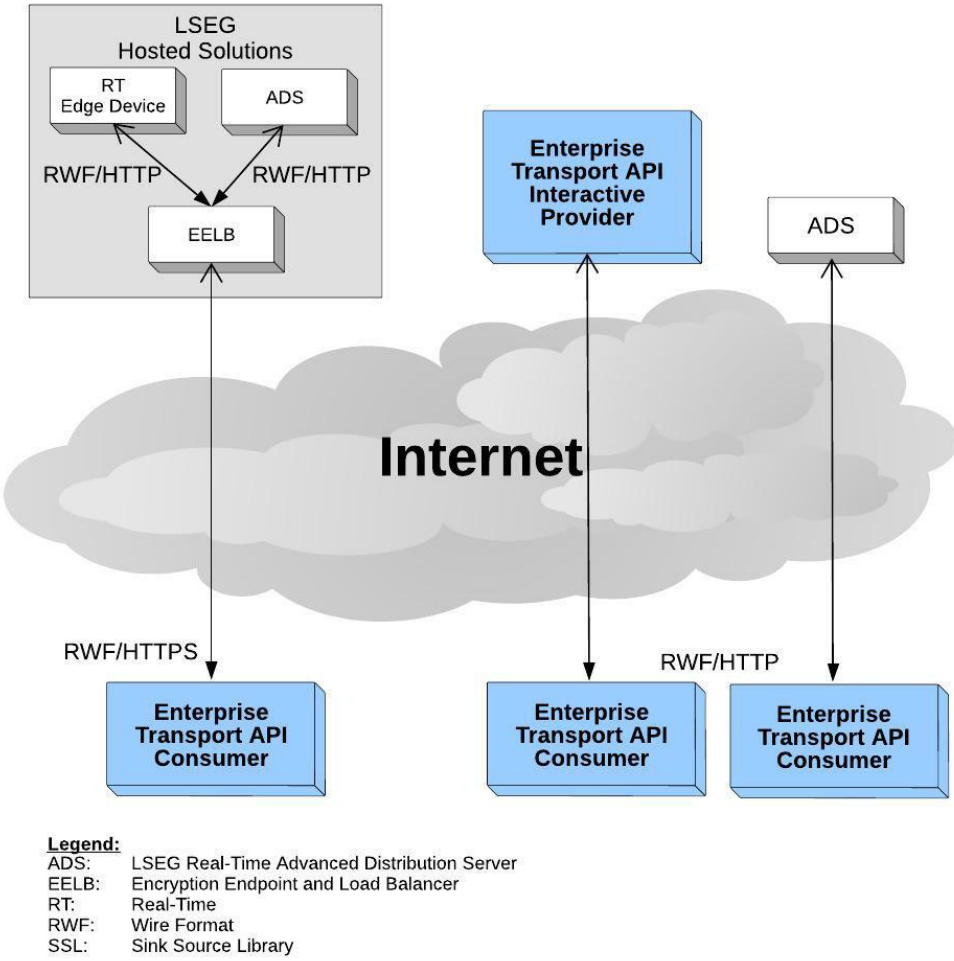


Figure 25. Enterprise Transport API and Internet Connectivity

4.7 Direct Connections

The Enterprise Transport API allows OMM interactive provider applications and consumer applications to directly connect to one another. This includes Open Message Model applications written to RFA. The following diagram illustrates various direct connect combinations.

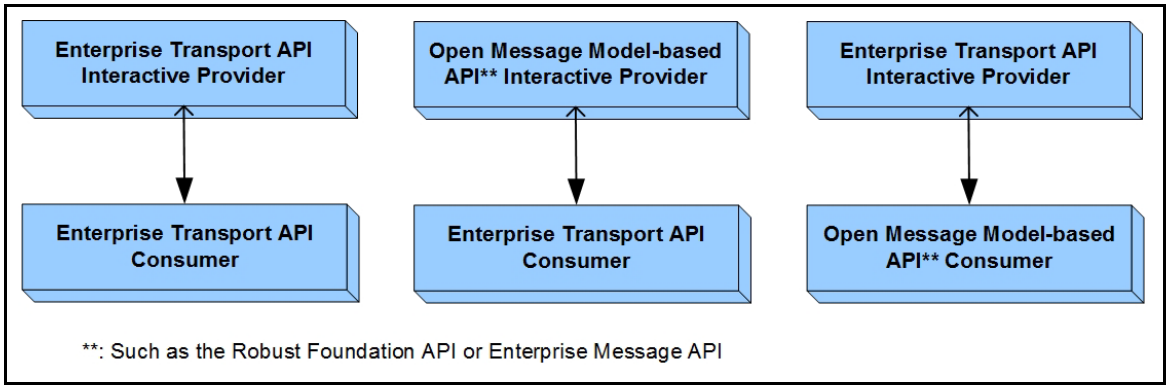


Figure 26. Transport API and Direct Connections

5 Model and Package Overviews

5.1 Enterprise Transport API Models

5.1.1 Open Message Model

The ***Open Message Model*** is a collection of message header and data constructs. Some Open Message Model message header constructs (such as the Update message) have implicit market logic associated with them, while others (such as the Generic message) allow for free-flowing bi-directional messaging. You can combine Open Message Model data constructs in various ways to model data ranging from simple (i.e., flat) primitive types to complex multi-level hierarchal data.

The layout and interpretation of any specific Open Message Model (also referred to as a domain model) is described within that model's definition and is not coupled with the API. The Open Message Model is a flexible and simple tool that provides the building blocks to design and produce domain models to meet the needs of the system and its users. The Enterprise Transport API provides structural representations of Open Message Model constructs and manages the Rssl Wire Format binary-encoded representation of the Open Message Model. Users can leverage LSEG-provided Open Message Model constructs to consume or provide Open Message Model data throughout the LSEG Real-Time Distribution System.

5.1.2 RWF

Rssl Wire Format is the encoded representation of the Open Message Model; a highly-optimized, binary format designed to reduce the cost of data distribution compared to previous wire formats. Binary encoding represents data in the machine's native manner, enabling further use in calculations or data manipulations. Rssl Wire Format allows for serializing Open Message Model message and data constructs in an efficient manner while still allowing you to model rich content types. You can use Rssl Wire Format to distribute field identifier-value pair data (similar to Marketfeed), self-describing data (similar to Qform), as well as more complex, nested hierarchal content.

5.1.3 Domain Message Model

A Domain Message Model describes a specific arrangement of Open Message Model message and data constructs. A Domain Message Model defines any:

- Specialized behavior associated with the domain
- Specific meanings or semantics associated with the message data

Unless a Domain Message Model specifies otherwise, any implicit market logic associated with a message still applies (e.g., an Update message indicates that previously received data is being modified by corresponding data from the Update message).

5.1.3.1 Domain Model

A ***Domain Model*** is a domain message model typically provided or consumed by an LSEG product (i.e., LSEG Real-Time Distribution System, Data Feed Direct, or LSEG Real-Time). Some currently-defined Domain Models allow for authenticating to a provider (e.g., Login), exchanging field or enumeration dictionaries (e.g., Dictionary), and providing or consuming various types of market data (e.g., Market Price, Market by Order, Market by Price). LSEG's defined models have a domain value of less than 128. For extended definitions of the currently-defined Domain Models, refer to the *Transport API Domain Model Usage Guide*.

5.1.3.2 User-Defined Domain Model

A ***User-Defined Domain Model*** is a Domain Message Model defined by a third party. These might be defined to solve a need specific to a user or system in a particular deployment and which is not resolved through the use of a Domain Model. Any user-defined model must use a domain value between 128 and 255.

Customers can have their domain model designer work with LSEG to define their model as a standard Domain Model. Working directly with LSEG can help ensure interoperability with future Domain Model definitions and with other LSEG products.

5.2 Packages

The Enterprise Transport API consists of several packages, each serving a different purpose within an application. While some packages are interdependent, others can be used alone or with other packages. Each package serves a distinct purpose as described in the following sections.

As needs evolve, additional packages can be added to the Enterprise Transport API.

5.2.1 Transport Package

The ***Transport Package*** provides a mechanism to efficiently distribute messages across a variety of communication protocols. This package provides a receiver-transparent way for senders to combine or pack multiple messages into one outbound packet, and it will internally fragment and reassemble messages which exceed the size of an outbound packet. This package exposes structural representations to manage connection properties and information. The Transport Package includes interface functions that assist with establishing connections and the sending or receiving of data. This package utilizes some header files from the Data Package, but has no other dependencies other than system libraries.

To access all transport functionality, an application must use the LSEG.Eta.Transports namespace.

The Transport Package is described in more detail in Section 10.

5.2.2 Codec Package

The ***Codec Package*** defines object-oriented representations for everything you need to encode and decode Open Message Model content. This includes definitions that:

- Expose data types (primitive and container types) and manage their Rssl Wire Format binary representation. These data types in turn make up components of Open Message Model data.
  - Primitive types are simple, atomically updating constructs, usually provided by the operating system (e.g., Integer, Date).
  - Container types can model more complex data and be modified more granularly than a primitive type (e.g., field identifier-value pairs, key-value pairs, self-describing name-value pairs).
- Expose message classes and manage their Rssl Wire Format binary-encoded representation. The Codec defines message header elements that flow between various applications in LSEG Real-Time Distribution System (e.g., update messages). Some header elements are standard to the market data environment (such as conflation information, state information, permission information, and item key elements used for stream identification). Message headers contain generic attributes in which usage and meaning are defined within specific DMMs (e.g., Market Price, Market By Order). All messages can carry payload information of varying format and layouts.

To access codec package functionality, an application must use the LSEG.Eta.Codec namespace.

The codec package is described with more detail in Chapter 11 and Chapter 12.

6 Building an OMM Consumer

6.1 Overview

This chapter provides an overview of how to create an consumer application. A consumer application can establish a connection to other interactive provider applications, including the LSEG Real-Time Distribution System, Data Feed Direct, and Delivery Platform. After connecting successfully, a consumer can then consume (i.e., send data requests and receive responses) and publish data (i.e., post data) or forward data (i.e., Round Trip Time messages).

The following steps summarize the general process:

1. Establish network communication.
2. Log in.
3. Obtain source directory information.
4. Load or download all necessary dictionary information.
5. Issue requests, process responses, forward generic messages, and/or post information.
6. Log out and shut down.

The **Consumer** example application included with the Enterprise Transport API products provides an example implementation of a consumer application. The application is written with simplicity in mind and demonstrates the uses of the Enterprise Transport API. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

6.2 Establish Network Communication

The first step of any Enterprise Transport API consumer application is to establish a network connection with its peer component (i.e., another application with which to interact). A consumer typically creates an outbound connection to the well-known hostname and port of an Interactive Provider. The consumer uses the **`Transport.Connect()`** function to initiate the connection and then performs any additional connection initialization processes as described in this document.

After the consumer's connection is active, ping messages might need to be exchanged. The negotiated ping timeout is available via the **`IChannel`**. The connection can be terminated if ping heartbeats are not sent or received within the expected time frame. LSEG recommends sending ping messages at intervals one third the size of the ping timeout.

Detailed information and use case examples for using various transports provided by the Enterprise Transport API are specified in Chapter.

6.3 Perform Login Process

Applications authenticate with one another using the Login domain model. A consumer must register with the system using a Login request prior to issuing any other requests or opening any other streams.

After receiving a Login request, an interactive provider determines whether a user is permissioned to access the system. The interactive provider sends back a Login response, indicating to the consumer whether access is granted.

- If the application is denied, the Login stream is closed, and the consumer application cannot send additional requests.
- If the application is granted access, the Login response contains information about available features, such as Posting, Pause and Resume, and the use of Dynamic Views. The consumer application can use this information to tailor its interaction with the provider.

Content is encoded and decoded using the Message Package (described in Chapter) and the Data Package (described in Chapter). Further information about Login domain usage and messaging is available in the *Transport API LSEG Domain Model Usage Guide*.

6.4 Obtain Source Directory Information

The Source Directory domain model conveys information about all available services in the system. A consumer typically requests a Source Directory to retrieve information about available services and their capabilities. This includes information about supported domain types, the service's state, the quality of service, and any item group information associated with the service. At minimum, LSEG recommends that the application requests the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains the service name and **`serviceId`** information for all available services. When the consumer discovers an appropriate service, it uses the service's **`serviceId`** on all subsequent requests to that service.
- The Source Directory State filter contains status information for service, which informs the consumer whether the service is Up and available, or Down and unavailable.
- The Source Directory Group filter conveys item group status information, including information about group states, as well as the merging of groups. Additional information on item groups is available in .

Content is encoded and decoded using the Enterprise Transport API's Message Package (as described in Chapter) and Data Package (as described in Chapter). Information about the Source Directory domain and its associated filter entry content is available in the *Enterprise Transport API LSEG Domain Model Usage Guide*.

6.5 Load or Download Necessary Dictionary Information

Some data requires the use of a dictionary for encoding or decoding. This dictionary typically defines type and formatting information and directs the application as to how to encode or decode specific pieces of information. Content that uses the **`FieldList`** type requires the use of a field dictionary (usually the **RDMFieldDictionary**, though it could also be a user-defined or modified field dictionary).

A source directory message should provide information about:

- Any dictionaries required to decode the content provided on a service.
- Which dictionaries are available for download.

A consumer application can determine whether to load necessary dictionary information from a local file or download the information from the provider if available.

- If loading from a file, the Enterprise Transport API offers several utility functions to load and manage a properly-formatted field dictionary.
- If downloading information, the application issues a request using the Dictionary domain model. The provider application should respond with a dictionary response. Because a dictionary response often contains a large amount of content, it is typically broken into a multi-part message. the Enterprise Transport API offers several utility functions for encoding and decoding of the Dictionary domain content.

For information on the utility functions used in both instances and for information about the Dictionary domain and its expected content formats, refer to the *Transport API LSEG Domain Model Usage Guide*.

Content is encoded and decoded using the Enterprise Transport API Message Package (as described in) and the Enterprise Transport API Data Package (as described in).

6.6 Issue Requests and/or Post Information

After the consumer application successfully logs in and obtains Source Directory and Dictionary information, it can request additional content. When issuing the request, the consuming application can use the **`serviceId`** of the desired service, along with the stream's identifying information. Requests can be sent for any domain using the formats defined in that domain model specification. Domains provided by LSEG are defined in the *Enterprise Transport API LSEG Domain Model Usage Guide*.

At this point, a consumer application can also post information to capable provider applications. For more information, refer to Section 13.9.

Content is encoded and decoded using the Enterprise Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View).

6.7 Log Out and Shut Down

When the consumer application is done retrieving, forwarding, or posting content, it should close all open streams and shut down the network connection. Issuing an **`ICloseMsg`** for the **`streamId`** associated with the Login closes all streams opened by the consumer.

- For more information on closing streams, refer to Section 12.2.5.
- For information on the Message Package, refer to Chapter 12, Message Package Detailed View.

When shutting down the consumer, the application should release any unwritten pool buffers. Calling **`IChannel.Close`** terminates the connection to the provider application. Detailed information and transport code examples are provided in Chapter 10, Transport Package Detailed View.

6.8 Additional Consumer Details

The following locations provide specific details about using consumers and the Enterprise Transport API:

- The **Consumer** application demonstrates one way of implementing of a consumer application. The application's source code contain additional information about specific implementation and behaviors.
- For reviewing high-level encoding and decoding concepts, refer to Chapter 9, Encoding and Decoding Conventions.
- For a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 10, Transport Package Detailed View.
- For a detailed look at the Data Package, typically used for encoding and decoding payload content, refer to Chapter 11, Data Package Detailed View.
- For a detailed look at the Message Package, used for all message encoding and decoding, refer to Chapter 12, Message Package Detailed View.
- For specific information about the Domain Message Models required by this application type, refer to the *Enterprise Transport API LSEG Domain Model Usage Guide*.

7 Building an OMM Interactive Provider

7.1 Overview

This chapter provides a high-level description of how to create an OMM interactive provider application. An OMM interactive provider application opens a listening socket on a well-known port allowing consumer applications to connect. After connecting, consumers can request data from the interactive provider.

The following steps summarize this process:

- Establish network communication
- Accept incoming connections
- Handle login requests
- Provide source directory information
- Provide or download necessary dictionaries
- Handle requests and post messages
- Dispatch Round Trip Time messages
- Sends out messages for round trip latency monitoring.
- Disconnect consumers and shut down

The **Provider** example application included with the Enterprise Transport API package provides one way of implementing an OMM interactive provider. The application is written with simplicity in mind and demonstrates the use of the Enterprise Transport API. Portions of the functionality are abstracted for easy reuse, though you might need to customize it to achieve your own unique performance and functionality goals.

7.2 Establish Network Communication

The first step of any Enterprise Transport API Interactive Provider application is to establish a listening socket, usually on a well-known port so that consumer applications can easily connect. The provider uses the **`Transport.Bind()`** function to open the port and listen for incoming connection attempts.

Whenever a consumer application attempts to connect, the provider uses the **`IServer.Accept`** function to begin the connection initialization process.

Once the connection is active, the consumer and provider applications might need to exchange ping messages. A negotiated ping timeout is available via **`IChannel`** corresponding to each connection (this value might differ on a per-connection basis). The provider may choose to terminate a connection if ping heartbeats are not sent or received within the expected time frame. LSEG recommends sending ping messages at intervals one-third the size of the ping timeout.

For detailed information and use cases for the various transports provided by the Enterprise Transport API, refer to Chapter 10, Transport Package Detailed View.

7.3 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM interactive provider must handle the consumer's Login request messages and supply appropriate responses.

After receiving a Login request, the interactive provider can perform any necessary authentication and permissioning.

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

Content is encoded and decoded using the Transport API Message Package (as described in) and the Transport API Data Package (as described in). For further information on Login domain usage and messaging, refer to the *Enterprise Transport API LSEG Domain Model Usage Guide* .

7.4 Provide Source Directory Information

The Source Directory domain model conveys information about all available services in the system. A consumer typically requests a Source Directory to retrieve information about available services and their capabilities. This includes information about supported domain types, the service's state, the Quality of Service, and any item group information associated with the service. LSEG recommends that at a minimum, an interactive provider supply the Info, State, and Group filters for the Source Directory.

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

Content is encoded and decoded using the Enterprise Transport API's Message Package (as described in Chapter) and Data Package (as described in Chapter). For details on the Source Directory domain and all of its associated filter entry content, refer to the *Enterprise Transport API LSEG Domain Model Usage Guide*.

7.5 Provide or Download Necessary Dictionaries

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

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

If connected to a supporting LSEG Real-Time Advanced Distribution Hub, a provider application may also download the RWFFld and RWFEnum dictionaries to retrieve the appropriate dictionary information for providing field list content. A provider can use this feature to ensure it has the appropriate version of the dictionary or to encode data. The LSEG Real-Time Advanced Distribution Hub that supports the Provider Dictionary Download feature sends a Login request message containing the **SupportProviderDictionaryDownload** login element. The dictionary request is sent using the Dictionary domain model.[1]

The Enterprise Transport API offers several utility functions for loading, downloading, and managing a properly-formatted field dictionary. There are also utility functions provided to help the provider encode into an appropriate format for downloading or decoding downloaded dictionary.

Content is encoded and decoded using the Enterprise Transport API Message Package (as described in Chapter) and the Transport API Data Package (as described in Chapter).

Information about the Login and Dictionary domains, their expected content and formatting, and dictionary utility functions, is available in the *Enterprise Transport API LSEG Domain Model Usage Guide*.

7.6 Handle Requests and Post Messages

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

- Comparing **IMsgKey** identification information received against the content available from the provider.
- Determining whether it can provide the requested Quality of Service.
- Ensuring that the consumer does not already have a stream open for the requested information.

If a provider can service a request, it should send appropriate responses. However, if the provider cannot satisfy the request, the provider should send an **IStatusMsg** to indicate the reason and close the stream. All requests and responses should follow specific formatting as defined in the domain model specification. For details on all domains provided by LSEG, refer to the *Enterprise Transport API LSEG Domain Model Usage Guide*.

If a provider application receives a Post message, the provider should determine the correct handling for the post. This depends on the application's role in the system and might involve storing the post in its cache or passing it farther up into the system. If the provider is the destination for the Post, the provider should send any requested acknowledgments, following the guidelines described in .

Content is typically encoded and decoded using the Enterprise Transport API's Message Package (as described in Chapter) and Data Package (as described in Chapter).

1. Because this is instantiated by the provider, the application should use a **streamId** with a negative value. Additional details are provided in subsequent chapters.

7.7 Disconnect Consumers and Shut Down

When shutting down, the provider application should close the listening socket by calling **`IServer.Close`**. Closing the listening socket prevents new connection attempts. The provider application can either leave consumer connections intact or shut them down.

If the provider decides to close consumer connections, the provider should send an **`IStatusMsg`** on each connection's login stream, thus closing the stream. At this point, the consumer should assume that its other open streams are also closed. The provider should then release any unwritten pool buffers it has obtained from **`IChannel.GetBuffer`** and call **`IChannel.Close`** for each connected client.

For detailed information and use case examples for the transport, refer to Chapter 10, Transport Package Detailed View.

7.8 Additional Interactive Provider Details

For specific details about OMM interactive providers and the Enterprise Transport API use, refer to the following locations:

- The **Provider** application demonstrates one implementation of an OMM interactive provider application. The application's source code has additional information about specific implementation and behaviors.
- To review high-level encoding and decoding concepts, refer to Chapter 9, Encoding and Decoding Conventions.
- For a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 10, Transport Package Detailed View.
- For a detailed look at the Data Package, typically used for encoding and decoding payload content, refer to Chapter 11, Data Package Detailed View.
- For a detailed look at the Message Package, used for all message encoding and decoding, refer to Chapter 12, Message Package Detailed View.
- For specific information about Domain Message Models required by this application type, refer to the *Enterprise Transport API C# Edition LSEG Domain Model Usage Guide*.

8 Building an OMM Non-Interactive Provider

8.1 Overview

This chapter provides an outline of how to create an OMM non-interactive provider application which can establish a connection to an LSEG Real-Time Advanced Distribution Hub server. Once connected, an non-interactive provider can publish information into the LSEG Real-Time Advanced Distribution Hub cache without needing to handle requests for the information. The LSEG Real-Time Advanced Distribution Hub can cache the information and along with other LSEG Real-Time Distribution System components, provide the information to any consumer applications that indicate interest.

The general process can be summarized by the following steps:

- Establish network communication
- Perform Login process
- Perform Dictionary Download
- Provide Source Directory information
- Provide content
- Log out and shut down

Included with the Enterprise Transport API package, the **NIProvider** example application provides an implementation of an non-interactive provider written with simplicity in mind and demonstrates the use of the Enterprise Transport API. Portions of the functionality are abstracted for easy reuse, though you might need to modify it to achieve your own performance and functionality goals.

Content is encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View).

8.2 Establish Network Communication

The first step of any non-interactive provider application is to establish network communication with an LSEG Real-Time Advanced Data Hub server. To do so, the OMM non-interactive provider typically creates an outbound connection to the well-known hostname and port of an LSEG Real-Time Advanced Distribution Hub. The non-interactive provider application uses the **`Transport.Connect()`** method to initiate the connection process and then performs connection initialization processes as described in this document.

After establishing a connection, ping messages might need to be exchanged. The negotiated ping timeout is available via the **`IChannel`**. If ping heartbeats are not sent or received within the expected time frame, the connection can be terminated. LSEG recommends sending ping messages at intervals one-third the size of the ping timeout.

For detailed information on the various transports provided by the Enterprise Transport API and associated use case examples, refer to Chapter 10, Transport Package Detailed View.

8.3 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM non-interactive provider must register with the system using a Login request[1] prior to providing any content.

After receiving a Login request, the LSEG Real-Time Advanced Distribution Hub determines whether the non-interactive provider is permissioned to access the system. The LSEG Real-Time Advanced Distribution Hub sends a Login response to the non-interactive provider which indicates whether the LSEG Real-Time Advanced Distribution Hub has granted it access. If the application is denied, the LSEG Real-Time Advanced Distribution Hub closes the Login stream and the non-interactive provider application cannot perform any additional

1. Because this is done in an interactive manner, the non-interactive provider should assign a **`streamId`** with a positive value (which the LSEG Real-Time Advanced Distribution Hub will reference) when sending its response.

communication. If the application gains access to the LSEG Real-Time Advanced Distribution Hub, the Login response informs the application of this. The provider must now provide a Source Directory and/or download dictionary.

For details on using the Login domain and expected message content, refer to the *Enterprise Transport API LSEG Domain Model Usage Guide*.

8.4 Perform Dictionary Download

If connected to an LSEG Real-Time Advanced Distribution Hub that support dictionary downloads, an OMM non-interactive provider can download the RWFFId and RWFEnum dictionaries to retrieve appropriate information when providing field list content. A Non-Interactive Provider can use this feature to ensure they are using the correct version of the dictionary or to encode data. An LSEG Real-Time Advanced Distribution Hub that supports the Provider Dictionary Download feature sends a Login response message containing the **`SupportProviderDictionaryDownload`** login element. The dictionary request is send using the Dictionary domain model[2].

The Transport API offers several utility functions you can use to download and manage a properly-formatted field dictionary. The API also includes other utility functions that help the provider encode into an appropriate format for downloading or decoding a downloaded dictionary.

For details on using the Login domain, expected message content, and dictionary utility functions, refer to the Enterprise Transport API *LSEG Data Models Usage Guide*.

8.5 Provide Source Directory Information

The Source Directory domain model conveys information about all available services in the system. After completing the Login process, an OMM non-interactive provider must provide a Source Directory refresh[3] indicating:

- Service, service state, Quality of Service, and capability information associated with the non-interactive provider.
- Supported domain types and any item group information associated with the service.

At a minimum, LSEG recommends that the non-interactive provider send the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains service name and **`serviceId`** information for all available services, though non-interactive providers typically provide data on only one service.
- The Source Directory State filter contains status information for service. This informs the LSEG Real-Time Advanced Distribution Hub whether the service is Up and available or Down and unavailable.
- The Source Directory Group filter conveys item group status information, including information about group states as well as the merging of groups. For additional information about item groups, refer to Section 13.4.

For details on the Source Directory domain and all of its associated filter entry content, refer to the *Enterprise Transport API LSEG Domain Model Usage Guide*.

8.6 Provide Content

After providing a Source Directory, the non-interactive provider application can begin pushing content to the LSEG Real-Time Advanced Distribution Hub. Each unique information stream should begin with an **`IRefreshMsg`**, conveying all necessary identification information for the content[4]. The initial identifying refresh can be followed by other status or update messages. Some LSEG Real-Time Advanced Distribution Hub functionality, such as cache rebuilding, may require that non-interactive provider applications publish the message key on all **`IRefreshMsg`**s. For more information, refer to component-specific documentation.

---

2. Because this is instantiated by the provider, the application should use a **`streamId`** with a negative value.
3. Because this is instantiated by the provider, the non-interactive provider should use a **`streamId`** with a negative value.
4. Because the provider instantiates these information streams, a negative value **`streamId`** should be used for each stream. Additional details are provided in subsequent chapters.

NOTE: Some components, depending on their specific functionality and configuration, require that non-interactive provider applications publish the **msgKey** in **UpdateMsg**s. To avoid component or transport migration issues, non-interactive provider applications can choose to always include this information, however this incurs additional bandwidth use and overhead. When designing your application, read the documentation for your other components to ensure that you take into account any other requirements.

Content is typically encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View).

8.7 Log Out and Shut Down

After publishing content to the system, the non-interactive provider application should close all open streams and shut down the network connection.

- For more information about closing streams, refer to Section 12.2.5.
- For information about the Message Package, refer to Chapter 12, Message Package Detailed View.

When shutting down the provider, the application should release all unwritten pool buffers. Calling terminates the connection to the LSEG Real-Time Advanced Distribution Hub. Detailed information for transport and associated use cases are provided in Chapter 10, Transport Package Detailed View.

8.8 Additional Non-Interactive Provider Details

For specific details about OMM non-interactive providers and Enterprise Transport API use, refer to the following locations:

- The **NIProvider** application demonstrates one implementation of an OMM non-interactive provider application. The application's source code has additional information about specific implementation and behaviors.
- For reviewing high-level encoding and decoding concepts, refer to Chapter 9, Encoding and Decoding Conventions.
- For a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 10, Transport Package Detailed View.
- For a detailed look at the Data Package, typically used for encoding and decoding payload content, refer to Chapter 11, Data Package Detailed View.
- For a detailed look at the Message Package, used for all message encoding and decoding, refer to Chapter 12, Message Package Detailed View.
- For specific information about the Domain Message Models required by the application, refer to the *Enterprise Transport API C# Edition LSEG Domain Model Usage Guide*.

9 Encoding and Decoding Conventions

9.1 Concepts

The Enterprise Transport API Codec package allows the user to encode and decode constructs and various content. The Codec Package defines a single encode iterator type and a single decode iterator type. The Enterprise Transport API supports single-iterator encoding and decoding such that a single instance can encode or decode the full depth and breadth of a user's content. The application controls the depth of decoding, so you can skip content of no interest. Less efficiently, you can continue to leverage the Enterprise Transport API to use separate iterator instances and hence allow the user to separate portions of content across iterators when encoding or decoding.

The Codec package does not provide inherent threading or locking capability. Separate iterator and type instances do not cause contention and do not share resources between instances. Any needed threading, locking, or thread-model implementation is at the discretion of the application. Different application threads can encode or decode different messages without requiring a lock; thus each thread must use its own iterator instance and each message should be encoded or decoded using unique and independent buffers. Though possible, LSEG recommends that you do not encode or decode related messages (ones that flow on the same stream) on different threads as this can scramble the delivery order.

9.1.1 Data Types

The Enterprise Transport API offers a wide variety of data types categorized into two groups:

- ***Primitive Types***: A primitive type represents simple, atomically updating information such as values like integers, dates, and ASCII string buffers (refer to Section 11.2).
- ***Container Types***: A container type can model data representations more intricately and manage dynamic content at a more granular level than primitive types. Container types represent complex information such as field identifier-value, name-value, or key-value pairs (refer to Section 11.3). The Enterprise Transport API offers several uniform, homogeneous container types (i.e., all entries house the same type of data). Additionally, there are several non-uniform, heterogeneous container types in which different entries can hold different types of data.

9.1.2 Composite Pattern of Data Types

The following diagram illustrates the use of Enterprise Transport API data types to resemble a composite pattern.

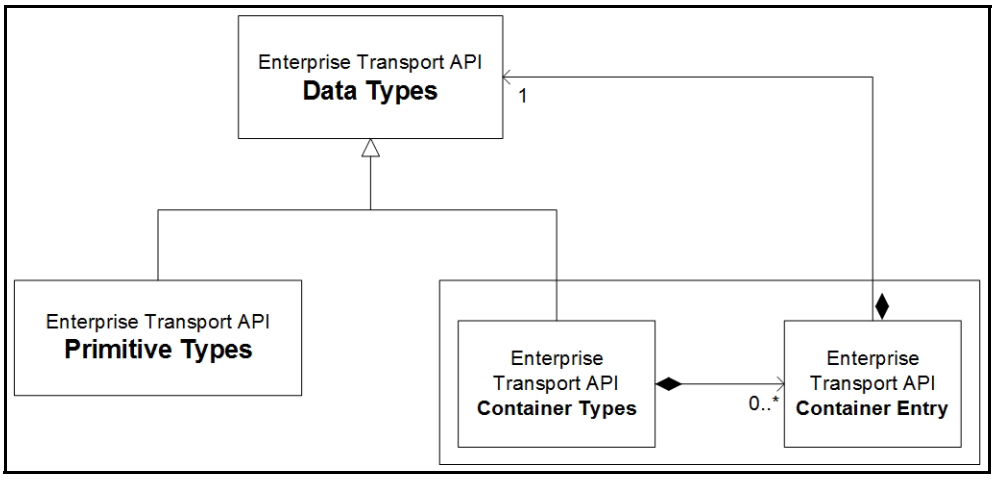


Figure 27. Enterprise Transport API and the Composite Pattern

The diagram highlights the following:

- Being made up of both primitive and container types, Enterprise Transport API data type values mirror the composite pattern's component.
- Enterprise Transport API primitive types mimic the composite pattern's leaf, conveying concrete information for the user.
- The Enterprise Transport API container type and its entries are similar to the composite pattern's composite. This allows for housing other container types and, in some cases such as field and element lists, housing primitive types.

The housing of other types is also referred to as ***nesting***. Nesting allows:

- Messages to house other messages or container types
- Container types to house other messages, container, or primitive types

This provides the flexibility for domain model definitions and applications to arrange and nest data types in whatever way best achieves their goals.

9.2 Encoding Semantics

Because the Transport API supports several styles of encoding, the user can choose whichever method best fits their needs.

9.2.1 Init and Complete Suffixes

Encoding functions that have a suffix of **`Init`** or **`Complete`** (e.g. **`FieldEntry.EncodeInit()`** and **`FieldEntry.EncodeComplete()`**) allow the user to encode the type part-by-part, serializing each portion of data with each called function.

Functions without a suffix of **`Init`** or **`Complete`** (e.g. **`FieldEntry.Encode()`**, **`Int.Encode()`**, or **`Msg.Encode()`**) perform encoding within a single call, typically used for encoding simple types like Integer or incorporating previously encoded data (referred to as ***pre-encoded*** data).

9.2.2 The Encode Iterator: EncodeIterator

To encode content you must use an **EncodeIterator** and can use a single encode iterator to manage the entire encoding process[1] (including state and position information).

For example, if you want to encode a message that contains an **FieldList** composed of various primitive types, you can use the same **EncodeIterator** to encode all contents. In this case, initialize the iterator before encoding the message, and then pass the iterator as a parameter when encoding each portion. You do not need to perform additional initialization or clearing. When encoding finishes, you can determine the total encoded length and clear the iterator, reusing it for another encoding. If needed, you can use individual iterators for each level of encoding or for pre-encoding portions of data. However, when using separate iterators, you must initialize each iterator before starting the associated encoding process.

Initialization of an **EncodeIterator** consists of several steps. After creating the iterator, clear it using **EncodeIterator.Clear()**. Each **EncodeIterator** requires a **ITransportBuffer** (provided via **EncodeIterator.SetBufferAndRWFVersion()**) into which it encodes. Rssl Wire Format version information can also be populated on the iterator, ensuring that the proper version of the wire format is encoded (refer to Section 9.5.1).

9.2.2.1 EncodeIterator Functions

NOTE: Additional encoding examples are provided throughout this manual as well as in the Enterprise Transport API package's example applications.

The following table describes functions that you can use with **EncodeIterator**.

FUNCTION NAME	DESCRIPTION
EncodeIterator.Clear()	Clears members necessary for encoding and readies the iterator for reuse. You must clear the **EncodeIterator** prior to starting any encoding process. For performance purposes, only those members necessary for proper functionality are cleared.
EncodeIterator.SetBufferAndRWFVersion()	Associates an **EncodeIterator** with the **ITransportBuffer** into which it encodes. **ITransportBuffer.Data** should refer to sufficient space for encoding. Rssl Wire Format Versioning information ensures that the Enterprise Transport API uses the appropriate wire format version while encoding. Rssl Wire Format information is typically available on the connection between applications. Refer to Section 9.5.1.
SetEncodeIteratorRWFVersion	Associates RWF Versioning information to the **EncodeIterator**, ensuring that The Transport API uses the appropriate wire format version while encoding. Rssl Wire Format information is typically available on the connection between applications. Refer to Section 9.5.1.
RealignBuffer	If an encoding process exceeds the space allocated in the current **ITransportBuffer**, this function dynamically associates a new, larger buffer with the encoding process, allowing encoding to continue.

Table 4: EncodeIterator Utility Functions

1. A single **EncodeIterator** can support up to sixteen levels of nesting, allowing for sixteen **Init** calls without a single **Complete** call. Because the most complex Domain Model currently requires only five levels, sixteen is believed to be sufficient. Should an encoding require more than sixteen levels of nesting, multiple iterators can be used.

9.2.2.2 EncodeIterator: Basic Use Example

The following example illustrates how to initialize **EncodeIterator** in a typical fashion and set the buffer's length once encoding completes.

```
/* create and clear iterator to prepare for encoding */
EncodeIterator encodeIter = new EncodeIterator();
encodeIter.Clear();

/* associate buffer and iterator (code assumes buffer has sufficient memory) and set proper protocol
    version information on iterator (typically obtained from Channel associated with the connection
    once it becomes active)*/
CodecReturnCode ret;
if ((ret = encodeIter.SetBufferAndRWFVersion(buffer, chnl.MajorVersion, chnl.MinorVersion)) <
CodecReturnCode.SUCCESS)
{
    System.Console.WriteLine("EncodeIterator.SetBufferAndRWFVersion() failed with return code: {0}",
    ret.GetAsString());
}

/* Perform all content encoding now that iterator is prepared. */
```

Code Example 1: EncodeIterator Usage Example

9.2.3 Content Roll Back with Example

Every **Complete** method has a **success** parameter, which allows you to discard unsuccessfully encoded content and roll back to the last successfully encoded portion.

For example, you begin encoding a list that contains multiple entries, but the tenth entry in the list fails to encode. To salvage the successful portion of the encoding, pass the **success** parameter as **false** when calling the failed entry's **Complete** method. This rolls back encoding to the end of the last successful entry. The remaining **Complete** methods should be called, after which the application can use the encoded content. You can begin a new encoding for the remaining entries.

The following example demonstrates the use of the roll back procedure. This example encodes an **Map** with two entries. The first entry succeeds; so **success** is passed in as **true**. However, encoding the second entry's contents fails, so the second map entry is rolled back, and the map is completed. To highlight the rollback feature, only those portions relevant to the example are included.

```
/* example shows encoding a map with two entries, where second entry content fails so it is
    rolled back */
retCode = map.EncodeInit(encIter, 0, 0);

/* Encode the first map entry - this one succeeds */
retCode = mapEntry.EncodeInit(encIter, entryKey, 0);

/* encode contents - assume this succeeds */
/* Passing true for the success parameter completes encoding of this entry */
retCode = mapEntry.EncodeComplete(encIter, true);

/* Encode the second map entry - this one fails */
retCode = mapEntry.EncodeInit(encIter, entryKey, 0);

/* encode contents - assume this fails */
/* Passing false for the success parameter rolls back the encoding to the end of the previous entry */
retCode = mapEntry.EncodeComplete(encIter, false);

/* Now complete encoding of the map - this results in only one entry being contained in the map */
retCode = map.EncodeComplete(encIter, true);
```

Code Example 2: Encoding Rollback Example

9.3 Decoding Semantics and DecodeIterator

Using the Enterprise Transport API, applications can decode the full depth of the content or skip over portions in which the application is not interested. Each container type provided by the Enterprise Transport API includes functionality for decoding the container header and decoding each entry in the container. If an application wishes to decode information present in a container entry, it can invoke the specific decode function associated with the nested type. When nested content is completely decoded, the next container entry can be decoded. If an application wishes to skip decoding data nested in a container entry, it can simply call the container entry decode method again without invoking the decoder for nested content. A decoding application will typically loop on decode until **CodecReturnCode.END_OF_CONTAINER** is returned.

9.3.1 The Decode Iterator: DecodeIterator

All decoding requires the use of a **DecodeIterator**. You can use a single decode iterator to manage the full decoding process, internally managing various state and position information while decoding.

For example, when decoding a message that contains a **FieldList** composed of various primitive types, you can use the same **DecodeIterator** to decode all contents, including primitive types. In this case, you want to initialize the iterator before decoding the message and then pass the iterator as a parameter when decoding other portions (without additional initialization or clearing). After you completely decode all needed content, you can clear the iterator and reuse it for another decoding. If needed, you can use individual iterators for each level of decoding. However, if you use separate iterators, you must initialize each iterator before the decoding process that it manages.

Initialization of a **DecodeIterator** consists of several steps. After the iterator is created, use **Clear** to clear **DecodeIterator**. Each **DecodeIterator** requires an **ITransportBuffer** (provided via **SetBufferAndRWFVersion**) from which to decode. Rssl Wire Format version information can also be populated on the iterator, thus decoding the appropriate version of the wire format (refer to Section 9.5.1).

9.3.2 Functions for use with DecodeIterator

The following table describes the methods that you can use with **DecodeIterator**. The methods listed below that take **ITransportBuffer**s also support **Buffer**s (from the Codec package). **ITransportBuffer**s are used with the Transport package. **Buffer**s (from the Codec package) are available for use with user-defined transports.

METHOD	DESCRIPTION
Clear	Clears members necessary for decoding and readies the iterator for reuse. You must clear **DecodeIterator** before decoding content. For performance purposes, only those members required for proper functionality are cleared.
SetBufferAndRWFVersion	Associates the **DecodeIterator** with the **ITransportBuffer** from which to decode and Rssl Wire Format version information. Set **ITransportBuffer.data** to refer to the content to be decoded. Rssl Wire Format Version information ensures that the Enterprise Transport API uses the appropriate wire format version when encoding. Rssl Wire Format information is typically available on the connection between applications. Refer to Section 9.5.1.
FinishDecodeEntries	The decoding process typically runs until the end of each container, indicated by **CodecReturnCode.END_OF_CONTAINER**. This method will skip past remaining entries in the container and perform necessary synchronization between the content and iterator so that decoding can continue.

Table 5: DecodeIterator Utility Methods

9.3.3 DecodeIterator: Basic Use Example

The following example demonstrates a typical **DecodeIterator** initialization process.

```
/* create and clear iterator to prepare for decoding */
DecodeIterator decodeIter = new DecodeIterator();
decodeIter.Clear();
/* associate buffer and iterator (code assumes buffer has sufficient memory) and set proper
protocol version information on iterator (typically obtained from Channel associated with
the connection once it becomes active)
*/

CodecReturnCode ret;
if ((ret = decodeIter.SetBufferAndRWFVersion(buffer, chnl.MajorVersion, chnl.MinorVersion)) <
CodecReturnCode.SUCCESS)
{
 System.Console.WriteLine($"DecodeIterator.SetBufferAndRWFVersion() failed with return code:
     {ret.GetAsString()}");
}
/* Perform all content decoding now that iterator is prepared. */
```

Code Example 3: DecodeIterator Usage Example

9.4 Return Code Values

Codec functionality returns codes indicating success or failure.

- On failure conditions, these codes inform the user of the error.
- On success conditions, these codes provide the application additional direction regarding the next encoding steps.

When using the Codec package, return codes greater than or equal to **CodecReturnCode.SUCCESS** indicate some type of specific success code, while codes less than **CodecReturnCode.SUCCESS** indicate some type of specific failure.

NOTE: The Transport Layer has special semantics associated with its return codes. It does not follow the same semantics as the Codec package. For detailed handling instructions and return code information, refer to Chapter 10, Transport Package Detailed View.

9.4.1 Success Codes

The following table describes success values of return codes associated with the Codec.

RETURN CODE	DESCRIPTION
CodecReturnCode.SUCCESS	Indicates operational success. Does not indicate next steps, though additional encoding or decoding might be required.
CodecReturnCode.ENCODE_MSG_KEY_ATTRIB	Indicates that initial message encoding was successful and now the application should encode **IMsgKey** attributes. This return occurs if the application indicates that the message should include **IMsgKey** attributes when calling **Msg.EncodeInit (MsgKeyFlags.HAS_ATTRIB)** without populating pre-encoded data into **MsgKey.EncAttrib**. For further details, refer to Section 12.1.3 and Code Example 41.
CodecReturnCode.ENCODE_EXTENDED_HEADER	Indicates that initial message encoding (and **msgKey** attribute encoding) was successful, and the application should now encode **ExtendedHeader** content. This return occurs if an application indicates that the message should include **extendedHeader** content when calling **Msg.EncodeInit** without populating pre-encoded data into the **extendedHeader**. For further details on message encoding information, refer to Chapter 12, Message Package Detailed View.
CodecReturnCode.ENCODE_CONTAINER	Indicates that initial encoding succeeded and that the application should now encode the specified **containerType**. • For details on container types, refer to Section 11.3. • For details on encoding messages, refer to Chapter 12, Message Package Detailed View.
CodecReturnCode.SET_COMPLETE	Indicates that **FieldList** or **ElementList** encoding is complete. Additionally encoded entries are encoded in the standard way with no additional data optimizations. For further information, refer to Section 11.6.
CodecReturnCode.DICT_PART_ENCODED	Indicates that the dictionary encoding utility method successfully encoded part of a dictionary message (because dictionary messages tend to be large, they might be segmented into a multi-part message). • For specific information about the Dictionary domain and the utility functions provided by the Transport API, refer to the *Enterprise Transport API C# Edition LSEG Domain Model Usage Guide*. • For more details on message fragmentation, refer to Section 13.1.

Table 6: Codec Package Success

RETURN CODE	DESCRIPTION
CodecReturnCode.BLANK_DATA	Indicates that the decoded **primitiveType** is a blank value. The contents of the primitive type should be ignored; any display or calculation should treat the value as blank. For further details on primitive types, refer to Section 11.2.
CodecReturnCode.NO_DATA	Indicates that the **containerType** being decoded contains no data and was decoded from an empty payload. Informs the application not to continue to decode container entries (as none exist).
CodecReturnCode.END_OF_CONTAINER	Indicates that the decoding process has reached the end of the current container. If decoding nested content, additional decoding might still be needed. The application can move back up the nesting stack and continue decoding the next container entry by calling the container's specific entry decode method. For example, if decoding an **FieldList** contained in an **MapEntry**, when this code is returned, it signifies that the contained field list decoding is complete. For details on container types, refer to Section 11.3.
CodecReturnCode.SET_SKIPPED	Indicates that **FieldList** or **ElementList** decoding skipped over contained, set-defined data because a set definition database was not provided. Any standard encoded entries will still be decoded. For further information on set definitions, refer to Section 11.6.
CodecReturnCode.SET_DEF_DB_EMPTY	Indicates that decoding of a set definition database completed successfully, but the database was empty. For further information, refer to Section 11.6.

Table 6: Codec Package Success (Continued)

9.4.2 Failure Codes

RETURN CODE	DESCRIPTION
CodecReturnCode.FAILURE	Indicates a general failure, used when no specific details are available.
CodecReturnCode.BUFFER_TOO_SMALL	Indicates that the **ITransportBuffer** on the **EncodeIterator** lacks sufficient space for encoding.
CodecReturnCode.INVALID_ARGUMENT	Indicates an invalid argument was provided to an encoding or decoding method.
CodecReturnCode.ENCODING_UNAVAILABLE	Indicates that the invoked method does not contain encoding functionality for the specified type. There might be other ways to encode content or the type might be invalid in the combination being performed.
CodecReturnCode.UNSUPPORTED_DATA_TYPE	Indicates that the type is not supported for the operation being performed. This might indicate a **PrimitiveType** is used where a **ContainerType** is expected or the opposite.
CodecReturnCode.UNEXPECTED_ENCODER_CALL	Indicates that encoding functionality was used in an unexpected sequence or the called method is not expected in this encoding.
CodecReturnCode.INCOMPLETE_DATA	Indicates that the **ITransportBuffer** on the **DecodeIterator** does not have enough data for proper decoding.
CodecReturnCode.INVALID_DATA	Indicates that invalid data was provided to the invoked method.

Table 7: Codec Package Failure Return Codes

RETURN CODE	DESCRIPTION
CodecReturnCode.ITERATOR_OVERRUN	Indicates that the application is attempting to nest more levels of content than is supported by a single **EncodeIterator**[a]. If this occurs, you should use multiple iterators for encoding.
CodecReturnCode.VALUE_OUT_OF_RANGE	Indicates that a value being encoded using a set definition exceeds the allowable range for the type as specified in the definition. For further information on set definitions, refer to Section 11.6.
CodecReturnCode.SET_DEF_NOT_PROVIDED	Indicates that **FieldList** or **ElementList** encoding requires a set definition database which was not provided. For more information, refer to Section 11.6.
CodecReturnCode.TOO_MANY_LOCAL_SET_DEFS	Indicates that encoding exceeds the maximum number of allowed local set definitions. Currently 15 local set definitions are allowed per database. For more information, refer to Section 11.6.
CodecReturnCode.DUPLICATE_LOCAL_SET_DEFS	Indicates that content includes a duplicate set definition that collides with a definition already stored in the database. For more information, refer to Section 11.6.
CodecReturnCode.ILLEGAL_LOCAL_SET_DEF	Indicates that the **SetId** associated with a contained definition exceeds the allowable value. Currently **SetId** values up to **15** are allowed. For more information, refer to Section 11.6.

Table 7: Codec Package Failure Return Codes (Continued)

a. A single **EncodeIterator** can support up to sixteen levels of nesting (this allows for sixteen **Init** calls without a single **Complete** call). Currently, the most complex Domain Model requires five levels, so sixteen is sufficient. If an encoding requires more than sixteen levels of nesting, multiple iterators can be employed.

9.5 Versioning

The Transport API supports two types of versioning:

- Protocol Versioning: Allows for the exchange of protocol type and version information across a connection established with the Transport Package. Protocol and version information can be provided to the **`EncodeIterator`** and **`DecodeIterator`** to ensure the proper handling and use of the appropriate wire format version.

  NOTE: LSEG strongly recommends that you write all Enterprise Transport API applications to leverage wire format versioning.

- Library Versioning: Allows for applications to programmatically query library version information. Library versioning ensures that expected libraries are used and that all versions match in the application.

9.5.1 Protocol Versioning

Consumer and provider applications using the Transport can provide protocol type and version information. This data is supplied as part of **`ConnectOptions`** or **`BindOptions`** and populated via the **`ProtocolType`**, **`MajorVersion`**, and **`MinorVersion`** methods. When establishing a connection, data is exchanged and negotiated between client and server:ProtocolType.

- If the client's specified **`ProtocolType`** does not match the server's specified **`ProtocolType`**, the connection is refused.
- If the **`ProtocolType`** information matches, version information is compared and a compatible version determined.

After a connection becomes active, negotiated version information is available via the **`IChannel`** from both client and server and can be used for encoding and decoding:

- To populate version information on an **`EncodeIterator`**, call the **`EncodeIterator.SetBufferAndRWFVersion`** method.
- To populate version information on an **`DecodeIterator`**, call the **`DecodeIterator.SetBufferAndRWFVersion`** method.

The Transport layer is data neutral and does not change or depend on data distribution. Versioning information is provided only to help client and server applications manage the data they communicate. For further details on the Transport, refer to Chapter 10, Transport Package Detailed View.

NOTE: Properly using Enterprise Transport API's versioning functionality helps minimize future impacts associated with underlying format modifications and enhancements, ensuring compatibility with other Enterprise Transport API-enabled components.

Typically, an increase in the major version is associated with the introduction of an incompatible change. An increase in the minor version tends to signify the introduction of a compatible change or extension.MinorVersion

The Codec Package contains several defined values that you can use with protocol versioning:

	DESCRIPTION
Codec.ProtocolType	Defines the **`ProtocolType`** value associated with Rssl Wire Format. Define other protocols using different **`ProtocolType`** values.
Codec.MajorVersion	Sets the value associated with the current major version. If incompatible changes are introduced, this value is incremented.
Codec.MinorVersion	Sets the value associated with the current minor version. If extensions or compatible changes are introduced, this value is incremented.

Table 8: Codec Methods

9.5.2 Library Versioning

Each Transport API library embeds its own version data as well as internal LSEG build version data. Any issues raised to support should include this version data.

MEMBER	DESCRIPTION
ProductVersion	Contains the library's version.
ProductInternalVersion	Contains the internal LSEG build data.
ProductDate	Contains the build date for the product release.

Table 9: `ILibraryVersionInfo` Methods

Additionally, each Transport API library includes a **`Transport.QueryVersion`** function which you can programmatically extract library version information from **`ILibraryVersionInfo`**.

10 Transport Package Detailed View

10.1 Concepts

The Enterprise Transport API offers a Transport Package capable of communicating with other Open Message Model-based components, including but not limited to LSEG Real-Time Distribution System, LSEG Real-Time, EDF Direct, and other LSEG Real-Time Distribution System API Open Message Model-based applications. The Transport Package efficiently sends and receives data across TCP/IP-based networks, connection types, and presents a message-based interface to applications for ease of reading and writing data.

The package exposes a feature set that includes a receiver-transparent way for senders to combine or pack multiple messages into one outbound packet, as well as transparent fragmentation and reassembly of messages which exceed the size of an outbound packet. Class representations are provided for managing connections (referred to as channels).

The transport layer offers multiple degrees of thread safety, all programmatically configurable by the application. This ranges from a fully thread-safe option[1] to the ability for an application to turn off all protective locking[2]. Threading implementation and thread-model selection is managed by the application. The transport provides different locking options to provide maximum flexibility to the user. For more information, refer to Section 10.2.3.

The transport supports both non-blocking and blocking I/O models, however use of blocking I/O is not recommended. When a blocking operation is occurring, control will not be returned to the application until the operation has fully completed (e.g. all information is written). This prevents the application from performing additional tasks, including heartbeat sending and monitoring, while the transport operation may be waiting for the operating system. By employing an I/O notification mechanism (e.g. select, poll), an application can leverage a non-blocking I/O model, using the I/O notification to alert the application when data is available to read or when output space is available for writing to. The following sections are written with an emphasis on non-blocking I/O use, though blocking behavior is also described. All examples are written from a non-blocking I/O perspective.

10.1.1 Transport Types

The transport supports configuration of multiple connection types for different systems, while providing a single interface for a look and feel that is similar among all connections and components. Developers should ensure that the components to which they intend to connect are configured to support the appropriate transport type.

10.1.1.1 Socket Transport

The Enterprise Transport API provides a transport for efficiently distributing messages across a TCP/IP-based reliable network (**ConnectionType.SOCKET**). This transport is capable of connecting to various Open Message Model-based components, including but not limited to LSEG Real-Time Distribution System, LSEG Real-Time, LDF Direct, and other Enterprise Transport API or RFA Open Message Model-based applications. On specific platforms, applications can also leverage TLS Encrypted (**ConnectionType.ENCRYPTED**) connection types for Internet connectivity.

The socket transport allows for both establishing outbound connections and for creating listening sockets to accept inbound connections. Once a connection is established, both connected components can send and receive information. Outbound connections are typically created by OMM consumer applications to connect to an LSEG Real-Time Advanced Distribution Server or OMM interactive provider, or by OMM non-interactive provider applications to connect to an LSEG Real-Time Advanced Distribution Hub. Listening sockets are typically created by OMM interactive provider applications to allow OMM consumer applications or LSEG Real-Time Advanced Distribution Servers to instantiate connections to it and request data.

1. When this option is enabled, Transport can function correctly during simultaneous execution by multiple application threads.
2. When this option is enabled, all locking is disabled for additional performance. If required, the application must provide any necessary thread safety.

10.1.2 IChannel Object

The **IChannel** object represents a connection that can send or receive information across a network, regardless of whether the connection is outbound or accepted by a listening socket. The Transport Package internally manages any memory associated with an **IChannel** object, and the application does not need to create nor free memory (associated with the channel). **IChannel** is typically used to perform any action on the connection that it represents (e.g. reading, writing, disconnecting, etc). See the subsequent sections for more information about **IChannel** use within the transport.

For information on how to set the Channel connection types using the **ConnectOptions** parameters, refer to Section 10.1.2.3.

The following table describes the methods of the **IChannel** object.

METHOD	DESCRIPTION
Blocking	A Boolean representing the blocking mode of **IChannel**.
BufferUsage	Gets the total number of used buffers for this **IChannel**.
Close	Close the **IChannel**.
ConnectionType	An enumerated value that indicates the type of underlying connection being used. For more information, refer to Section 10.1.2.2.
Flush	Flush this **IChannel**. For more details, refer to Section 10.10.2.
GetBuffer	Retrieves an **ITransportBuffer** for use. For more details, refer to Section 10.9.
HostName	Provides the name of the host to which a consumer or Non-Interactive Provider application connects.
Info	Gets information about this **IChannel**. For more details, refer to Section 10.14.2.
Init	Continues **IChannel** initialization for non-blocking channels. For more details, refer to Section 10.5.
IOCtl	Set or get some I/O values programmatically. For more details, refer to Section 10.14.6.
MajorVersion	When an **IChannel** becomes active for a client or server, this is populated with the negotiated major version number that is associated with the content being sent on this connection. Typically, a major version increase is associated with the introduction of incompatible change. The transport layer is data-neutral and allows the flow of any type of content. **majorVersion** is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
MinorVersion	When an **IChannel** becomes active for a client or server, this is populated with the negotiated minor version number that is associated with the content being sent on this connection. Typically, a minor version increase is associated with a fully backward compatible change or extension. The transport layer is data-neutral and allows the flow of any type of content. **minorVersion** is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
OldSocketId	It is possible for a file descriptor to change over time, typically due to some kind of connection keep-alive mechanism. If this occurs, this is typically communicated via a return code of **TransportReturnCode.READ_FD_CHANGE** (for further information, refer to Section 10.6). The previous **Socket** is stored in **OldSocket** so the application can properly unregister and then register the new **Socket** with their I/O notification mechanism.
PackBuffer	For more details, refer to Section 10.11.
Ping	Send a ping (i.e. heart beat) message to the far end of the connection.

Table 10: IChannel Methods

METHOD	DESCRIPTION
PingTimeOut	When an **IChannel** becomes active for a client or server, this is populated with the negotiated ping timeout value. This is the number of seconds after which no communication can result in a connection being terminated. Both client and server applications should send heartbeat information within this interval. The typically used rule of thumb is to send a heartbeat every **pingTimeout**/3 seconds. For more information, refer to Section 10.12.
Port	Provides the server port number to which the consumer or Non-Interactive Provider application is connected.
ProtocolType	When an **IChannel** becomes active for a client or server, this is populated with the **ProtocolType** associated with the content being sent on this connection (i.e., **ProtocolType.RWF**). If the **ProtocolType** indicated by a server does not match the **ProtocolType** that a client specifies, the connection will be rejected. The transport layer is data-neutral and allows the flow of any type of content. **ProtocolType** is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
Read	Read on this **IChannel**. For more details, refer to Section 10.6.
ReleaseBuffer	Release an **ITransportBuffer**. Should only be used if the buffer could not be successfully written.
Socket	Represents a file descriptor that can be used in some kind of I/O notification mechanism (e.g. select, poll). This is the file descriptor associated with this end of the network connection; the file descriptor value may be different from the other end of the connection.
State	The state associated with the **IChannel**. Until the channel has completed its initialization handshake and has transitioned to an active state, no reading or writing can be performed. Table 11 describes channel state values.
UserSpecObject	A reference that can be set by the user of the **IChannel**. This value can be set directly or via the connection options and is not modified by the transport. This information can be useful for coupling this **IChannel** with other user created information, such as a watch list associated with this connection.
Write	Write on this **IChannel**. For more details, refer to Section 10.9.

Table 10: IChannel Methods(Continued)

10.1.2.1 IChannel State Values

STATE	DESCRIPTION
ACTIVE	Indicates that an **IChannel** is active. This channel can perform any connection-related actions, such as reading or writing.
CLOSED	Indicates that an **IChannel** has been closed. This typically occurs as a result of an error inside of a transport method call and is often related to a socket being closed or becoming unavailable. Appropriate error value return codes and **Error** information should be available for the user.
INACTIVE	Indicates that an **IChannel** is inactive. This channel cannot be used. This state typically occurs after a channel is closed by the user.
INITIALIZING	Indicates that an **IChannel** requires additional initialization. This initialization is typically additional connection handshake messages that need to be exchanged. When using blocking I/O, an **IChannel** is typically active when it is returned and no additional initialization is required by the user.

Table 11: IChannel State Values

10.1.2.2 ConnectionType Values

Connection types are used in several areas of the transport. When creating a connection, an application can specify which connection type to use (refer to Section 10.3). Additionally, after a connection is established, the **`IChannel.ConnectionType`** will indicate the connection type being used.

CONNECTIONTYPE	DESCRIPTION
ConnectionType.SOCKET	Indicates that the **`IChannel`** uses a standard, TCP-based socket connection. This type can be used to connect between any Transport-based applications.
ConnectionType.ENCRYPTED	Indicates that the **`IChannel`** tunnels using encryption. The encryption use is transparent to the client application. For a server to accept encrypted connection types the use of an external encryption/decryption device is required (encryption / decryption is not performed by the server). For more information, refer to Section 4.6.

Table 12: ConnectionType Values

10.1.2.3 Channel Connection Types

The following table summarizes possible Channel connection types and the **ConnectOptions** parameter values that you can use to set them.

CHANNEL CONNECTION TYPE	CONNECTOPTIONS.CONNECTIONTYPE	ENCRYPTIONOPTIONS.ENCRYPTEDPROTOCOL	SUBPROTOCOL LIST POSSIBILITIES
Unencrypted Socket	ConnectionType.SOCKET	Not used	Not used; RWF is implied.
Encrypted Socket	ConnectionType.ENCRYPTED	**ConnectOptions .EncryptionOpts .EncryptedProtocol= ConnectionType . SOCKET.**	

Table 13: Channel Settings for Socket Connection Types

10.1.3 IServer Object

The **IServer** object is used to represent a server that is listening for incoming connection requests. Any memory associated with an **IServer** object is internally managed by the Transport Package, and the application does not need to create nor destroy this type. The **IServer** is typically used to accept or reject incoming connection attempts. The following table describes the members of the **IServer** structure.

For information on how to set the Server connection types using the **BindOptions** parameters, refer to Section 10.1.3.1.

For more information about **IServer** use within the transport, see the subsequent sections.

METHOD	DESCRIPTION
SocketId	Represents a Socket object that can be used in some kind of I/O notification mechanism (e.g. select, poll). This is the file descriptor associated with listening socket. When triggered, this typically indicates that there is an incoming connection and **IServer.Accept** should be called.
State	The state associated with the **IServer**. A server is typically returned as active unless an error occurred during the **Transport.Bind()** call. Table 6 describes possible state values.
PortNumber	The port number that this **IServer** is bound to and listening for incoming connections on.
UserSpecObject	A reference that can be set by the user of the **IServer**. This value can be set directly or via the bind options and is not modified by the transport. This information can be useful for coupling this **IServer** with other user created information, such as a list of associated **IChannel** objects.
Accept	Accepts an incoming connection. For more details, refer to Section 10.4.2.
BufferUsage	Returns the total number of used buffers for the **IServer**.
Close	Closes an **IServer**. Active Channels accepted from this **IServer** will not be closed.
Info	Gets information about the **IServer**. For more details, refer to section Section 10.14.4.
IOCtrl	Allows changing some I/O values programmatically for an **IServer**. For more details, refer to Section 10.14.6.

Table 14: IServer Methods

10.1.3.1 Server Connection Types

The following table summarizes possible Server connection types and the **BindOptions** parameter values that you can use to set them.

SERVER CONNECTION TYPE	BINDOPTIONS.CONNECTIONTYPE	SUBPROTOCOL LIST POSSIBILITIES
Unencrypted Socket	ConnectionType.SOCKET	Not used; RWF is implied.
Encrypted Socket	ConnectionType.ENCRYPTED	

Table 15: Server Settings for Socket Connection Types

10.1.4 Options Common to Both IChannel and IServer

IServer and IChannel share the following option:

- **TcpOpts**, which sets the **TcpNoDelay** option (for details, refer to Section 10.1.4.1).

10.1.4.1 TcpOpts Object

OPTION	DESCRIPTION
TcpNoDelay	If set to **true**, Nagle's Algorithm is disabled for all accepted connections. Nagle's Algorithm allows more efficient use of TCP by delaying and combining small packets to reduce repeated overhead of TCP headers. Disabling Nagle's Algorithm can lead to lower latency by removing this delay, but can add increased bandwidth use as a result of the additional TCP header used with each packet.

Table 16: TcpOpts Options

10.1.5 Transport Error Handling

Many Transport Package methods take a parameter for returning detailed error information. This **Error** object is set and populated only in the event of an error condition and should only be inspected when a specific failure code is returned from the method itself.

In several cases, positive return values are reserved or have special meaning, for example bytes remaining to write to the network. As a result, some negative return codes might be used to indicate success (e.g. **TransportReturnCode.READ_PING**). Any specific transport-related success or failure error handling is described along with the method that requires it.

Error methods are described in the following table.

METHOD	DESCRIPTION
Channel	A reference to the **IChannel** object on which the error occurred.
ErrorId	A Transport API-specific return code that specifies what error occurred. Refer to the following sections for specific error conditions that might arise.
SysError	Populated with the system error number associated with the failure. This information is only available when the failure occurs as a result of a system function, and will be populated by **0** otherwise.
Text	Detailed text describing the error. This can include-specific error information, underlying library-specific error information, or a combination of both.

Table 17: Error Methods

10.1.6 General Transport Return Codes

It is important that the application monitors return values from all Transport API methods that provide return-codes. Where specific error values are returned or special handling is required, the subsequent sections describe the possible return codes from Transport functionality. The following table lists general error codes. For Transport return codes specific to a particular method, refer to that method's section:

- **`IChannel.Init`** return codes: Section 10.5.4.
- **`IChannel.Read`** return codes: Section 10.6.2.
- **`IChannel.Write`** return codes: Section 10.9.5.
- **`IChannel.Flush`** return codes:Section 10.10.3.

TRANSPORT RETURN CODE	DESCRIPTION
TransportReturnCode.SUCCESS	Indicates successful completion of the operation.
TransportReturnCode.FAILURE	Indicates that initialization has failed and cannot progress. The **`IChannel.state`** should be **ChannelState.CLOSED**. See the **`Error`** content for more information.
TransportReturnCode.INIT_NOT_INITIALIZED	Indicates that the Transport has not been initialized. See the **`Error`** content for more details. For details on initializing, refer to Section 10.2.

Table 18: General Transport Return Codes

10.1.7 Application Lifecycle

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

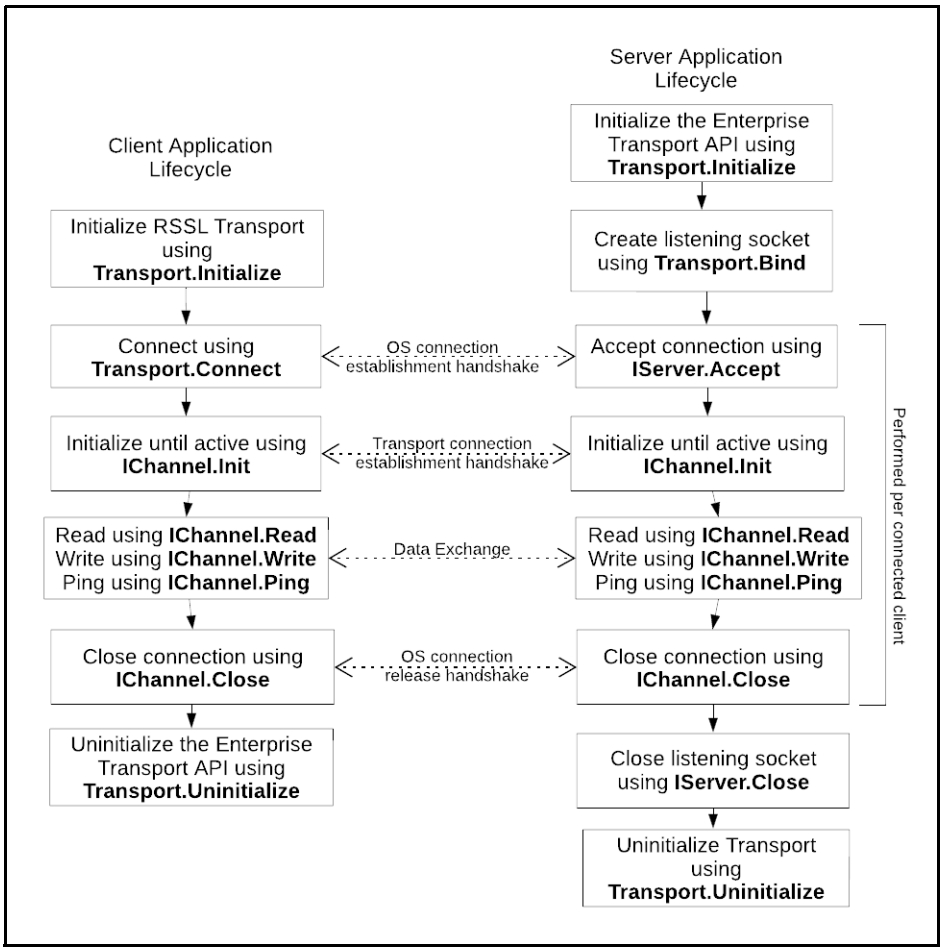


Figure 28. Transport Application Lifecycle

10.2 Initializing and Uninitializing the Transport

Every application using the transport, client or server, must first initialize it. This initialization process allows the Transport to pre-allocate internal memory associated with buffering and channel management.

Similarly, when an application has completed its usage of the Transport, it must uninitialize it. The uninitialization process

10.2.1 Initialization and Uninitialization Methods

The following table provides additional information about the Transport methods used for initializing and uninitializing.

METHOD	DESCRIPTION
Initialize	The first Transport function that an application should normally call. This creates and initializes internal memory. The **Initialize** method also allows the user to specify the locking model they want applied to the Transport. For more information, refer to Section 10.2.3.
Uninitialize	The last Transport method that an application should call. This uninitializes internal resources. These resources will eventually be garbage collected.

Table 19: Initialization and Uninitialization Methods

10.2.2 Initialization Reference Counting with Example

Both the **Initialize** and **Uninitialize** methods use reference counting. This allows only the first call to **Initialize** to perform any memory allocation and only the last necessary call to **Uninitialize** to undo the work of initialize. Only a single **Initialize** call need be made within an application, however this call must be the first Transport method call performed.

The following example demonstrates the use of **Initialize** and **Uninitialize**.

```
InitArgs initArgs = new InitArgs();
initArgs.GlobalLocking = true;
/* Starting Transport use, must call initialize first */
if (Transport.Initialize(initArgs, out Error error) != TransportReturnCode.SUCCESS)
{
   Console.WriteLine("Transport.Initialize failed. {0}", error.Text);
   /* End application */
   return 0;
}

/* Any transport use occurs here - see following sections for all other functionality */
/* All Transport use is complete, must uninitialize */
Transport.Uninitialize();

/* End application */
return 0;
```

Code Example 4: Transport Initialization and Uninitialization

10.2.3 Transport Locking Models

The Transport offers the choice of several locking models. These locking models are designed to offer maximum flexibility and allow the transport to be used in the manner that best fits the application's design. There are three types of locking that occur in the Transport. Global locking is used to protect any resources that are shared across connections or channels, such as connection pools. Read and Write Channel locking is used to protect any resources that are shared within a single connection or channel, such as a channel's buffer pool. Shared pool locking is used to protect a server's shared buffer pool, which is used to share one pool of buffers across multiple connections.

All three types of locking can be enabled or disabled, depending on the needs of the application, particularly:

- Global locking is controlled by **`InitArgs.GlobalLocking`**, with **`InitArgs`** as a parameter to the **`Transport.Initialize()`** method. After global locking is chosen, it cannot be changed without uninitializing and reinitializing the Transport. This behavior ensures that a locking change is not pushed onto pre-established connections.
- For client connections, IChannel locking is controlled on a per channel basis via **`ConnectOptions.ChannelReadLocking`** and **`ConnectOptions.ChannelWriteLocking`**, with **`ConnectOptions`** as a parameter to the **`Transport.Connect`** method. Once channel locking is chosen, it cannot be changed without closing and reconnecting the connection.
- For server connections, Channel locking is controlled on a per-channel basis via **`AcceptOptions.ChannelReadLocking`** and **`AcceptOptions.ChannelWriteLocking`**, with **`AcceptOptions`** as a parameter to the **`IServer.Accept`** method. Once channel locking is chosen, it cannot be changed without closing and re-accepting a connection.
- Shared pool locking is controlled on a per-server basis via **`BindOptions.SharedPoolLock`**, with **`BindOptions`** as a parameter to the **`Transport.Bind()`** method (for more information, refer to Section 10.4.1.1).

The following table describes the locking models and when to use each one.

LOCK MODEL	DESCRIPTION
None	The "no locking" model can be used for single-threaded applications to avoid any locking overhead as there is no risk of multiple thread access. It is additionally useful for multi-threaded applications that utilize the Transport from within a single thread, when the locking is managed by the application. An application can read a Channel from one thread and write to the same Channel using a different thread. This requires synchronization while creating and destroying connections so the use of Global lock is preferable.
Global, Channel (and Shared if using a Server)	Both global locking and channel locking will be enabled. This, in addition to enabling shared pool locking, will provide full thread safety. This setting allows for accessing the same channel from multiple threads. Note that writing messages from multiple threads can result in ordering issues and it is not recommended to write related messages across different threads. Reading across multiple threads can also introduce ordering issues associated with information received, which may or may not impact ordering of related messages.
Global	Global locking is enabled and channel locking is disabled. This allows for any globally shared resources (accessed through Transport methods) to be protected, but any channel related resources are not thread safe. This model allows for each channel to be handled by its own dedicated thread, but channel creation and destruction can occur across threads.
Channel	Global locking is disabled and Channel locking is enabled. This allows for accessing the same channel from multiple threads for reading and writing, but globally shared resources to be unprotected.
Shared	Global locking is disabled, Channel locking is disabled, and Shared locking is enabled. This allows for sharing of the shared pool buffers.

Table 20: Locking Types

10.3 Creating the Connection

The transport package allows for outbound connections to be established and managed. An outbound connection allows an application to connect to a listening socket network, often to some type of Provider running on a well known port number.

10.3.1 Network Topologies

The Enterprise Transport API supports one type of network topologies:

- **`unified`**: A **`unified`** network topology is one where the **`IChannel`** uses the same connection information (**`address:port`**) to send and receive all content.

On TCP-based networks, the Enterprise Transport API supports only a **`unified`** topology (**`ConnectionType.SOCKET`** and **`ConnectionType.ENCRYPTED`**).

For configuration information on network topologies, refer to Table 23.

10.3.1.1 TCP-based Networks

If an application needs to communicate with multiple devices using a **`ConnectionType.SOCKET`** or **`ConnectionType.ENCRYPTED`** connection type, a unique (point-to-point) connection is required for each device. Any content that needs to go to all devices must be written (or “fanned out”) on all connections, which is the application’s responsibility. The following diagram illustrates this scenario:

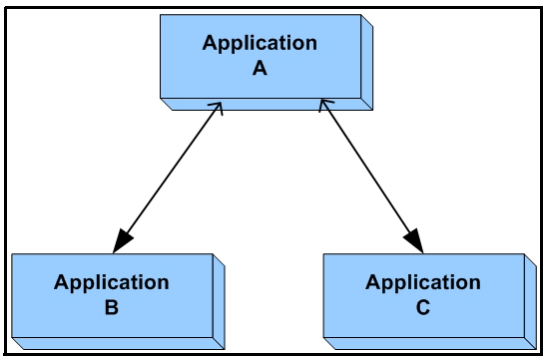


Figure 29. Unified TCP Network

In Figure 29, Application A has a unique, point-to-point connection with each of the applications B and C. If Application A wants to send the same content to both applications B and C, Application A must send the same content over each connection. In this scenario, if content is sent over only one connection, only the application on the corresponding end of that connection receives the content.

For TCP connections, consumer and non-interactive provider applications connect as shown in the following diagram. The arrows used in the figure depict the directions in which connections are established. Consumers typically connect to a well known port number associated with some kind of Interactive Provider (e.g., the LSEG Real-Time Advanced Distribution Server or LSEG Real-Time), while non-interactive providers typically connect to a well known port on the LSEG Real-Time Advanced Distribution Hub.

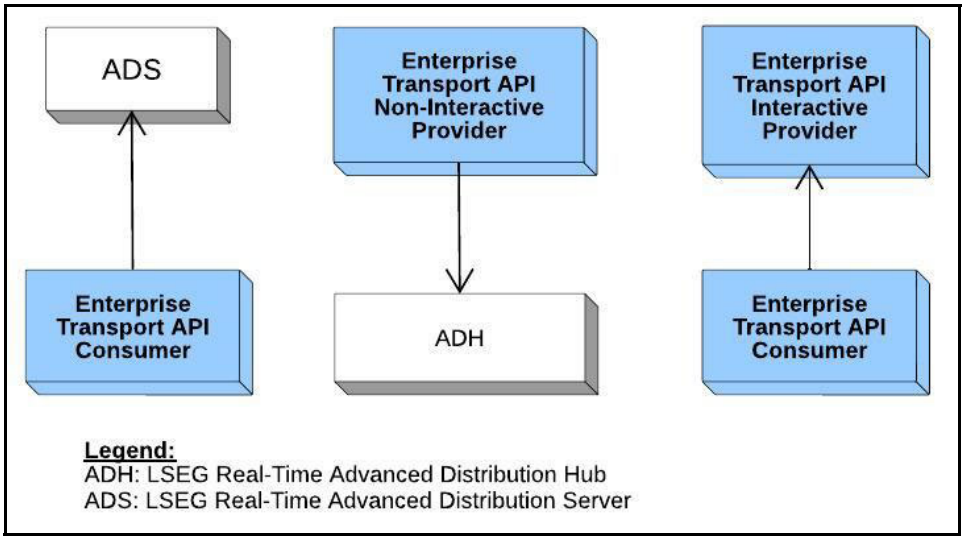


Figure 30. TCP Connection Creation

10.3.2 Creating the Outbound Connection: Transport.Connect Method

An application can create an outbound connection by using the **`Transport.Connect()`** method.

METHOD	DESCRIPTION
Transport.Connect()	Establishes an outbound connection, which can leverage standard sockets. Returns an **`IChannel`** that represents the connection to the user. In the event of an error, null reference is returned and additional information can be found in the **`Error`** structure. Connection options are passed in via an **`ConnectOptions`** object described in Table 22. Once a connection is established and transitions to the **`ChannelState.ACTIVE`** state, this **`IChannel`** can be used for other transport operations. For more information about channel initialization, refer to Section 10.5.

Table 21: `Transport.Connect()` Method

10.3.2.1 ConnectOptions Methods

METHOD	DESCRIPTION
Blocking	If set to **true**, blocking I/O will be used for this **IChannel**. When I/O is used in a blocking manner on an **IChannel**, any reading or writing will complete before control is returned to the application. In addition, the **Transport.Connect()** method will complete any initialization on the **IChannel** prior to returning it. Blocking I/O prevents the application from performing any operations until the I/O operation is completed. Blocking I/O is typically not recommended. An application can leverage an I/O notification mechanism to allow efficient reading and writing, while using other cycles to perform other necessary work in the application. An I/O notification mechanism enables the application to read when data is available, and write when output space is available.
ChannelReadLocking	Accessor method used to set or check if the connection will use locking on reading.
ChannelWriteLocking	Accessor method used to set or check if the connection will use locking on writing.
Clear	Clears this object, so that it can be reused.
ComponentVersion	An optional, user-defined component version string appended behind the standard Enterprise Transport API component version information. If the combined component version length exceeds the maximum supported by the Enterprise Transport API, the user-defined information will be truncated.
CompressionType	The type of compression the client would like performed for this connection. Compression is negotiated between the client and server and may not be performed if only the client has it enabled. For more information about supported compression types and compression negotiation, refer to Section 10.4.3.
ConnectTimeout	Gets or sets the connect timeout when establishing a connect to a remote host in milliseconds. The default value is 10000 miliseconds.
ConnectionType	Specifies the type of connection to establish. Creation of TCP-based socket and encrypted socket connection types are available across all supported platforms. Connection Types are described in more detail in Section 10.1.2.2.
GuaranteedOutputBuffers	A guaranteed number of buffers made available for this **IChannel** to use while writing data. Guaranteed output buffers are allocated at initialization time. For more information, refer to Section 10.8.
MajorVersion	The major version of the protocol that the client intends to exchange over the connection. This value is negotiated with the server at connection time. The outcome of the negotiation is provided via the **MajorVersion** information on the **IChannel**. Typically, a major version increase is associated with the introduction of incompatible change. The transport layer is data-neutral and allows the flow of any type of content. **MajorVersion** is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
MinorVersion	The minor version of the protocol that the client intends to exchange over the connection. This value is negotiated with the server at connection time. The outcome of the negotiation is provided via the **MinorVersion** information on the **IChannel**. Typically, a minor version increase is associated with a fully backward compatible change or extension. The transport layer is data-neutral and allows the flow of any type of content. **MinorVersion** is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
NumInputBuffers	The number of sequential input buffers to allocate for reading data into. This controls the maximum number of bytes that can be handled with a single network read operation. Input buffers are allocated at initialization time.

Table 22: ConnectOptions Methods

METHOD	DESCRIPTION
PingTimeout	The clients desired ping timeout value. This may change through the negotiation process between the client and the server. After the connection becomes active, the actual negotiated value becomes available through the **PingTimeout** value on the **IChannel**. When determining the desired ping timeout, the typically used rule of thumb is to send a heartbeat every **PingTimeout**/3 seconds. For more information, refer to Section 10.12.
ProtocolType	The protocol type that the client intends to exchange over the connection. If the **ProtocolType** indicated by a server does not match the **ProtocolType** that a client specifies, the connection will be rejected. When an **IChannel** becomes active for a client or server, this information becomes available via the **ProtocolType** on the **IChannel**. The transport layer is data neutral and does not change nor depend on any information in content being distributed. This information is provided to help client and server applications manage the information they are communicating. For more details, refer to Section 9.5.1.
EncryptionOpts	**EncryptionOptions** is an object that configures an encrypted tunnel connection. For further details, refer to Section 10.3.2.3.
ProxyOptions	**ProxyOptions** is an object that configures a proxy for use with connections. **ProxyOptions** uses the variables: **ProxyHostName** and **ProxyPort**. For further details, refer to Section 10.3.2.4.
ReceiveTimeout	Gets or sets the receive time out for reading data from **Socket**.
SendTimeout	Gets or sets the send time out for writing data to **Socket**.
SysRecvBufSize	Accessor method used to set or get the system's receive buffer size used for this connection. Not setting or setting to **0** indicates to use the default size of 64 KB. This can also be set or changed via **IChannel.IOCtl**.
SysSendBufSize	Accessor method, used to set or get the system's send buffer size used for this connection. No setting or setting to **0** indicates to use the default size of 64KB.
TcpOpts	**TcpOpts** is an object which contains the **TcpNoDelay** option for use with TCP-based connection types. You can use **TcpOpts** for **ConnectionType.SOCKET** and **ConnectionType.ENCRYPTED** connections. For further information about the **TcpOpts** object, refer to Section 10.1.4.1.
UnifiedNetworkInfo	**UnifiedNetworkInfo** object representing connection parameters used when sending and receiving on same network. This is typically used with **ConnectionType.SOCKET** and **ConnectionType.ENCRYPTED** networks. **UnifiedNetworkInfo** is described in more detail in Section 10.3.2.2.
UserSpecObject	A reference that can be set by the application. This value is not modified by the transport, but will be preserved and stored in the **UserSpecPtr** of the **IChannel** returned from **Transport.Connect()**. This information can be useful for coupling this **IChannel** with other user created information, such as a watch list associated with this connection.

Table 22: ConnectOptions Methods (Continued)

10.3.2.2 UnifiedNetworkInfo Options

METHOD	DESCRIPTION
Address	Configures the address or hostname to use in a unified network configuration. All content will be sent and received on this **`Address:ServiceName`** pair.
InterfaceName	A character representation of an IP address or hostname associated with the local network interface to use for sending and receiving content. This value is intended for use in systems which have multiple network interface cards, and if unspecified, the default network interface is used.
ServiceName	Configures the numeric port number or service name (as defined in the **etc/services** file) to use in a unified network configuration. All content will be sent and received on this **`Address:ServiceName`** pair.

Table 23: `UnifiedNetworkInfo` Method Options

10.3.2.3 EncryptionOptions Method Options

For further details on ETA encryption and using these options, refer to Section 10.15.

METHOD	DESCRIPTION
AuthenticationTimeout	Gets or sets the timeout in millisecond for server to complete the authentication with the client. Defaults to 10000 milliseconds.
EncryptedProtocol	**Required**. Specifies the transport protocol to use in the encrypted connection. Available values include: • **ConnectionType.SOCKET**: SSL encrypted TCP-based transport.
EncryptionProtocolFlags	**Required**. Sets the security protocol for encrypted connections. Available flags include: • **ENC_NONE == 0**: Allows the operating system to choose the best protocol to use, and to block protocols that are not secure. Unless your app has a specific reason not to, you should use this field. • **ENC_TLSV1_2 == 0x04**: Sets ETA to use TLSv1.2 protocol. • **ENC_TLSV1_3 = 0x08**: Sets ETA to use TLSv1.3 protocol.
TlsCipherSuites	Specifies a collection of cipher suites allowed for TLS negotiation. The operating system default is used if not set. Use extreme caution when changing this setting. **WARNING!** This option supports only on a Linux system with OpenSSL 1.1.1 or higher or a macOS.

Table 24: `EncryptionOptions` Method

10.3.2.4 ProxyOptions Method Options

METHOD	DESCRIPTION
ProxyHostName	**Required**. Sets the hostname of a proxy for use with a socket or encrypted-socket connection. For further details on ETA tunneling behavior, refer to Section 10.15.
ProxyPassword	Optional. Specifies the password to use for authenticated proxy connections.
ProxyPort	**Required**. Sets the port on a proxy for use with a socket or encrypted-socket connection. For further details on ETA tunneling behavior, refer to Section 10.15.
ProxyUserName	Optional. Specifies the user name to use for authenticated proxy connections.

Table 25: `ProxyOptions` Method

10.3.3 Transport.Connect Outbound Connection Creation Example

The following example demonstrates basic **Transport.Connect()** use in a non-blocking manner. The application first populates the **ConnectOptions** and then attempts to connect. If the connection succeeds, the application then registers the **IChannel.Socket** with the I/O notification mechanism and continues with connection initialization (as described in Section 10.5).

```
IChannel channel;

ConnectOptions cOpts = new ConnectOptions();
/* populate connect options, then pass to Transport.Connect method - Transport should already be
   initialized */
cOpts.ConnectionType = ConnectionType.SOCKET; /* use standard socket connection */
cOpts.UnifiedNetworkInfo.Address = "localhost"; /* connect to server running on same machine */
cOpts.UnifiedNetworkInfo.ServiceName = "14002"; /* server is running on port number 14002 */
cOpts.PingTimeout = 30; /* clients desired ping timeout is 30 seconds, pings should be sent every 10 */
cOpts.Blocking = false; /* perform non-blocking I/O */
cOpts.CompressionType = CompressionType.NONE; /* client does not desire compression for this
   connection */

/* populate version and protocol with RWF information */
cOpts.PotocolType = Codec.ProtocolType();
cOpts.MajorVersion = Codec.MajorVersion();
cOpts.MinorVersion = Codec.MinorVersion();
if ((channel = Transport.Connect(cOpts, out Error error)) == null)
{
   Console.WriteLine("Transport.Connect() failed: {0}({1})\n", error.ErrorId, error.Text);
   /* End application, uninitialize to clean up first */
   Transport.Uninitialize();
   return;
}

List<Socket> readList = new List<Socket>();
List<Socket> writeList = new List<Socket>();
/* Connection was successful, add Socket to I/O notification mechanism and initialize connection */
try
{
      /* register for read and write select */
      readList.Add(channel.Socket);
      writeList.Add(channel.Socket);

      Socket.Select(readList, writeList, null, 0);

      if(readList.Count > 0)
      {
      /* Channel is ready to read */
      }

      if(writeList.Count > 0)
      {
      /* Channel is ready to write *
      }
```

```
}
Catch (Exception e)
{
/* Socket.Select() can throw numerous exceptions. */
/* For this example catch all as Exception. */
// handle exception and abort.
return;
}
/* Continue on with connection initialization process, refer to Section xx for more details. */
```

Code Example 5: Creating a Connection using Transport.Connect

10.4 Server Creation and Accepting Connections

10.4.1 Creating a Listening Socket

The Transport Package allows you to establish and manage listening sockets, typically associated with a server. Listening sockets can be leveraged to create an application that accepts connections created through the use of **`Transport.Connect()`**. Listening sockets are used mainly by OMM interactive provider applications and are typically established on a well-known port number (known by other connecting applications).

Figure 31. Transport API Server Creation

An application can create a listening socket connection by using the **`Transport.Bind()`** method, described in the following table.

METHOD	DESCRIPTION
Transport.Bind()	Establishes a listening socket connection, which supports connections from standard socket **`Transport.Connect()`** users. Returns an **`IServer`** that represents the listening socket connection to the user. In the event of an error, null is returned and additional information can be found in the **`Error`** object. Options are passed in via an **`BindOptions`** object described in . Once a listening socket is established, this **`IServer`** can begin accepting connections. For more information, refer to Section 10.4.2.

Table 26: `Transport.Bind()` Method

10.4.1.1 BindOptions Methods

METHOD	DESCRIPTION
ChannelsBlocking	If set to **true**, blocking I/O will be used for all connected **IChannel**s. When I/O is used in a blocking manner on an **IChannel**, any reading or writing will complete before control is returned to the application. Blocking I/O prevents the application from performing any operations until the I/O operation is completed. Blocking I/O is typically not recommended. An application can leverage an I/O notification mechanism to allow efficient reading and writing, while using other cycles to perform other necessary work in the application. An I/O notification mechanism enables the application to read when data is available, and write when output space is available.
ClientToServerPings	If set to **true**, heartbeat messages are required to flow from the client to the server. If set to **false**, the client is not required to send heartbeats. LSEG Real-Time Distribution System and other LSEG components typically require this to be set to **true**.
ComponentVersion	An optional, user-defined component version string appended behind the standard ETA component version information. If the combined component version length exceeds the maximum supported by the Enterprise Transport API, the user-defined information will be truncated.
CompressionType	The type of compression the server wants to apply for this connection. Compression is negotiated between the client and server and may not be performed if only the server has this enabled. The server can force compression, regardless of client settings, by using the **ForceCompression** option. For more information about supported compression types and compression negotiation, refer to Section 10.4.3.
CompressionLevel	Sets the level of compression to apply. Allowable values are **0** to **9**. • A **CompressionLevel** of **1** results in the fastest compression. • A **CompressionLevel** of **9** results in the best compression. • A **CompressionLevel** of **6** is a compromise between speed and compression. • A **CompressionLevel** of **0** will copy the data with no compression applied. For more information on supported compression levels, refer to Section 10.4.3.
ConnectionType	Specifies the type of connection to establish. **ConnectionType**s are described in more detail in Section 10.1.2.2. ETA supports TCP-based connection types. If **ConnectionType** is set to **ConnectionType.ENCRYPTED**, all incoming TCP connections are encrypted. **NOTE:** The server supports standard TCP connection.
BindEncryptionOpts	If the **ConnectionType** is set to **ConnectionType.ENCRYPTED**, **BindEncryptionOpts** sets the TLS encryption options. For information, refer to Section 10.4.1.2.
ForceCompression	If set to **true**, this forcibly enables compression, regardless of client preference. When enabled, compression will use the **CompressionType** and **CompressionLevel** specified by the server. If set to **false**, compression is negotiated between the client and server. For more information about supported compression types and compression negotiation, refer to Section 10.4.3.
GuaranteedOutputBuffers	A guaranteed number of buffers made available for each **IChannel** to use while writing data. Each buffer is created to contain **MaxFragmentSize** bytes. Guaranteed output buffers are allocated at initialization time. For more information, refer to Section 10.8.
InterfaceName	A character representation of an IP address or hostname for the local network interface to which to bind. The Transport will establish connections on the specified interface. This value is intended for use in systems which have multiple network interface cards. If not populated, a connection can be accepted on all interfaces (**INADDR_LOOPBACK** is used). If the loopback address (127.0.0.1) is specified, connections can be accepted only when instantiating from the local machine (**INADDR_LOOPBACK** is used).

Table 27: `BindOptions` Methods

METHOD	DESCRIPTION
MajorVersion	Specifies the major version of the protocol supported by the server. The actual major version used is negotiated with the client at connection time. The outcome of the negotiation is provided via **MajorVersion** on the **IChannel**. Typically, the major version increases with the introduction of an significant (i.e., incompatible) change. The transport layer is data-neutral and allows the flow of any type of content. **MajorVersion** is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
MaxFragmentSize	The maximum size buffer that will be written to the network. If a larger buffer is required, the Transport will internally fragment the larger buffer into smaller **MaxFragmentSize** buffers. This is different from application level message fragmentation done via the Message Package (as discussed in). Any guaranteed, shared, or input buffers created will use this size. This value is passed to all connected client applications and enforces a common message size between components. For more information about Transport buffer fragmentation, refer to Section 10.9.
MaxOutputBuffers	The maximum number of output buffers allowed for use by each **IChannel**. (**MaxOutputBuffers** - **GuaranteedOutputBuffers**) is equal to the number of shared pool buffers that each **IChannel** is allowed to use. Shared pool buffers are only used if all **GuaranteedOutputBuffers** are unavailable. If equal to the **GuaranteedOutputBuffers** value, no shared pool buffers are available.
MinorVersion	The minor version of the protocol supported by the server. The actual minor version used is negotiated with the client at connection time. The outcome of the negotiation is provided via **MinorVersion** on the **IChannel**. Typically, the minor version increases with the introduction of a fully backward-compatible change or extension. The transport layer is data-neutral and allows the flow of any type of content. **MinorVersion** is provided to help client and server applications manage the information they communicate. For more details, refer toSection 9.5.1.
MinPingTimeout	The server's lowest allowable ping timeout value. This is the lowest possible value allowed in the negotiation between client and servers **PingTimeout** values. After the connection becomes active, the actual negotiated value becomes available through the **PingTimeout** value on the **IChannel**. When determining the desired ping timeout, the rule of thumb is to send a heartbeat every **PingTimeout**/3 seconds. For more information, refer to Section 10.12.
NumInputBuffers	The number of sequential input buffers used by each **IChannel** for data reading. This controls the maximum number of bytes that can be handled with a single network read operation on each channel. Each input buffer will be created to contain **MaxFragmentSize** bytes. Input buffers are allocated at initialization time.
PingTimeout	After the connection becomes active, the actual negotiated value becomes available through the **PingTimeout** value on the **IChannel**. When determining the desired ping timeout, the rule of thumb is to send a heartbeat every **PingTimeout**/3 seconds. For more information, refer to Section 10.12.
ProtocolType	Sets the protocol type that the server uses on its connections. The server rejects connections from clients that do not use the specified **ProtocolType**. Server must bind using RSSL_RWF_PROTOCOL_TYPE. When an **IChannel** becomes active for a client or server, this information becomes available via the **ProtocolType** on the **IChannel**. The transport layer is data-neutral and allows the flow of any type of content. **ProtocolType** is provided to help client and server applications manage the information they communicate. For more details, refer toSection 9.5.1.

Table 27: `BindOptions` Methods (Continued)

METHOD	DESCRIPTION
ServerBlocking	If set to **true**, blocking I/O will be used for this **`IServer`**. When I/O is used in a blocking manner on an **`IServer`**, the **`IServer.Accept`** method will complete any initialization on the **`IChannel`** prior to returning it. Blocking I/O prevents the application from performing any operations until the I/O operation is completed. Blocking I/O is typically not recommended. An application can leverage an I/O notification mechanism to allow efficient use, while using other cycles to perform other necessary work in the application.
ServerToClientPings	If set to **true**, heartbeat messages are required to flow from the server to the client. If set to **false**, the server is not required to send heartbeats. LSEG Real-Time Distribution System and other LSEG components typically require this to be set to **true**.
ServiceName	A character representation of a numeric port number or service name (as defined in the **etc/services** file) on which to bind and open a listening socket.
SharedPoolSize	The maximum number of buffers to make available as part of the shared buffer pool. The shared buffer pool can be drawn upon by any connected **`IChannel`**, where each channel is allowed to use up to (**`MaxOutputBuffers`** - **`GuaranteedOutputBuffers`**) number of buffers. Each shared pool buffer will be created to contain **`MaxFragmentSize`** bytes. If set to **0**, a default of 1,048,567 shared pool buffers will be allowed. The shared pool is not fully allocated at bind time. As needed, shared pool buffers are added and reused until the server is shut down. For more information, refer to Section 10.8.
	NOTE: It is considered an invalid configuration to allow more shared pool buffers (**`MaxOutputBuffers`** - **`GuaranteedOutputBuffers`**) than the **`SharedPoolSize`**. If this happens, an error is returned from **`Transport.Bind()`**.
SharedPoolLock	If set to **true**, the shared buffer pool will have its own locking performed. This setting is independent of any other locking mode options. Enabling a shared pool lock allows shared pool use to remain thread safe while still disabling channel locking. For more information, refer to Section 10.2.3.
SysRecvBufSize	Sets the system's receive buffer size for this connection. A missing value, or a setting of 0 sets the buffer to the default size of 64K. Setting **`SysSendBufSize`** is done via **`AcceptOptions`** (for details, refer to Table 30). This can also be set or changed via **`IChannel.IOCtl`** for values less than or equal to 64K. For values larger than 64K, you must use this method to set **`SysRecvBufSize`** prior to the bind system call.
TcpOpts	Specifies the **`TcpOpts`** class, which contains the **`TcpNoDelay`** option for use with TCP-based connection types (i.e., **ConnectionType.SOCKET** and **ConnectionType.ENCRYPTED** connection types). For further information about **`TcpNoDelay`**, refer to Section 10.3.2.4.
UserSpecObject	A reference that can be set by the application. This value is not modified by the transport, but is preserved and stored in the **`UserSpecObject`** of the **`IServer`** returned from **`Transport.Bind()`** if a **`UserSpecObject`** was not specified in the **`AcceptOptions`**. This information can be useful for coupling this **`IServer`** with other user-created information, such as a list of connected **`IChannel`**s.

Table 27: `BindOptions` Methods (Continued)

10.4.1.2 BindEncryptionOptions Method Options

METHOD	DESCRIPTION
AuthenticationTimeout	Gets or sets the timeout in millisecond for server to complete the authentication with the client. Defaults to 10000 milliseconds.
EncryptionProtocolFlags	Specifies the TLS version to be accepted by the server. By default, Enterprise Transport API accepts both TLS 1.3 and TLS 1.2. However, the OS on which the server runs must support a minimum version of OpenSSL 1.1.1 for TLS 1.3 to be supported by API.
ServerCertificate	**Required**. Specifies the ***filename*** of the server certificate. For details on server-side certificate usage, refer to Section 10.15.1.
ServerPrivateKey	**Required**. Specifies the filename of the server's private key. For more details about private key usage, refer to Section 10.15.1.
TlsCipherSuites	Specifies a collection of cipher suites allowed for TLS negotiation. The operating system default is used if not set. Use extreme caution when changing this setting.
	WARNING! This option supports only on a Linux system with OpenSSL 1.1.1 or higher or a macOS.

Table 28: BindEncryptionOptions Methods

10.4.1.3 Transport.Bind Listening Socket Connection Creation Example

The following example demonstrates basic **Bind** use in a non-blocking manner. The application first populates the **BindOptions** and then attempts to create a listening socket. If the bind succeeds, the application then registers the **IServer.Socket** with the I/O notification mechanism and waits to be alerted of incoming connection attempts. For more details on accepting or rejecting incoming connection attempts, refer to Section 10.4.2.

```
IServer server = null;
BindOptions bOpts = new BindOptions();

/* populate bind options, then pass to bind method - ETA should already be initialized */
bOpts.ServiceName = "14002"; /* server is running on port number 14002 */
bOpts.PingTimeout = 45; /* servers desired ping timeout is 45 seconds, pings should be sent every 15 */
bOpts.MinPingTimeout = 30; /* min acceptable ping timeout is 30 seconds, pings should be sent every 10 */
/* set up buffering, configure for shared and guaranteed pools */
bOpts.GuaranteedOutputBuffers = 1000;
bOpts.MaxOutputBuffers = 2000;
bOpts.SharedPoolSize = 50000;
bOpts.SharedPoolLock = true;
bOpts.ServerBlocking = false; /* perform non-blocking I/O */
bOpts.ChannelsBlocking = false; /* perform non-blocking I/O */
bOpts.CompressionType = CompressionType.NONE; /* server does not desire compression for this
   connection */
/* populate version and protocol with RWF information or protocol specific info */
bOpts.protocolType = Codec.ProtocolType();
bOpts.majorVersion = Codec.MajorVersion();
bOpts.minorVersion = Codec.MinorVersion();
if ((server = Transport.Bind(bOpts, out Error error)) is null)
{
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Bind. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
   /* End application, uninitialize to clean up first */
   Transport.Uninitialize();
   return null;
}
/* Bind was successful, add IServer.Socket to check for readability to accept incoming connections */
List<Socket> serverWaitList = new() { server.Socket };

try
{
   Socket.Select(serverWaitList, null, null, (int)(timeOut.TotalMilliseconds * 1000));

   /* Use accept for incoming connections, read and write data to established connections, etc */
}
catch (Exception exp)
{
   Console.WriteLine($" Socket.Select() Exception {exp.Message}");
}
```

Code Example 6: Creating a Listening Socket Using `Transport.Bind()`

10.4.2 Accepting Connection Requests

After establishing a listening socket, the **IServer.Socket** can be registered with an I/O notification mechanism. An alert from the I/O notification mechanism on the server's **Socket** indicates that a connection request has been detected. An application can begin the process of accepting or rejecting the connection by using the **IServer.Accept** method.

METHOD	DESCRIPTION
IServer.Accept	Uses the **IServer** that represents the listening socket connection and begins the process of accepting the incoming connection request. Returns an **IChannel** that represents the client connection. In the event of an error, NULL is returned and additional information can be found in the **Error** object. The **IServer.Accept** method can also begin the rejection process for a connection through the use of the **AcceptOptions** object as described in Section 10.4.2.1. Once a connection is established and transitions to **ChannelState.ACTIVE**, this **IChannel** can be used for other transport operations. For more information about channel initialization, refer to Section 10.5.

Table 29: IServer.Accept Method

10.4.2.1 AcceptOptions Method

METHOD	DESCRIPTION
ChannelReadLocking	Sets or checks whether the connection will use locking on reading.
ChannelWriteLocking	Sets or checks whether the connection will use locking on writing.
Clear	Clears the object for reuse.
NakMount	Indicates that the server wants to reject the incoming connection. This may be due to some kind of connection limit being reached. For non-blocking connections to successfully complete rejection, the initialization process must still be completed. For more information about channel initialization, refer to Section 10.5.
ReceiveTimeout	Gets or sets the receive time out for reading data from Socket.
SendTimeout	Gets or sets the send time out for writing data to Socket.
SysSendBufSize	Sets or checks the system's send buffer size used for this connection. No setting or a setting of 0 indicates to use the default (64K). **SysRecvBufSize** is set via the **BindOptions** (for details, refer to Section 10.4.1.1.
UserSpecObject	A reference that can be set by the application. This value is not modified by the transport, but will be preserved and stored in the **UserSpecObject** of the **IChannel** returned from **IServer.Accept**. If this value is not set, the **IChannel.UserSpecObject** will be set to the **UserSpecObject** associated with the **IServer** that is accepting this connection.

Table 30: AcceptOptions Methods

10.4.2.2 IServer.Accept Accepting Connection Example

The following example demonstrates basic **IServer.Accept** use. The application first populates the **AcceptOptions** and then attempts to accept the incoming connection request. If the accept succeeds, the application then registers the new **IChannel.Socket** with the I/O notification mechanism and continues with connection initialization, described in Section 10.5.

```
/* Accept is typically called when servers socketId indicates activity */
IChannel channel = null;
AcceptOptions aOpts = new AcceptOptions();
/* populate accept options, then pass to Accept method - ETA should already be initialized */
aOpts.nakMount = false; /* allow the connection */
if ((chnl = srvr.Accept(aOpts, out Error error)) is null)
{
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Accept. Error Text: {2}",
         error.ErrorId, error.SysError, error.Text);
   /* End application, uninitialize to clean up first */
   Transport.Uninitialize();
   return null;
}
/* Accept was successful, register for read and write I/O notification for the channel's socket */

List<Socket> readList = new() { chnl.Socket };
List<Socket> writeList = new() { chnl.Socket };

try
{
   Socket.Select(readList, writeList, null, 0);

   if(readList.Count > 0)
   {
      /* Channel is ready to read */

      /* Continue the connection initialization process, for details, refer to xxx */
   }

   if(writeList.Count > 0)
   {
      /* Channel is ready to write */
   }
}
Catch (Exception e)
{
   /* Socket.Select() can throw numerous exceptions. */
   /* For this example catch all as Exception. */
   // handle exception and abort.
   return;
}
```

Code Example 7: Accepting Connection Attempts using `IServer.Accept`

10.4.3 Compression Support

As mentioned, the Transport supports the use of data compression. The client and server negotiate compression behavior during the connection establishment process, with the server determining supported compression methods by referencing the **`BindOptions.CompressionType`** parameter (refer to the ENUMs specified in Section 10.4.3.1).

Additionally:

- You can configure the server to support multiple compression types by including the appropriate bitmasks.
- When using zlib, you can configure the the quality of compression by setting the **`BindOptions.CompressionLevel`** parameter.

A client requests compression by setting the **`ConnectOptions.CompressionType`** parameter to one of the values specified in Section 10.4.3.1. If the client's configured compression type matches one of the types specified by the server, that compression type is used for the connection. After establishing a connection, the server or client can verify at any time the type of compression in use on a channel by calling **`IChannel.Info`** (refer to Section 10.14).

You can set the server to force compression for its connections by enabling the **`BindOptions.ForceCompression`** parameter, in which case the server's **`CompressionType`** is used regardless of the client's configuration.

NOTE: If you set the server to force compression, use only one compression type in **`BindOptions.compressionType`**.

10.4.3.1 Compression Types

The Transport API supports the following compression options:

COMPRESSION TYPE	COMPRESSION LEVEL SUPPORTED	DEFAULT COMPRESSION THRESHOLD	DESCRIPTION
NONE	n/a	n/a	No compression.
ZLIB	Yes	30 bytes	Use zlib compression. Zlib, an open source utility, employs a variation of the LZ77 algorithm to compress and decompress data.
LZ4	No	300 bytes	Use LZ4 compression. LZ4, an open source utility, employs a variation of the LZ77 algorithm to compress and decompress data. **NOTE:** Though LZ4 compression consumes less CPU than Zlib, LZ4 does not achieve the same reduction in size.

Table 31: CompressionType Values

10.4.3.2 Compression Level

The server's specified **`CompressionLevel`** determines the quality of the compression, where:

- Lower values favor faster compression
- Higher values compress data into smaller sizes

Currently only zlib supports the use of compression levels.

10.4.3.3 Compression Threshold

Different compression types have different behaviors and compression efficiency can vary depending on buffer size. Because small buffer sizes might not compress well, the Transport API uses a compression threshold such that all buffers exceeding the threshold size are compressed. Default compression thresholds are specified in Section 10.4.3.1. You can change this threshold via the **IChannel.IOCtl** function (refer to Section 10.14).

If a message is larger than the compression threshold, you can prevent its compression through the use of the **IChannel.Write** method with the **WriteFlags.DO_NOT_COMPRESS** flag. For more information, refer to Section 10.9.

10.5 Channel Initialization

After an **IChannel** is returned from the client's **Transport.Connect()** or server's **IServer.Accept** call, the channel may need to continue the initialization process using the **IChannel.Init** function.

NOTE: For both client and server channels, to complete the channel initialization process, more than one call to **IChannel.Init** might be required.

Additional initialization is required as long as the **IChannel.State** is **ChannelState.INITIALIZING**.

- If using a non-blocking I/O, this is the typical state from which an **IChannel** starts and multiple initialization calls might be needed to transition to active.
- If using a blocking I/O, when successful, **Transport.Connect()** and **IServer.Accept()** return a completely initialized channel in an active state.

Internally, the initialization process involves several actions. The initialization includes any necessary connection handshake exchanges, including any HTTP or HTTPS negotiation. Compression, ping timeout, and versioning related negotiations also take place during the initialization process. This process involves exchanging several messages across the connection, and once all message exchanges have completed the **IChannel.State** will transition.

- If the connection is accepted (i.e., all negotiations were successful), the **IChannel.State** will become **ChannelState.ACTIVE**.
- If the connection is rejected (i.e., due to either failed negotiation or an **IServer** rejection of the connection by setting **NakMount** to **true**), the **IChannel.State** will become **ChannelState.CLOSED**, and the application should close the channel to clean up any associated resources.

10.5.1 IChannel.Init Method

METHOD	DESCRIPTION
IChannel.Init	Continues initialization of an **IChannel**. This channel could originate from **Transport.Connect()** or **IServer.Accept**. This method exchanges various messages to perform necessary negotiations and handshakes to complete channel initialization. If using blocking I/O, this method is typically not used because **Transport.Connect()** and **IServer.Accept** return active channels. Requires the use of the **InProgInfo** object, refer to Section 10.5.2. The **IChannel** can be used for all additional transport functionality (e.g. reading, writing) after the **state** transitions to **ChannelState.ACTIVE**. If a connection is rejected or initialization fails, the state transitions to **ChannelState.CLOSED**, and the application should close the channel to clean up any associated resources. The return values are described in Section 10.5.4.

Table 32: IChannel.Init Method

10.5.2 InProgInfo Object

Use the **InProgInfo** object with the **IChannel.Init** method to initialize a channel.

In certain circumstances, the initialization process might need to create new or additional underlying connections. If this occurs, the application must unregister the previous **Socket** and register the new **Socket** with the I/O notification mechanism in use with associated information being conveyed by **InProgInfo** and **InProgFlags**.

METHOD	DESCRIPTION
Flags	Combination of bit values to indicate special behaviors and presence of optional **InProgInfo** content. **flags** uses the following enumeration values: • **InProgFlags.NONE**: Indicates that channel initialization is still in progress and subsequent calls to **IChannel.Init** are needed for completion. The call did not change the **Socket**. • **InProgFlags.SCKT_CHNL_CHANGE**: Indicates that the call changed the **Socket**. The previous **Socket** is now stored in **InProgInfo.OldSocket** so it can be unregistered with the I/O notification mechanism. The new **Socket** is stored in **InProgInfo.NewSocket** so it can be registered with the I/O notification mechanism. However, channel initialization is still in progress and subsequent calls to **IChannel.Init** are needed to complete it.
OldSocket	Populated if flags indicate that **IChannel.Init** needs to perform a socket change. If this occurs, the **OldSocket** contains the **Socket** associated with the previous connection so the application can unregister this with the I/O notification mechanism.
NewSocket	Populated if flags indicate that **IChannel.Init** needs to perform a socket change. If this occurs, the **NewSocket** contains the **Socket** associated with the new connection so the application can register this with the I/O notification mechanism.

Table 33: InProgInfo Methods

10.5.3 Calling IChannel.Init

Typically, calls to **IChannel.Init** are driven by I/O on the connection, however this can also be accomplished by using a timer to periodically call the method or looping on a call until the channel transitions to active or a failure occurs. Other than any overhead associated with the method call, there is no harm in calling **IChannel.Init** more frequently than required. If work is not required, the method returns, indicating that the connection is still in progress.

If using I/O, a client application should register the **IChannel.Socket** with the read, write, and exception file descriptor sets. When the write descriptor alerts the user that the **Socket** is ready for writing, **IChannel.Init** is called (this sends the initial connection handshake message). When the read file descriptor alerts the user that the **Socket** has data to read, **IChannel.Init** is called - this typically reads the next portion of the handshake. This process would continue until the connection is active.

A server application would typically register the **IChannel.Socket** with the read and exception file descriptor sets. When the read descriptor alerts the user that the **Socket** has data to read, **IChannel.Init** is called, which typically reads the initial portion of the handshake and sends out any necessary response. This process continues until the connection is active.

10.5.4 IChannel.Init Return Codes

The following table defines the return codes that can occur when using **IChannel.Init**.

RETURN CODE	DESCRIPTION
TransportReturnCode.SUCCESS	Indicates the initialization process completed successfully. The **IChannel.state** should be **ChannelState.ACTIVE**.
TransportReturnCode.FAILURE	Indicates that initialization has failed and cannot progress. The **IChannel.state** should be **ChannelState.CLOSED**, and the application should close the channel to clean up associated resources. For more details, refer to the **Error** content.
TransportReturnCode.CHAN_INIT_IN_PROGRESS	Indicates that initialization is still in progress. Check **InProgInfo.Flags** to determine whether the **Socket** changed. The **IChannel.State** should be **ChannelState.INITIALIZING**.
TransportReturnCode.CHAN_INIT_REFUSED	Indicates the connection was rejected. For more details, refer to the **Error** content.
TransportReturnCode.INIT_NOT_INITIALIZED	Indicates that the Transport is not initialized. For more details, refer to the **Error** content. For information on initializing, refer to Section 10.2.

Table 34: IChannel.Init Return Codes

10.5.5 IChannel.Init Example

The example below shows general use of **IChannel.Init**. Use of I/O notification is assumed, and the example assumes that the code is being executed due to some I/O notification.

```
/* IChannel.Init is typically called based on activity on the socketId, though a timer or
   looping can be used - the IChannel.Init function should continue to be called until the
   connection becomes active, at which point reading and writing can begin */
InProgInfo inProgInfo = new ();
if (channel.State == ChannelState.INITIALIZING)
{
   if ((retCode = channel.Init(inProgInfo, out Error error)) < TransportReturnCode.SUCCESS)
   {
      Console.Write("Error ({0}) (errno: {1}) encountered with Init Channel fd={2}.
         Error Text: {3}\n", error.ErrorId, error.SysError, channelFDValue, error.Text);
      CloseChannelCleanUpAndExit(channel, TransportReturnCode.FAILURE);
   }
   else
   {
      /* Deduce an action from return code of Channel.Init() */
      switch (retCode)
      {
         case TransportReturnCode.CHAN_INIT_IN_PROGRESS:
            {
               /* Switch to a new channel if required */

               if (inProgInfo.Flags == InProgFlags.SCKT_CHNL_CHANGE)
               {
```

```
                        /* SCKT_CHNL_CHANGE indicates that a socketId change has occurred as a result
                               of this call.
                        /*  New SocketFDValue need to be set to update current FD.
                        *  Shifting from the old socket to the new one handled by the library itself.
                        */

                        channelFDValue = inProgInfo.NewSocket.Handle.ToInt64();
                    }
                    else
                    {
                        Console.Write("Channel {0} in progress...\n", channelFDValue);
                    }
                }
                break;
            /* channel connection becomes active!
             *
             * Once a connection is established and transitions to the ChannelState.ACTIVE state,
             * this Channel can be used for other transport operations.
             */
             case TransportReturnCode.SUCCESS:
                {
                    Console.Write("Channel on fd {0} is now active - reading and writing can begin.\n",
                        channelFDValue);

                    /* Populate information from channel */
                    if ((retCode = channel.Info(channelInfo, out error)) != TransportReturnCode.SUCCESS)
                    {
                        Console.Write("Error ({0}) (errno: {1}) encountered with Init Channel fd={2}.
                            Error Text: {3}\n", error.ErrorId, error.SysError, channelFDValue, error.Text);
                        CloseChannelCleanUpAndExit(channel, TransportReturnCode.FAILURE);
                    }
                }
                break;
            default: /* Error handling */
                {
                    Console.Write("Bad return value fd={0} : <{1}>\n", channelFDValue, retCode);
                    /* Closes channel, cleans up and exits the application. */
                    CloseChannelCleanUpAndExit(channel, TransportReturnCode.FAILURE);
                }
                break;
            }
        }
    }
}
```

Code Example 8: `Channel` Initialization Process Using `IChannel.Init`

10.6 Reading Data

When a client or server **IChannel.State** is **ChannelState.ACTIVE**, an application can receive data from the connection by calling **IChannel.Read**. The arrival of this data is often announced by the I/O notification mechanism with which the **IChannel.Socket** is registered. The Transport reads data from the network as a byte stream, after which it determines **ITransportBuffer** boundaries and returns each buffer one by one. The **NumInputBuffers** connect or bind option controls the maximum length of the byte stream that the transport can internally process with each network read.

NOTE: When an **ITransportBuffer** is returned from **IChannel.Read**, the contents are only valid until the next call to **IChannel.Read**.

To reduce potentially unnecessary copies, returned information simply points into the internal **IChannel.Read** input buffer. If the application requires the contents of the buffer beyond the next **IChannel.Read** call, the application can copy the contents of the buffer and allow the user to control the duration of the life cycle of the memory.

If the connection uses compression, the **IChannel.Read** method will perform any necessary decompression prior to returning information to the application. For available compression types, refer to Section 10.4.3.

It is possible for **IChannel.Read** to succeed and return a NULL buffer. When this occurs, it indicates that a portion of a fragmented buffer has been received. The Transport Package internally reassembles all parts of the fragmented buffer and after processing the last fragment, returns the entire buffer to the user through **IChannel.Read**.

If a packed buffer is received, each call to **IChannel.Read** returns an individual message (i.e., portion of contents) from the packed buffer. Every subsequent call to **IChannel.Read** continues to return portions of the packed buffer until the buffer is emptied. Message packing is transparent to the application that receives a packed buffer. For more information about packing, refer to Section 10.11.

If a packed buffer is received and the application leverages WebSocket transport with JSON then **IChannel.Read** will return only one message with all the sending packed messages as items of JSON array.

10.6.1 IChannel.Read Method

METHOD	DESCRIPTION
IChannel.Read	Provides the user with data received from the connection. This method expects the **IChannel** to be in the active state. When data is available, an **ITransportBuffer** referring to the information is returned, which is valid until the next call to **IChannel.Read**. If a blocking I/O is used, the **IChannel.Read** method will not return until there is information to return or an error has occurred. A parameter passed into the function is used to convey return code information as well as communicate whether there is additional information to read. An I/O notification mechanism may not inform the user of this additional information as it has already been read from the socket and is contained in the IChannel.Read input buffer. Return values are described inSection 10.6.3.

Table 35: IChannel.Read Method

10.6.2 IChannel.Read Return Codes

The following table defines return codes that can occur when using **IChannel.Read**.

RETURN CODE	BUFFER CONTENTS	DESCRIPTION
TransportReturnCode.SUCCESS	Populated if the full buffer is available, null otherwise. The buffer's **length** indicates the number of bytes to which the **data** refers.	Indicates that the **IChannel.Read** call was successful and there are no remaining bytes in the input buffer. The I/O notification mechanism will notify the user when additional information arrives. The ping timer should be updated, refer to Section 10.12.
Any positive value > 0	Populated if full buffer is available, NULL otherwise. The buffer's **length** indicates the number of bytes to which the **data** refers.	Indicates that the **IChannel.Read** call was successful and there are remaining bytes in the input buffer. The I/O notification mechanism will not notify the user of these bytes. The **IChannel.Read** method should be called again to ensure that the remaining bytes are processed. The ping timer should be updated (for details, refer to Section 10.12). **NOTE:** If there are additional bytes to process, you should call **IChannel.Read** again. Because the bytes are already contained in the transport input buffer, an I/O notification mechanism will not alert the user of their presence.
TransportReturnCode.READ_WOULD_BLOCK	NULL	Indicates that the **IChannel.Read** call has nothing to return to the user.
TransportReturnCode.READ_PING	NULL	Indicates that a heartbeat message was received. The ping timer should be updated (for details, refer to Section 10.12).
TransportReturnCode.FAILURE	NULL	Indicates a failure condition, often that the connection is no longer available. The **IChannel** should be closed (for details, refer to Section 10.13). For more details, refer to **Error** content.
TransportReturnCode.READ_FD_CHANGE	NULL	Indicates that the connections **Socket** has changed. This can occur as a result of internal connection keep-alive mechanisms. The previous **Socket**is stored in the **IChannel.OldSocket** so it can be removed from the I/O notification mechanism. The **IChannel.NewSocket** contains the new file descriptor, which should be registered with the I/O notification mechanism.
TransportReturnCode.READ_IN_PROGRESS	NULL	Indicates that an **IChannel.Read** call on the **IChannel** is already in progress. This can be due to another thread performing the same operation.
TransportReturnCode.INIT_NOT_INITIALIZED	NULL	Indicates that the Transport has not been initialized. See the **Error** content for more details. For information on initializing, refer to Section 10.2.

Table 36: IChannel.Read Return Codes

10.6.3 IChannel.Read Example

The following example shows typical use of **IChannel.Read** and assumes use of an I/O notification mechanism. This code would be similar for client or server based **IChannel** structures.

```
ITransportBuffer msgBuf = channel.Read(readArgs, out error);

if (msgBuf != null)
{
   /* if a buffer is returned, we have data to process and code is success */
   /* set flag for server message received */
   receivedServerMsg = true;
}
else
{
   /* Deduce an action from the return code of IChannel.Read() */
   retCode = readArgs.ReadRetVal;
   switch (retCode)
   {
      /* Acknowledge that a ping has been received */
      case TransportReturnCode.READ_PING:
         /* Update ping monitor */
         /* set flag for server message received */
         receivedServerMsg = true;
         Console.Write("Ping message has been received successfully from the server due to
            ping message ...\n\n");
         break;

      /* Switch to a new channel if required */
      case TransportReturnCode.READ_FD_CHANGE:
         /* READ_FD_CHANGE indicates that a socketId change has occurred as a result of this call.
         *  New SocketFDValue need to be set to update current FD.
         *  Shifting from the old socket to the new one handled by the library itself.
         */

         channelFDValue = inProgInfo.NewSocket.Handle.ToInt64();
         break;

      case TransportReturnCode.READ_WOULD_BLOCK:
         /* Nothing to read */
         break;
      case TransportReturnCode.READ_IN_PROGRESS:
         /* Reading from multiple threads */
         break;
      case TransportReturnCode.INIT_NOT_INITIALIZED:
      case TransportReturnCode.FAILURE:/* fall through to default. */
      default: /* Error handling */

         if (retCode < 0)
         {
            Console.Write("Error ({0}) (errno: {1}) encountered with Read. Error Text: {2}\n",
```

```
                    error.ErrorId, error.SysError, error.Text);
                /* Closes channel/connection, cleans up and exits the application. */
                CloseChannelCleanUpAndExit(channel, TransportReturnCode.FAILURE);
            }
            break;
    }
}
```

Code Example 9: Receiving Data Using `IChannel.Read`

10.7 Writing Data: Overview

When a client or server **`IChannel.State`** is **`ChannelState.ACTIVE`**, it is possible for an application to write data to the connection. Writing involves a multi-step process. Because the Transport provides efficient buffer management, the user must obtain a **`ITransportBuffer`** from the Transport buffer pool (refer to Section 10.8). This can be the guaranteed output buffer pool associated with an **`IChannel`** or the shared buffer pool associated with an **`IServer`**.

After a buffer is acquired, the user can populate the **`ITransportBuffer.Data`**.

At this point, the user can choose to pack additional information into the same buffer (refer to Section 10.11) or add the buffer to the transports outbound queue (refer to Section 10.9). If queued information cannot be passed to the network, a function is provided to allow the application to continue attempts to flush data to the connection (refer to Section 10.10.2). An I/O notification mechanism can be used to help with determining when the network is able to accept additional bytes for writing. The Transport can continue to queue data, even if the network is unable to write. The following figure depicts this process and the following sections describe the functionality used to write information to the connection.

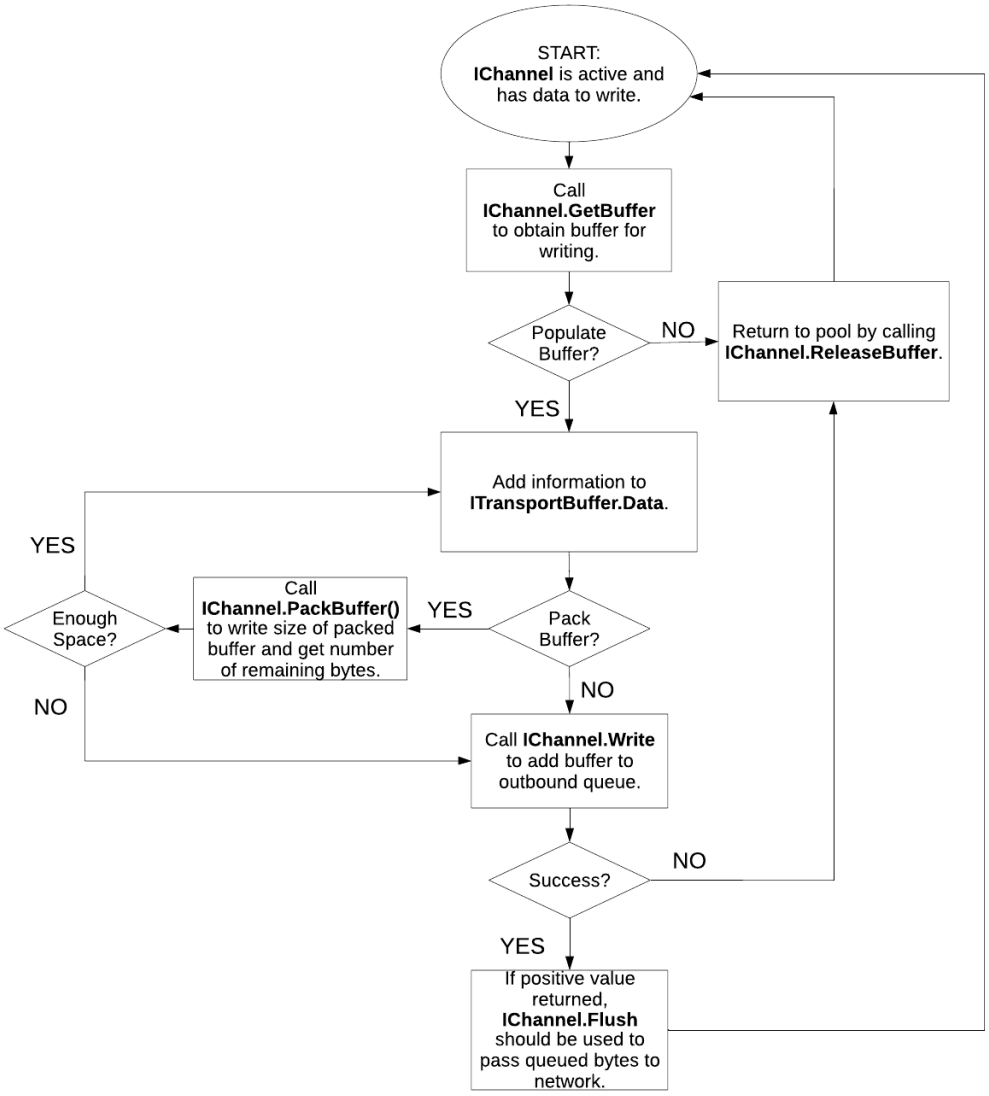


Figure 32. Enterprise Transport API Writing Flow Chart

10.8 Writing Data: Obtaining a Buffer

To write information, the user must obtain an **ITransportBuffer** from the Transport buffer pool. This buffer can originate from the guaranteed output buffer pool associated with the **IChannel** or the shared buffer pool associated with the **IServer**. After acquiring a buffer, the user can populate the **ITransportBuffer.Data**. If the buffer is not used or the **IChannel.Write** method call fails, the buffer must be released back into the pool to ensure proper reuse and cleanup. If the buffer is successfully passed to **IChannel.Write**, when flushed to the network the buffer will be returned to the correct pool by the transport.

The number of buffers made available to an **IChannel** is configurable through **ConnectOptions** or **BindOptions**. When connecting, the **GuaranteedOutputBuffers** setting controls the number of available buffers. When connections are accepted by an **IServer**, the **MaxOutputBuffers** parameter controls the number of available buffers per connection. This value is the sum of the number of **GuaranteedOutputBuffers** and any available shared pool buffers. For more information about available **Transport.Connect()** and **Transport.Bind()** methods, refer toTable 24 andTable 33.

10.8.1 Buffer Management Functions

METHOD NAME	DESCRIPTION
IChannel.GetBuffer	Obtains a **ITransportBuffer** of the requested size from the guaranteed or shared buffer pool. If the requested size is larger than the **MaxFragmentSize**, the transport will create and return the buffer to the user. When written, this buffer will be fragmented by the **IChannel.Write** method (refer to Section 10.9). Because of some additional book keeping required when packing, the application must specify whether a buffer should be 'packable' when calling **IChannel.GetBuffer**. For more information on packing, refer to Section 10.11. For performance purposes, an application is not permitted to request a buffer larger than **MaxFragmentSize** and have the buffer be 'packable.' If the buffer is not used or the **IChannel.Write** call fails, the buffer must be returned to the pool using **IChannel.ReleaseBuffer**. If the **IChannel.Write** call is successful, the buffer will be returned to the correct pool by the transport. Return values are described in Table 45.
IChannel.ReleaseBuffer	Releases a **ITransportBuffer** back to the correct pool. This should only be called with buffers that originate from **IChannel.GetBuffer** and are not successfully passed to **IChannel.Write**.
IChannel.BufferUsage	Returns the number of buffers currently in use by the **IChannel**, this includes buffers that the application holds and buffers internally queued and waiting to be flushed to the connection.
IServer.BufferUsage	Returns the number of shared pool buffers currently in use by all channels connected to the **IServer**, this includes shared pool buffers that the application holds and shared pool buffers internally queued and waiting to be flushed.

Table 37: Buffer Management Methods

10.8.2 IChannel.GetBuffer Return Codes

The following table defines and values that can occur while using **IChannel.GetBuffer**.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	A **ITransportBuffer** is returned to the user.
NULL buffer returned Error Code: TransportReturnCode.NO_BUFFERS	NULL is returned to the user. This value indicates that there are no buffers available to the user. See **Error** content for more details. This typically occurs because all available buffers are queued and pending flushing to the connection. The application can use **IChannel.Flush** to attempt releasing buffers back to the pool (refer to Section 10.10.2). Additionally, the **IChannel.IOCtl** method can be used to increase the number of **guaranteedOutputBuffers** (refer to Section 10.14).
NULL buffer returned Error Code: TransportReturnCode.FAILURE	NULL is returned to the user. This value indicates that some type of general failure has occurred. The **IChannel** should be closed, refer to Section 10.13. See **Error** content for more details.
NULL buffer returned Error Code: TransportReturnCode.INIT_NOT_INITIALIZED	Indicates that the Transport has not been initialized. See the **Error** content for more details. For information on initializing, refer to Section 10.2.

Table 38: IChannel.GetBuffer Return Codes

10.9 Writing Data to a Buffer

After an **ITransportBuffer** is obtained from **IChannel.GetBuffer** and populated with the user's data, the buffer can be passed to the **IChannel.Write** method. Though the name seems to imply it, this method may not write the contents of the buffer to the connection. By queuing, the Transport can attempt to use the network layer more efficiently by combining multiple buffers into a single socket write operation. Additionally, queuing allows the application to continue to 'write' data, even while the network has no available space in the output buffer. If **IChannel.Write** does not pass all data to the socket, unwritten data will remain in the outbound queue for future writing. If an error occurs, any **ITransportBuffer** that has not been successfully passed to **IChannel.Write** should be released to the pool using **IChannel.ReleaseBuffer**. The following table describes the **IChannel.Write** method as well as some additional parameters associated with it.

The example in demonstrates the use of **IChannel.GetBuffer** and **IChannel.ReleaseBuffer**.

10.9.1 IChannel.Write Method

METHOD	DESCRIPTION
IChannel.Write	Performs any writing or queuing of data. This function expects the **IChannel** to be in the active state and the buffer to be properly populated, where length reflects the actual number of bytes used. If blocking I/O is used, the **IChannel.Write** method will not return until data was written to the connection or an error has occurred. This function allows for several modifications to be specified for this call. For more information, refer to Section 10.9.2. The Transport supports writing data at different priority levels. For more details on priority levels, refer to Section 10.10.1. The application can pass in two integer values used for reporting information about the number of bytes that will be written. • The **UncompressedBytesWritten** parameter will return the number of bytes to be written, including any transport header overhead but not taking into account any compression. • The **BytesWritten** parameter will return the number of bytes to be written, including any transport header overhead and taking into account any compression. If compression is disabled, **UncompressedBytesWritten** and **BytesWritten** should match. The number of bytes saved through the compression process can be calculated by (**UncompressedBytesWritten** - **BytesWritten**). Return values are described in Section 10.9.5.
	NOTE: Before passing a buffer to **IChannel.Write**, it is required that the application set length to the number of bytes actually used. This ensures that only the required bytes are written to the network.

Table 39: IChannel.Write Method

10.9.2 Write Flags Values

NOTE: Before passing a buffer to **IChannel.Write**, it is required that the application set **Data** to **ITransportBuffer.Data** properly. This ensures that only the required data are written to the network.

WRITEFLAG	MEANING
NO_FLAGS	No modification will be performed to this **IChannel.Write** operation.
DO_NOT_COMPRESS	Though the connection might have compression enabled, this flag value indicates that this message will not be compressed. This flag value applies only to the contents of the **ITransportBuffer** passed in with this **IChannel.Write** call.
DIRECT_SOCKET_WRITE	When set, the **IChannel.Write** method will attempt to pass the contents of the **ITransportBuffer** directly to the socket write operation, bypassing any internal transport queuing. If any information is currently queued, this buffer will also be queued and the **IChannel.Flush** method will be invoked to ensure proper ordering of outbound data. Use of this modification will result in a higher CPU writing cost however it might decrease latency when internal queues are empty. This can be useful for writing at low data rates or when the return codes from **IChannel.Write** and **IChannel.Flush** indicate that data is not queued.

Table 40: WriteFlags Values

10.9.3 Compression

The **IChannel.Write** method performs all necessary compression associated with the connection. Because of information order changes, compression can only be applied to a single priority level. If writing data using different priorities, the first priority level used will leverage compression and all other priority levels will be sent uncompressed. For available compression types, refer to Section 10.4.3.

10.9.4 Fragmentation

In addition to compression, the **IChannel.Write** method performs any necessary fragmentation of large buffers. This fragmentation process subdivides one large buffer into smaller **MaxFragmentSize** portions, where each part is placed into a buffer acquired from the pool associated with the **IChannel**. If the fragmentation cannot fully complete, often due to a shortage of pool buffers, this is indicated by the **TransportReturnCode.WRITE_CALL_AGAIN** code. In this situation, the application should use **IChannel.Flush** to write queued buffers to the connection - this will release buffers back to the pool. When additional pool buffers are available, the application can call **IChannel.Write** with the same buffer to continue the fragmentation process from where it left off. The Transport keeps track of necessary information to identify and track individual fragmented messages. This allows an application to write unrelated messages between portions of a fragmented buffer as well as writing multiple fragmented messages that may be interleaved.

NOTE: In the event that the connection is unable to accept additional bytes to write, the Transport queues on the user's behalf. The application can attempt to pass queued data to the network by using the **IChannel.Write**method.

10.9.5 IChannel.Write Return Codes

The following table lists all that can occur when using the **IChannel.Write** method.

RETURN CODE	DESCRIPTION
TransportReturnCode.SUCCESS	Indicates that the **IChannel.Write** method was successful and additional bytes have not been internally queued. The **IChannel.Flush** method does not need to be called. The application should not release the **ITransportBuffer**; the Enterprise Transport API will release it.
Any positive value > 0	Indicates that the **IChannel.Write** method has succeeded and there is information internally queued by the transport. To pass internally queued information to the connection, the **IChannel.Flush** method must be called. This information can be queued because there is not sufficient space in the connections output buffer. An I/O notification mechanism can be used to indicate when the has write availability. The application should not release the **ITransportBuffer**; the Enterprise Transport API will release it.
TransportReturnCode.WRITE_FLUSH_FAILED	Indicates that the **IChannel.Write** method has succeeded, however an internal attempt to flush data to the socket has failed - the channel's state should be inspected. This might not be a failure condition and can occur if there is no available socket output buffer space. If the flush failure is unrecoverable, the **IChannel.State** will transition to **ChannelState.CLOSED**. If the connection closes, **Error** information will be populated. The application should not release the **ITransportBuffer**; the Enterprise Transport API will release it.
TransportReturnCode.WRITE_CALL_AGAIN	Indicates that a large buffer could not be fully fragmented with this **IChannel.Write** call. This is typically due to all pool buffers being unavailable. An application can use **IChannel.Flush** to free up pool buffers or use **IChannel.IOCtl** to increase the number of available pool buffers. After pool buffers become available again, the same buffer should be used to call **IChannel.Write** an additional time (the same priority level must be used to ensure fragments are ordered properly). This will continue the fragmentation process from where it left off. If the application does not subsequently pass the **ITransportBuffer** to **IChannel.Write**, the buffer should be released by calling **IChannel.ReleaseBuffer**.
TransportReturnCode.FAILURE	Indicates that a general write failure has occurred. The **IChannel** should be closed (refer to Section 10.13). For more details, refer to any **Error** content. The application should release the **ITransportBuffer** by calling **IChannel.ReleaseBuffer**.
TransportReturnCode.BUFFER_TOO_SMALL	Indicates that either the buffer has been corrupted, possibly by exceeding the allowable length, or it is not a valid pool buffer. For more details, refer to any **Error** content. If this **ITransportBuffer** was obtained from **IChannel.GetBuffer**, the application should release it by calling **IChannel.ReleaseBuffer**.
TransportReturnCode.INIT_NOT_INITIALIZED	Indicates that the Transport has not been initialized. • For more details, refer to any **Error** content. • For information on initializing, refer to Section 10.2. The application's attempt to call **IChannel.GetBuffer** should have failed for the same reason, so an **ITransportBuffer** should not be present.

Table 41: IChannel.Write Return Codes

10.9.6 IChannel.GetBuffer and IChannel.Write Example

The following example shows typical use of **IChannel.GetBuffer** and **IChannel.Write**. This code would be similar for client or server based **IChannel** structures.

```
/* Ask for a 6000 byte packable buffer to write multiple messages into */
if ((buffer = chnl.GetBuffer(6000, true, out Error error)) != null)
{
   /* if a buffer is returned, we can populate and write, encode a Msg into the buffer */
   /* set the buffer and version on an EncodeIterator */
   encIter.Clear();
   encIter.SetBufferAndRWFVersion(buffer, chnl.MajorVersion, chnl.MinorVersion);
   /* populate message and encode it. For more details on message encoding, refer to Section 12.2.10.1 */
   retCode = msg.Encode(encIter);
   /* Instead of writing, lets continue packing messages into the buffer */
   /* This will take the existing buffer and return how many bytes remain to continue encoding into */
   if ((retCode = chnl.PackBuffer(buffer, out Error error)) < TransportReturnCode.SUCCESS)
   {
      Console.WriteLine("Error ({0}) (errno: {1}) encountered with PackBuffer. Error Text: {2}",
         error.ErrorId, error.SysError, error.Text);
      /* Buffer must be released - return code from ReleaseBuffer can be checked */
      chnl.ReleaseBuffer(buffer, out error);
      /* Connection should be closed, return failure */
   }
   /* check retCode, if there is enough bytes remaining, continue to pack additional messages */
   /* encode an additional message *
   /* set the buffer and version on an EncodeIterator */
   encIter.SetBufferAndRWFVersion(buffer, chnl.MajorVersion, chnl.MinorVersion);
   /* populate message and encode it - for more details on message encoding, refer to Section 12.2.10.1
   */
   retCode = msg.Encode(encIter);
   /* Instead of writing, let's continue packing messages into the buffer */
   /* This will take the existing buffer and return the number of bytes available for encoding */
   if ((retCode = chnl.packBuffer(buffer, out error)) < TransportReturnCodes.SUCCESS)
   {
      Console.WriteLine("Error ({0}) (errno: {1}) encountered with PackBuffer. Error Text: {2}",
         error.ErrorId, error.SysError, error.Text);
      /* Buffer must be released - return code from ReleaseBuffer can be checked */
      chnl.ReleaseBuffer(buffer, out error);
      /* Connection should be closed, return failure */
   }
   /* Packing can continue like this until the application determines its time to stop - this can be due
   to the buffer not containing enough space for an additional message, a timer alerting that enough
   pack time has elapsed, etc */
   /* After packing is complete, write the buffer as normal */
   WriteArgs.Priority = WritePriorities.HIGH;
   retCode = chnl.Write(buffer, writeArgs, out error);
   /* For a full, write error-handling example, refer to the Example in Section 10.9.6. */
}
else
```

```
{
    /* Use the Flush method (Section 10.10.2) to free buffers back to the pool */
}
```

Code Example 10: Writing Data Using `IChannel.Write`, `IChannel.GetBuffer`, and `IChannel.ReleaseBuffer`

10.10 Managing Outbound Queues

Because it may not be possible for the **IChannel.Write** method to pass all data to the underlying socket, some data may be queued by the Transport. Applications can use the **IChannel.Flush** method to continue attempting to pass queued data to the connection.

10.10.1 Ordering Queued Data: IChannel.Write Priorities

Using the **IChannel.Write** method, an application can associate a priority with each **ITransportBuffer**. Priority information is used to determine outbound ordering of data, and can allow for higher priority information to be written to the connection before lower priority data, even if the lower priority data was passed to **IChannel.Write** first. Only queued data will incur any ordering changes due to priority, and data directly written to the socket by **IChannel.Write** will not be impacted.

Priority ordering occurs as part of the **IChannel.Flush** call (refer to Section 10.10.2), where the **PriorityFlushStrategy** determines how to handle each priority level. The default **PriorityFlushStrategy** writes buffers in the order: High, Medium, High, Low, High, Medium. This provides a slight advantage to the medium priority level and a greater advantage to high priority data. Data order is preserved within each priority level (thus, if all buffers are written with the same priority, data is not reordered). If a particular priority level being flushed does not have content, **IChannel.Flush** will move to the next priority in the **PriorityFlushStrategy**. The **PriorityFlushStrategy** can be changed for each **IChannel** by using the **IChannel.IOCtl** method (refer to Section 10.14).

10.10.1.1 Priority Ordering

The following figure presents an example of a possible priority write ordering. On the left, there are three queues and each queue is associated with one of the available **IChannel.Write** priority values. As the user calls **IChannel.Write** and assigns priorities to their buffers, they will be queued at the appropriate priority level. As the **IChannel.Flush** method is called, buffers are removed from the queues in a manner that follows the **PriorityFlushStrategy**.

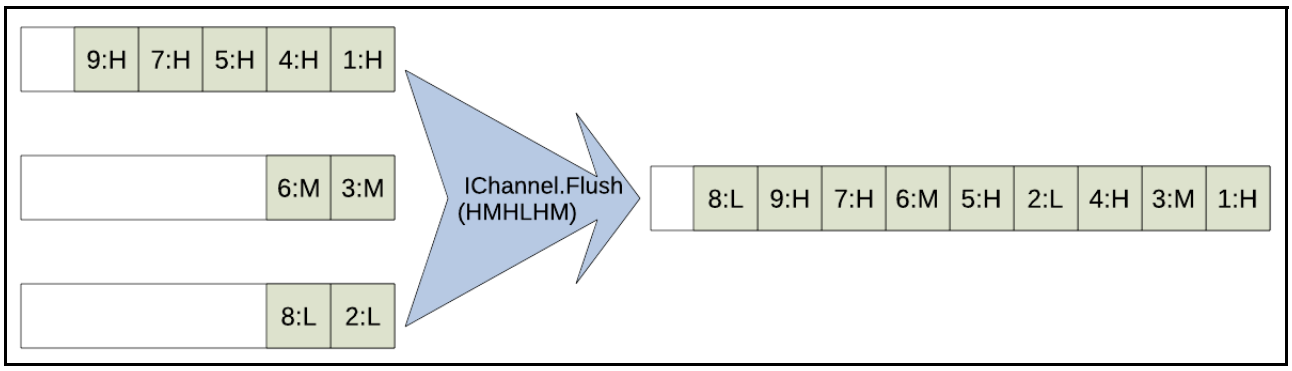


Figure 33. IChannel.Write Priority Scenario

On the left side of the figure there are three outbound queues, one for each priority value. As buffers enter the queues (as a result of an **IChannel.Write** call), they are marked with a number and the priority value associated with their queue. The number indicates the order the buffers were passed to **IChannel.Write**, so the buffer marked **1** was the first buffer into **IChannel.Write**, the buffer marked **5** was the 5th buffer into **IChannel.Write**. Buffers are marked **H** if they are in the high priority queue, **M** if they are in the medium priority queue, or **L** if they are in the low priority queue. Buffers leave the queue (as a result of a **IChannel.Flush** call) in the order specified by the **PriorityFlushStrategy**, which by default is **HMHLHM**. In , the queue on the right side represents the order in which buffers are written to the network and the order that they will be returned when **IChannel.Read** is called. The buffers will still be marked with their ***number*:*priority*** information so it is easy to see how data is reordered by any priority writing.

Notice that though data was reordered between various priorities, individual priority levels are not reordered. Thus, all buffers in the high priority are written in the order they are queued, even though some medium and low buffers are sent as well.

10.10.1.2 WritePriorities Values

PRIORITYVALUE	MEANING
HIGH	If not directly written to the socket, this **ITransportBuffer** will be flushed at the high priority.
MEDIUM	If not directly written to the socket, this **ITransportBuffer** will be flushed at the medium priority.
LOW	If not directly written to the socket, this **ITransportBuffer** will be flushed at the low priority.

Table 42: WritePriorities Values

10.10.2 IChannel.Flush Method

If all available output space is used for a connection, data might be queued as a result. An I/O notification mechanism can be used to alert the application when output space becomes available on a connection.

NOTE: The return value from **IChannel.Flush** indicates whether there are any queued bytes left to pass to the connection. If this is a positive value (typical when operating system output buffers lack space), the application should continue to call **IChannel.Flush** until all bytes have been written.

METHOD NAME	DESCRIPTION
IChannel.Flush	Writes queued data to the connection. This method expects the **IChannel** to be in the active state. If data is not queued, the **IChannel.Flush** method is not required and should return immediately. This method performs any buffer reordering that might occur due to priorities passed in on the **IChannel.Write** method. For more information about priority writing, refer to Section 10.10.1. Return values are described in Table 51.

Table 43: IChannel.Flush Method

10.10.3 IChannel.Flush Return Codes

The following table defines the return **TransportReturnCodes** that can occur when using **IChannel.Flush**.

RETURN CODE	DESCRIPTION
TransportReturnCode.SUCCESS	Indicates that the **IChannel.Flush** method has succeeded and additional bytes are not internally queued. The **IChannel.Flush** method need not be called.
Any positive value > 0	Indicates that the **IChannel.Flush** method has succeeded, however data is still internally queued by the transport. The **IChannel.Flush** method must be called again. Data might still be queued because the connections output buffer does not have sufficient space. An I/O notification mechanism can indicate when the **Socket** has write availability.
TransportReturnCode.FAILURE	Indicates that a general failure has occurred, often because the underlying connection is unavailable or closed. The **IChannel** should be closed (refer to Section 10.13). For more details, refer to the **Error** content.
TransportReturnCode.INIT_NOT_INITIALIZED	Indicates that the Transport is not initialized. For more details, refer to the **Error** content. For information on initializing, refer to Section 10.2.

Table 44: IChannel.Flush Return Codes

10.10.4 IChannel.Flush Example

The following example shows typical use of **IChannel.Flush**. This example assumes use of an I/O notification mechanism. This code would be similar for client or server based **IChannel** structures.

```
/* Channel.Flush() use, be sure to keep track of the return values from flush so data is not stranded in
the output buffer - flush may need to be called again to continue attempting to pass data to th
connection */
/* Assuming this section of code was called because of a write selector notification *
/* this section of code was called because of a write file descriptor alert *

TransportReturnCode retCode;
if ((retCode = channel.Flush(out error)) > TransportReturnCode.SUCCESS)
{
   /* There is still data left to flush, leave our write notification enabled so we get called again.
    * If everything wasn't flushed, it usually indicates that the TCP output buffer cannot accept
      more yet */
}
else
{
   switch (retCode)
   {
      case TransportReturnCode.SUCCESS:
         {
            /* Everything has been flushed, no data is left to send - unset/clear write fd notification
            */
            opWrite = false;
            writeSocketList.Clear();
         }
         break;
      case TransportReturnCode.INIT_NOT_INITIALIZED:
      case TransportReturnCode.FAILURE:
      default:
         {
            Console.WriteLine("Error ({0}) (errno: {1}) encountered with Init Channel fd={2}.
                  Error Text: {3}", error.ErrorId, error.SysError, channelFDValue, error.Text);
            CloseChannelCleanUpAndExit(channel, error, TransportReturnCode.FAILURE);
         }
         break;
   }
}
```

Code Example 11: IChannel.Flush Use

10.11 Packing Additional Data into a Buffer

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

It is up to the writing application to determine when to stop packing, and the mechanism used can vary greatly. One simple algorithm is to pack a fixed number of messages each time. A slightly more complex technique could use the returned length from **`IChannel.PackBuffer`** to determine the amount of remaining space and pack until the buffer is nearly full. Both of these mechanisms can introduce a variable amount of latency as they both depend on the rate at which data arrives (i.e., the packed buffer will not be written until enough data arrives to fill it). One method that can balance this is to use a timer to limit the amount of time a packed buffer is held. If the buffer is full prior to the timer expiring, the data is written, otherwise whenever the timer expires, whatever is in the buffer will be written (regardless of the amount of data in the buffer). This limits latency to a maximum, acceptable amount as set by the duration of the timer.

The **`IChannel.PackBuffer`** method packs multiple messages into one **`ITransportBuffer`**.

METHOD	DESCRIPTION
IChannel.PackBuffer	Packs the contents of a passed-in **`ITransportBuffer`** and returns the number of bytes remaining in the **`ITransportBuffer`**. An application can use the length returned to determine the amount of space available to continue packing buffers. For a buffer to allow packing, it must be requested from **`IChannel.GetBuffer`** as 'packable' and cannot exceed the **`MaxFragmentSize`**. Return values are described in Table 65.

Table 45: `IChannel.PackBuffer` Method

10.11.1 IChannel.PackBuffer Return Values

The following table defines return and error code values that can occur when using **`IChannel.PackBuffer`**.

RETURN CODE	DESCRIPTION
0 or greater Success Case	Indicates the amount of available bytes remaining in the buffer for packing. Zero means that bytes are not available for packing.
Less than 0 Failure Case	This value indicates that some type of general failure has occurred. The **`IChannel`** should be closed (refer to Section 10.13). For more details, refer to **`Error`** content.

Table 46: `IChannel.PackBuffer` Return Values

10.11.2 Example: IChannel.GetBuffer, IChannel.PackBuffer, and IChannel.Write

The following example shows typical use of **IChannel.GetBuffer**, **IChannel.PackBuffer**, and **IChannel.Write**. This code would be similar for client or server based **IChannel** objects.

```
/* IChannel.GetBuffer(), IChannel.PackBuffer() and IChannel.Write() use, be sure to keep track of the
return values from write so data is not stranded in the output buffer - flush may be required to
continue attempting to pass data to the connection */
ITransportBuffer buffer = null;
EncodeIterator encIter = new EncodeIterator();
Msg msg = new Msg();
WriteArgs writeArgs = new WriteArgs();

/* Ask for a 6000 byte packable buffer to write multiple messages into */
if ((buffer = chnl.GetBuffer(6000, true, out Error error)) != null)
{
   /* if a buffer is returned, we can populate and write, encode a Msg into the buffer */
   /* set the buffer and version on an EncodeIterator */
   encIter.Clear();
   encIter.SetBufferAndRWFVersion(buffer, chnl.MajorVersion, chnl.MinorVersion);
   /* populate message and encode it. For more details on message encoding, refer to Section 12.2.10.1 */
   retCode = msg.Encode(encIter);
   /* Instead of writing, lets continue packing messages into the buffer */
   /* This will take the existing buffer and return how many bytes remain to continue encoding into */
   if ((retCode = chnl.PackBuffer(buffer, out error)) < TransportReturnCode.SUCCESS)
   {
      Console.WriteLine("Error ({0}) (errno: {1}) encountered with PackBuffer. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
      /* Buffer must be released - return code from releaseBuffer can be checked */
      chnl.ReleaseBuffer(buffer, out error);
      /* Connection should be closed, return failure */
   }
   /* check retCode, if there is enough bytes remaining, continue to pack additional messages */
   /* encode an additional message */
   /* set the buffer and version on an EncodeIterator */
   encIter.SetBufferAndRWFVersion(buffer, chnl.MajorVersion, chnl.MinorVersion);
   /* populate message and encode it - for more details on message encoding, refer to Section 12.2.10.1
   */
   */
   retCode = msg.Encode(encIter);
   /* Instead of writing, let's continue packing messages into the buffer */
   /* This will take the existing buffer and return the number of bytes available for encoding */
   if ((retCode = chnl.PackBuffer(buffer, out error)) < TransportReturnCode.SUCCESS)
   {
      Console.WriteLine("Error ({0}) (errno: {1}) encountered with PackBuffer. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
      /* Buffer must be released - return code from releaseBuffer can be checked */
      chnl.ReleaseBuffer(buffer, out error);
      /* Connection should be closed, return failure */
   }
   /* Packing can continue like this until the application determines its time to stop - this can be due
```

```
   to the buffer not containing enough space for an additional message, a timer alerting that enough
   pack time has elapsed, etc */
   /* After packing is complete, write the buffer as normal */
   writeArgs.Priority = WritePriorities.HIGH;
   retCode = chnl.Write(buffer, writeArgs, out error);
   /* For a full, write error-handling example, refer to the Example in Section 10.9.6. */
}
else
{
   /* Use the flush method (Section 10.10.2) to free buffers back to the pool */
}
```

Code Example 12: Message Packing Using `IChannel.PackBuffer`

10.12 Ping Management

Ping or heartbeat messages indicate the continued presence of an application. These are typically required only when no other data is exchanged. For example, there may be long periods of time that elapse between requests made from a consumer application. In this situation, the consumer sends periodic heartbeat messages to inform the providing application that it is still connected. Because the provider application is likely sending data more frequently (providing updates on any streams the consumer has requested), the provider might not need to send heartbeats (as the other data sufficiently announces its continued presence). The application is responsible for managing the sending and receiving of heartbeat messages on each connection.

10.12.1 Ping Timeout

Applications are able to configure their desired **PingTimeout** values, where the ***ping timeout*** is the point at which a connection is terminated due to inactivity. Heartbeat messages are typically sent every one-third of the **PingTimeout**, ensuring that heartbeats are exchanged prior to a ping timeout. This can be useful for detecting a connection loss prior to any kind of network or operating system notification.

PingTimeout values are negotiated between a connecting client application and the server application, where the server can specify a minimum allowable ping timeout (via the **MinPingTimeout** option) and the direction in which heartbeats flow (via **ServerToClientPings** and **ClientToServerPings**). For more information on specifying these options, refer to Section 10.3.2.1 and Section 10.4.1.1. During negotiation, the lowest **PingTimeout** value is selected. Because **MinPingTimeout** sets the lowest possible value, if a client's specified **PingTimeout** value is less than **MinPingTimeout**, the connection uses the **MinPingTimeout** as its **PingTimeout** value. After a connection transitions to the active state, the negotiated **PingTimeout** is available through the **IChannel.PingTimeout**.

The Transport uses the following formula to determine the negotiated **PingTimeout** value:

```
/* Determine lesser of client or servers PingTimeout */
if (client.PingTimeout < server.PingTimeout)
   connection.PingTimeout = client.PingTimeout;
else
   connection.PingTimeout = server.PingTimeout;
/* Determine whether timeout is less than minimum allowable timeout */
if (connection.PingTimeout < server.MinPingTimeout)
   connection.PingTimeout = server.MinPingTimeout;
```

Code Example 13: Ping Negotiation Calculation

10.12.2 IChannel.Ping Method

An application typically monitors both messages and heartbeats. If bytes are flushed to the network, this is considered sufficient as a heartbeat so any timer mechanism associated with sending heartbeats can be reset. When bytes are received or **IChannel.Read** returns **TransportReturnCode.READ_PING** (refer to Section 10.6), this is comparable to receiving a heartbeat so any timer mechanism associated with receiving heartbeats can be reset. If either the sending or receiving heartbeat timer mechanism reaches or surpasses the **IChannel.PingTimeout** value, the connection should be closed.

The following table describes the **IChannel.Ping** method, used to send heartbeat messages.

METHOD	DESCRIPTION
IChannel.Ping	Attempts to write a heartbeat message on the connection. This method expects an active **IChannel**. If an application calls the **IChannel.Ping** method while other bytes are queued for output, the Transport layer suppresses the heartbeat message and attempts to flush bytes to the network on the user's behalf. Return values are described in Table 55.

Table 47: IChannel.Ping method

10.12.3 IChannel.Ping Return Values

The following table defines the return codes **TransportReturnCodes** that can occur when using **IChannel.Ping**.

RETURN CODE	DESCRIPTION
TransportReturnCode.SUCCESS	Indicates that the **IChannel.Ping** method succeeded and additional bytes are not internally queued.
Any positive value > 0	Indicates that queued data was sent as a heartbeat but data is still internally queued by the transport. The **IChannel.Flush** method must be called to continue passing queued bytes to the connection. Data might still be queued because the connections output buffer does not have sufficient space. An I/O notification mechanism indicate when the **Socket** has write availability.
TransportReturnCode.FAILURE	This value indicates that some type of general failure has occurred. The **IChannel** should be closed (refer to Section 10.13). For more details, refer to the **Error** content.

Table 48: IChannel.Ping Return Codes

10.12.4 IChannel.Ping Example

The following example shows typical use of **IChannel.Ping**. This example assumes use of some kind of timer mechanism to execute when necessary. This code would be similar for client or server based **IChannel** structures.

```
/* IChannel.Ping() use - this demonstrates sending of heartbeats */
/* Additionally, an application should determine if data or pings have been received, if not application
should determine if PingTimeout has elapsed, and if so connection should be closed */
/* First, send our ping, if there is other data queued, that will be flushed instead */
if ((retCode = chnl.Ping(out Error error)) > TransportReturnCode.SUCCESS)
{
   /* There is still data left to flush, leave our write notification enabled so we get called again,
   If everything wasn't flushed, it usually indicates that the TCP output buffer cannot accept more yet
   */
}
else
{
   switch (retCode)
   {
   case TransportReturnCode.SUCCESS:
      /* Ping message has been sent successfully */
   break;
   case TransportReturnCode.FAILURE:
      Console.WriteLine("Error ({0}) (errno: {1}) encountered with Ping. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
      /* Connection should be closed, return failure */
      break;
   default:
      Console.WriteLine("Unexpected return code ({0}) encountered!", retCode);
      /* Likely unrecoverable, connection should be closed */
   }
}
```

Code Example 14: IChannel.Ping Use

10.13 Closing Connections

10.13.1 Functions for Closing Connections

When an error occurs on a connection or an **IChannel** is being disconnected, the **IChannel.Close** method should be called to perform any necessary cleanup and to shutdown the underlying socket. This will release any pool-based resources back to their respective pools. If the application is holding any buffers obtained from **IChannel.GetBuffer**, they should be released using **IChannel.ReleaseBuffer** prior to closing the channel.

If a server is being shut down, use the **IServer.Close** method to close the listening socket and perform any necessary cleanup. All currently connected **IChannel**s will remain open. This allows applications to continue sending and receiving data, while preventing new applications from connecting. The server has the option of calling **IChannel.Close** to shut down any currently connected applications.

METHOD	DESCRIPTION
IChannel.Close	Closes a client- or server-based **IChannel**. This releases any pool-based resources back to their respective pools, closes the connection, and performs any additional necessary cleanup.
	NOTE: If an application is multi-threaded, all other threads that depend on the closed channel should complete their use prior to calling **IChannel.Close**.
IServer.Close	Closes a listening socket associated with an **IServer**. The **IServer.Close** releases any pool-based resources back to their respective pools, closes the listening socket, and performs any additional necessary cleanup. Established connections remain open, allowing for continued exchange of data. If needed, the server can use **IChannel.Close** to shutdown any remaining connections.

Table 49: Connection Closing Functionality

10.13.2 Close Connections Example

The following example shows typical use of **IChannel.Close** and **IServer.Close**.

```
/* IChannel.Close() */
if (chnl.Close(out Error error) < TransportReturnCode.SUCCESS)
{
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with channel close. Error Text: {2}",
   error.ErrorId, error.SysError, error.Text);
}
/* IServer.Close() */
if (srvr.Close(out error) < TransportReturnCode.SUCCESS)
{
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with server close. Error Text: {2}",
   error.ErrorId, error.SysError, error.Text);
}
```

Code Example 15: Closing a Connection Using IChannel.Close and IServer.Close

10.14 Utility Methods

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

10.14.1 General Transport Utility Methods

METHOD NAME	DESCRIPTION
IChannel.Info	Allows the application to query **IChannel** negotiated parameters and settings and retrieve all current settings. This includes **MaxFragmentSize** and negotiated compression information as well as many other values. See the **ChannelInfo** structure, defined in Table 51, for a full list of available settings.
IServer.Info	Allows the application to query **IServer** related values, such as current and peak shared pool buffer usage statistics. This populates the **ServerInfo** structure, defined in Table 53.
IChannel.IOCtl	Allows the application to change various settings associated with the **IChannel**. The available options are defined in Table 54.
IServer.IOCtl	Allows the application to change various settings associated with the **IServer**. The available options are defined in Table 55.

Table 50: Transport Utility Methods

10.14.2 ChannelInfo Methods

The following table describes the values available to the user through using the **IChannel.Info** method. This information is returned as part of the **ChannelInfo** object.

METHOD	DESCRIPTION
ClientToServerPings	Gets whether the client is expected to send heartbeat messages: • If set to **true**, heartbeat messages must flow from client to server. • If set to **false**, the client is not required to send heartbeats. LSEG Real-Time Distribution System and other LSEG components typically require this value to be set to **true**.
ComponentInfo	One **ComponentInfo** object will be present for each connected device that supports connected component versioning. For more detailed information on the **ComponentInfo** structure, refer to Section 10.14.4.
CompressionThreshold	Gets the compression threshold. Messages smaller than the threshold are not compressed; messages larger than the threshold are compressed.
CompressionType	Sets the type of compression to use on this connection. Refer to Section 10.4.3 for more information about supported compression types.
GuaranteedOutputBuffers	The guaranteed number of buffers which this **IChannel** can use while writing data. Each buffer can contain **MaxFragmentSize** bytes. Guaranteed output buffers are allocated at initialization time. For more details on obtaining a buffer, refer to Section 10.8. You can configure **GuaranteedOutputBuffers** using **IChannel.IOCtl**, as described in Section 10.14.6.
MaxFragmentSize	The maximum allowed buffer size which can be written to the network. If a larger buffer is required, the Transport will internally fragment the larger buffer into smaller buffers whose size is set to **MaxFragmentSize**. This is the largest size a user can request while still being 'packable.'

Table 51: ChannelInfo Methods

METHOD	DESCRIPTION
MaxOutputBuffers	The maximum number of output buffers which this **IChannel** can use. (**MaxOutputBuffers** - **GuaranteedOutputBuffers**) is equal to the number of shared pool buffers that this **IChannel** can use. Shared pool buffers are only used if all **GuaranteedOutputBuffers** are unavailable. If **MaxOutputBuffers** is equal to the **GuaranteedOutputBuffers** value, shared pool buffers are unavailable. You can configure **MaxOutputBuffers** using **IChannel.IOCtl**, as described in Section 10.14.6.
NumInputBuffers	Gets the number of sequential input buffers into which the **IChannel** reads data. This controls the maximum number of bytes that can be handled with a single network read operation on each channel. Each input buffer can contain **MaxFragmentSize** bytes. Input buffers are allocated at initialization time.
PingTimeout	Gets the negotiated ping timeout value. Typically, the rule of thumb in handling heartbeats is to send a heartbeat every **PingTimeout**/3 seconds. For more details on **PingTimeout**, refer to Section 10.12.1.
Port	Gets the server port number to which the consumer or non-interactive provider application connects.
PriorityFlushStrategy	The current priority level order used when flushing buffers to the connection, where **H** = High priority, **M** = Medium priority, and **L** = Low priority. Allows for up to 32 one-byte characters to be represented. When passed to **IChannel.Write**, each buffer is associated with the priority level at which it should be written. The default **PriorityFlushStrategy** writes buffers in the order: High, Medium, High, Low, High, Medium. This provides a slight advantage to the medium-priority level and a greater advantage to high-priority data. Data order is preserved within each priority level and if all buffers are written with the same priority, the order of data does not change. You can configure **PriorityFlushStrategy** using **IChannel.IOCtl**, as described in Section 10.14.6.
ServerToClientPings	Gets whether server is expected to send heartbeat messages: • If set to **true**, heartbeat messages must flow from server to client. • If set to **false**, the server is not required to send heartbeats. LSEG Real-Time Distribution System and other LSEG components typically require this value to be set to **true**.
SysRecvBufSize	Gets the size of the receive or input buffer associated with the underlying transport. The Transport has an additional input buffer controlled by **NumInputBuffers**. For some connection types, you can configure **SysRecvBufSize** using **IChannel.IOCtl**, as described in Section 10.14.6.
SysSendBufSize	Gets the size of the send or output buffer associated with the underlying transport. The Transport has additional output buffers, controlled by **MaxOutputBuffers** and **GuaranteedOutputBuffers**. For some connection types, you can configure **SysSendBufSize** using **IChannel.IOCtl**, as described in Section 10.14.6.

Table 51: ChannelInfo Methods (Continued)

10.14.3 ComponentInfo Method

METHOD	DESCRIPTION
ComponentVersion	A **Buffer** containing an ASCII string that indicates the product version of the connected component.

Table 52: componentInfo Methods

10.14.4 ServerInfo Methods

The following table describes values available to the user through the use of the **IServer.Info** method. This information is returned as part of the **ServerInfo** object.

METHOD	DESCRIPTION
CurrentBufferUsage	The number of currently used shared pool buffers across all users connected to the **IServer**.
PeakBufferUsage	The maximum achieved number of used shared pool buffers across all users connected to the **IServer**. This value can be reset through the use of **IServer.IOCtl**, as described in Section 10.14.7.

Table 53: ServerInfo Methods

10.14.5 IChannel.IOCtl IOCtlCodes

The following table provides a description of the available for use with the **IChannel.IOCtl** method.

OPTION ENUMERATION	DESCRIPTION
COMPRESSION_THRESHOLD	Allows an **IChannel** to change the size (in bytes) at which buffer compression occurs, must be greater than 30 bytes. Valueis an **int**. Default is 30 bytes for ZLIB and 300 bytes for LZ4.
HIGH_WATER_MARK	Allows an **IChannel** to change the internal output queue depth water mark, which has a default value of bytes. When the output queue exceeds this number of bytes, the **IChannel.Write** method internally attempts to flush content to the network. Value is an **int**.
MAX_NUM_BUFFERS	Allows an **IChannel** to change its **MaxOutputBuffers** setting. Value is an **int**.
NUM_GUARANTEED_BUFFERS	Allows an **IChannel** to change its **GuaranteedOutputBuffers** setting. Value is an **int**.
PRIORITY_FLUSH_ORDER	Allows an **IChannel** to change its **PriorityFlushStrategy**. Value, where each is either: • **H** for high priority • **M** for medium priority • **L** for low priority The should not exceed 32. At least one **H** and one **M** must be present, however no **L** is required. If low priority flushing is not specified, the low priority queue is flushed only when other data is not available for output.
SYSTEM_READ_BUFFERS	Allows an **IChannel** to change the TCP receive buffer size associated with the connection. Value
SYSTEM_WRITE_BUFFERS	Allows an **IChannel** to change the TCP send buffer size associated with the connection. Value.

Table 54: IChannel.IOCtl IOCtlCodes

10.14.6 IServer.IOCtl IOCtlCodes

The following table provides a description of the IOCtlCodes available for use with the **`IServer.IOCtl`** method.

IOCTLCODE	DESCRIPTION
SERVER_NUM_POOL_BUFFERS	Allows an **`IServer`** to change its **`SharedPoolSize`** setting. Value is an **`int.`**
SERVER_PEAK_BUF_RESET	Allows an **`IServer`** to reset the **`PeakBufferUsage`** statistic. Value is not required.

Table 55: `IServer.IOCtl` IOCtlCodes

10.15 Encrypted Connections

ETA supports encrypted TCP connections. To specify an encrypted connection, set **`ConnectionOption.ConnectionType`** to **`ConnectionType.ENCRYPTED`** on both client and server.

10.15.1 Server-Side Encryption

NOTE: To use server-side encryption, you must configure both client and server for encryption. If you configure the server for encryption but fail to configure the client, the connection will fail.

When **`BindOptions.ConnectionType`** is set to **`ConnectionType.ENCRYPTED`** (refer to Section 10.4.1.1), all connections between the server and its clients are encrypted. When configuring a server for encryption, you must obtain and install a server certificate and server private key (in **.PEM** format) obtained from a trusted certificate authority. You must maintain these files as appropriate.

WARNING! You must manage your certificate and private key with proper security controls.

NOTE: ETA does not support self-signed certificates.

10.15.2 Additional Encryption Options

You can configure various encryption features (such as which algorithms to use, which version of TLS to use, and etc.) using **`EncryptionOpts`**. For details, refer to Section 10.4.1.2.

You can specify TLS security protocols using **`EncryptionProtocolFlags`**.

For details on these options and their default settings, refer to Table 24.

10.16 HTTP Proxy Connections

ETA supports HTTP proxy tunneling for **`ConnectionType.SOCKET`** and **`ConnectionType.ENCRYPTED`** connection types.

For options you can use when configuring a proxy connection, refer to Section 10.3.2.4.

10.16.1 Proxy Authentication

You can configure some proxy servers to authenticate client applications before they pass through the proxy to their destination. The Enterprise Transport API supports the Basic authentication scheme.

Authentication schemes include:

- Establish the type of credentials an application must provide to the proxy server.
- Define how to encode the credentials required for authentication.
- Determine the “handshake” process during which messages are exchanged between the proxy and the application during the authentication process.

11 Data Package Detailed View

11.1 Concepts

The Data Package exposes a collection of data types that can combine in a variety of ways to assist with modeling user's data. These types are split into two categories:

- A ***Primitive Type*** represents simple, atomically updating information. Primitive types represent values like integers, dates, and ASCII string buffers (refer to Section 11.2).
- A ***Container Type*** models more intricate data representations than Enterprise Transport API primitive types and can manage dynamic content at a more granular level. Container types represent complex types like field identifier-value, name-value, or key-value pairs (refer to Section 11.3). The Enterprise Transport API offers several uniform (i.e., homogeneous) container types whose entries house the same type of data. Additionally, there are several non-uniform (i.e., heterogeneous) container types in which different entries can hold different types of data.

Primitive and Container types are also presented as a part of the **`DataTypes`** enumeration in the ranges:

- 0 to 127 are Primitive Types as described in Section 11.2.
- 128 to 255 are Container Types as described in Section 11.3.

11.2 Primitive Types

A primitive type represents some type of base, system information (such as integers, dates, or array values). If contained in a set of updating information, primitive types update atomically (incoming data replaces any previously held values). Primitive types support ranges from simple primitive types (e.g., an integer) to more complex primitive types (e.g., an array).

The **DataTypes** includes values that define the type of a primitive:

- Values between 0 and 63 are ***base primitive types***. Base primitive types support the full range of values allowed by the primitive type and are discussed in Table 56.

  When contained in a **FieldEntry** or **ElementEntry**, base primitive types can also represent a ***blank value***. A blank value indicates that no value is currently present and any previously stored or displayed primitive value should be cleared. When decoding any base primitive value, the interface method (See Table 56) returns **CodecReturnCode.BLANK_DATA**. To encode blank data into a **FieldEntry** or **ElementEntry**, refer to and .

- Values between 64 and 127 are ***set-defined primitive types***, which define fixed-length encodings for many of the base primitive types (e.g., **DataTypes.INT_1** is a one byte fixed-length encoding of **DataTypes.INT_1**). These types can be leveraged only within a Set Definition and encoded or decoded as part of a **FieldList** or **ElementList**. Only certain set-defined primitive types can represent blank values. For more details about set-defined primitive types, refer to Section 11.6.

The following table provides a brief description of each base primitive type, along with interface methods used for encoding and decoding. Several primitive types have a more detailed description following the table.

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION
DataTypes.UNKNOWN	None	Indicates that the type is unknown. **DataTypes.UNKNOWN** is valid only when decoding a Field List type and a dictionary look-up is required to determine the type. This type cannot be passed into encoding or decoding functions. **Encode Interface:** None **Decode Interface:** None
DataTypes.INT	Int[a]	A signed integer type. Can currently represent a value of up to 63 bits along with a one bit sign (positive or negative). **Encode Interface:** EncodeInt.Encode **Decode Interface:** DecodeInt.Decode
DataTypes.UINT	UInt[b]	An unsigned integer type. Can currently represent an unsigned value with precision of up to 64 bits. **Encode Interface:** EncodeUInt.Encode **Decode Interface:** DecodeUInt.Decode
DataTypes.FLOAT	Float	A four-byte, floating point type. Can represent the same range of values allowed with the **Float** type. Follows IEEE 754 specification. **Encode Interface:** EncodeFloat.Encode **Decode Interface:** DecodeFloat.Decode
DataTypes.DOUBLE	Double	An eight-byte, floating point type. Can represent the same range of values allowed with the **Double** type. Follows IEEE 754 specification. **Encode Interface:** EncodeDouble.Encode **Decode Interface:** DecodeDouble.Decode

Table 56: Enterprise Transport API Primitive Types

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION
DataTypes.REAL	Real[c]	An optimized Rssl Wire Format representation of a decimal or fractional value which typically requires less bytes on the wire than **`Float`** or **`Double`** types. The user specifies a value with a hint for converting to decimal or fractional representation. For more details on this type, refer to Section 11.2.2. **Encode Interface:** EncodeReal.Encode **Decode Interface:** DecodeReal.Decode
DataTypes.DATE	Date	Defines a date with month, day, and year values. For more details on this type, refer to Section 11.2.3. **Encode Interface:** EncodeDate.Encode **Decode Interface:** DecodeDate.Decode
DataTypes.TIME	Time	Defines a time with hour, minute, second, millisecond, microsecond, and nanosecond values. For more details on this type, refer to Section 11.2.4. **Encode Interface:** EncodeTime.Encode **Decode Interface:** DecodeTime.Decode
DataTypes.DATETIME	DateTime	Combined representation of date and time. Contains all members of **`DataTypes.DATE`** and **`DataTypes.TIME`**. For more details on this type, refer to Section 11.2.5. **Encode Interface:** EncodeDateTime.Encode **Decode Interface:** DecodeDateTime.Decode
DataTypes.QOS	Qos	Defines Quality of Service information such as data timeliness (e.g., real time) and rate (e.g., tick-by-tick). Allows a user to send Quality of Service information as part of the data payload. Similar information can also be conveyed using multiple message headers. For more details on this type, refer to Section 11.2.6. **Encode Interface:** EncodeQos.Encode **Decode Interface:** DecodeQos.Decode
DataTypes.STATE	State	Represents data and stream state information. Allows a user to send state information as part of data payload. Similar information can also be conveyed in several message headers. For more details on this type, refer to Section 11.2.7. **Encode Interface:** EncodeState.Encode **Decode Interface:** DecodeState.Decode
DataTypes.ENUM	Enum[d]	Represents an enumeration type, defined as an unsigned, two-byte value. Many times, this enumeration value is cross-referenced with an enumeration dictionary (e.g., **enumtype.def**) or a well-known, definition (e.g., those contained in **LSEG.Eta.Rdm**). **Encode Interface:** EncodeEnum.Encode **Decode Interface:** DecodeEnum.Decode
DataTypes.ARRAY	Array	The array type allows users to represent a simple base primitive type list (all primitive types except **`Array`**). The user can specify the base primitive type that an array carries and whether each is of a variable or fixed-length. Because the array is a primitive type, if any primitive value in the array updates, the entire array must be resent. For more details on this type, refer to Section 11.2.8. **Encode Interface:** Refer to . **Decode Interface:** Refer to .

Table 56: Enterprise Transport API Primitive Types (Continued)

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION
DataTypes.BUFFER	Buffer[e]	Represents a raw byte buffer type. Any semantics associated with the data in this buffer is provided from outside of the Enterprise Transport API, either via a field dictionary (e.g., **RDMFieldDictionary**) or a Domain Model Message definition. For more details on this type, refer to Section 11.2.9. **Encode Interface:** EncodeBuffer.Encode **Decode Interface:** DecodeBuffer.Decode
DataTypes.ASCII_STRING	Buffer[e]	Represents an ASCII string which should contain only characters that are valid in ASCII specification. Because this might be NULL terminated, use the provided length when accessing content. The Enterprise Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.9. **Encode Interface:** EncodeBuffer.Encode **Decode Interface:** DecodeBuffer.Decode
DataTypes.UTF8_STRING	Buffer[e]	Represents a UTF8 string which should follow the UTF8 encoding standard and contain only characters valid within that set. Because this might be NULL terminated, use the provided length when accessing content. The Enterprise Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.9. **Encode Interface:** EncodeBuffer.Encode **Decode Interface:** DecodeBuffer.Decode
DataTypes.RMTES_STRING	Buffer[e]	Represents an RMTES string which should follow the RMTES encoding standard and contain only characters valid within that set. For more details on this type, refer to Section 11.2.9. **Encode Interface:** EncodeBuffer.Encode **Decode Interface:** DecodeBuffer.Decode

Table 56: Enterprise Transport API Primitive Types (Continued)

a. This type allows a value ranging from (-2^{63}) to $(2^{63} - 1)$.
b. This type allows a value ranging from 0 up to $(2^{64} - 1)$.
c. This type allows a value ranging from (-2^{63}) to $(2^{63} - 1)$. This can be combined with hint values to add or remove up to seven trailing zeros, fourteen decimal places, or fractional denominators up to 256.
d. This type allows a value ranging from 0 to 65,535.
e. The Enterprise Transport API handles this type as opaque data, simply passing the length specified by the user and that number of bytes, no additional encoding or processing is done to any information contained in this type. Any specific encoding or decoding required for the information contained in this type is done outside of the scope of the Enterprise Transport API, before encoding or after decoding this type. This type allows for a length of up to 65,535 bytes.

11.2.1 DataTypes Methods

DataTypes contains the following methods:

MEMBER	DESCRIPTION
PrimitiveTypeSize	Returns the maximum encoded size for base and set-defined primitive types. If the type allows for content of varying length (e.g. **Array**, **Buffer**, etc.), a value of **255** is returned (though the maximum encoded length may exceed **255**).
IsPrimitiveType	• If the dataType represents a primitive type, returns **true**. • If the dataType represents a container type, returns **false**.
IsContainerType	• If the dataType represents a container type, returns **true**. • If the dataType represents a primitive type, returns **false**.
ToString	Returns a **String** representation for a **DataTypes** value.

Table 57: DataTypes Methods

11.2.2 Real

Real is a object that represents decimals or fractional values in a bandwidth-optimized format.

The **Real** preserves the precision of encoded numeric values by separating the numeric value from any decimal point or fractional denominator. Developers should note that in some conversion cases, there may be a loss of precision; this is an example of a narrowing precision conversion. Because the IEEE 754 specification (used for **float** and **double** types) cannot represent some values exactly, rounding (per the IEEE 754 specification) may occur when converting between **Real** representation and **float** or **double** representations, either using the provided helper methods or manually (using the conversion formulas provided). In cases where precision may be lost, converting to a string or using the provided string conversion helper as an intermediate point can help avoid the rounding precision loss.

11.2.2.1 Methods

Real contains the following members:

METHOD	DESCRIPTION
IsBlank	A Boolean value. Indicates whether data is considered blank. If true, other members should be ignored, if false other members determine the resultant value. This allows **Real** to be represented as blank when used as either a primitive type or a set-defined primitive type.
Hint	A hint enumeration value which defines how to interpret the value contained in **RealHints** values can add or remove up to seven trailing zeros, 14 decimal places, or fractional denominators up to 256. For more information about hint values, refer to Table 59.
ToLong	The raw value represented by the **Real** (omitting any decimal or denominator). Typically requires application of the **Hint** before interpreting or performing any calculations. This member can currently represent up to 63 bits and a one-bit sign (positive or negative).
ToDouble	Uses the formulas described in Section 11.2.2.3 to convert a **Real** to a **Double** type.
ToString	Converts a **Real** type to a numeric **String** representation. Blank is output as an empty zero length **String**.
Value(long, *hint*)	Sets the raw **long** value and **hint**.
Value(double, *hint*)	Uses the formulas described in Section 11.2.2.4 to convert a **double** and **hint** to a **Real** type.
Value(float, *hint*)	Uses the formulas described in Section 11.2.2.4 to convert a **float** and **hint** to a **Real** type.

Table 58: Real Methods

METHOD	DESCRIPTION
Value(String)	Converts a numeric **String** with denominator or decimal information to a **Real** type. Interprets a **String** of **+0** as a blank **Real**.
Encode	Encodes a **Real** into a buffer.
Decode	Decodes a **Real** from a buffer.
Equals	Compares one **Real** to another specified **Real**. Returns **true** if equal, **false** otherwise.
Copy	Performs a deep copy of one **Real** into another specified **Real**.
Blank	Clears the object and sets **IsBlank** to **true**.
Clear	Clears the object, so that you can reuse it. **IsBlank** is set to **false**.

Table 58: Real Methods

11.2.2.2 Hint Values

The following table defines the available **RealHints** values for use with **Real**. The conversion routines described in use **Real**'s **Hint** and **ToLong** value.

ENUM	DESCRIPTION
RealHints.EXPONENT_14	Negative exponent operation, equivalent to 10^{-14}. Shifts decimal by 14 positions.
RealHints.EXPONENT_13	Negative exponent operation, equivalent to 10^{-13}. Shifts decimal by 13 positions.
RealHints.EXPONENT_12	Negative exponent operation, equivalent to 10^{-12}. Shifts decimal by 12 positions.
RealHints.EXPONENT_11	Negative exponent operation, equivalent to 10^{-11}. Shifts decimal by 11 positions.
RealHints.EXPONENT_10	Negative exponent operation, equivalent to 10^{-10}. Shifts decimal by ten positions.
RealHints.EXPONENT_9	Negative exponent operation, equivalent to 10^{-9}. Shifts decimal by nine positions.
RealHints.EXPONENT_8	Negative exponent operation, equivalent to 10^{-8}. Shifts decimal by eight positions.
RealHints.EXPONENT_7	Negative exponent operation, equivalent to 10^{-7}. Shifts decimal by seven positions.
RealHints.EXPONENT_6	Negative exponent operation, equivalent to 10^{-6}. Shifts decimal by six positions.
RealHints.EXPONENT_5	Negative exponent operation, equivalent to 10^{-5}. Shifts decimal by five positions.
RealHints.EXPONENT_4	Negative exponent operation, equivalent to 10^{-4}. Shifts decimal by four positions.
RealHints.EXPONENT_3	Negative exponent operation, equivalent to 10^{-3}. Shifts decimal by three positions.
RealHints.EXPONENT_2	Negative exponent operation, equivalent to 10^{-2}. Shifts decimal by two positions.
RealHints.EXPONENT_1	Negative exponent operation, equivalent to 10^{-1}. Shifts decimal by one position.
RealHints.EXPONENT0	Exponent operation, equivalent to 10^{0}. **ToLong** does not change.
RealHints.EXPONENT1	Positive exponent operation, equivalent to 10^{1}. Depending on the type of conversion, this adds or removes one trailing zero.
RealHints.EXPONENT2	Positive exponent operation, equivalent to 10^{2}. Depending on the type of conversion, this adds or removes two trailing zeros.
RealHints.EXPONENT3	Positive exponent operation, equivalent to 10^{3}. Depending on the type of conversion, this adds or removes three trailing zeros.

Table 59: RealHints Enumeration Values

ENUM	DESCRIPTION
RealHints.EXPONENT4	Positive exponent operation, equivalent to 10^4. Depending on the type of conversion, this adds or removes four trailing zeros.
RealHints.EXPONENT5	Positive exponent operation, equivalent to 10^5. Depending on the type of conversion, this adds or removes five trailing zeros.
RealHints.EXPONENT6	Positive exponent operation, equivalent to 10^6. Depending on the type of conversion, this adds or removes six trailing zeros.
RealHints.EXPONENT7	Positive exponent operation, equivalent to 10^7. Depending on the type of conversion, this adds or removes seven trailing zeros.
RealHints.FRACTION_1	Fractional denominator operation, equivalent to 1/1. Value does not change.
RealHints.FRACTION_2	Fractional denominator operation, equivalent to 1/2. Depending on the type of conversion, this adds or removes a denominator of two.
RealHints.FRACTION_4	Fractional denominator operation, equivalent to 1/4. Depending on the type of conversion, this adds or removes a denominator of four.
RealHints.FRACTION_8	Fractional denominator operation, equivalent to 1/8. Depending on the type of conversion, this adds or removes a denominator of eight.
RealHints.FRACTION_16	Fractional denominator operation, equivalent to 1/16. Depending on the type of conversion, this adds or removes a denominator of 16.
RealHints.FRACTION_32	Fractional denominator operation, equivalent to 1/32. Depending on the type of conversion, this adds or removes a denominator of 32.
RealHints.FRACTION_64	Fractional denominator operation, equivalent to 1/64. Depending on the type of conversion, this adds or removes a denominator of 64.
RealHints.FRACTION_128	Fractional denominator operation, equivalent to 1/128. Depending on the type of conversion, this adds or removes a denominator of 128.
RealHints.FRACTION_256	Fractional denominator operation, equivalent to 1/256. Depending on the type of conversion, this adds or removes a denominator of 256.

Table 59: `RealHints` Enumeration Values (Continued)

11.2.2.3 Hint Use Case: Converting a Real to a Float or a Double

An application can convert between an **Real** and a **float** or **double** as needed. Converting an **Real** to a **double** or **float** is typically done to perform calculations or display data after receiving it.

The conversion process adds or removes decimal or denominator information from the value to optimize transmission sizes. In an **Real** type, the decimal or denominator information is indicated by the **Real.Hint**, and the **Real.ToLong** indicates the value (less any decimal or denominator). If the **Real.IsBlank** member is **true**, this is handled as blank regardless of information contained in the **Real.Hint** and **Real.ToLong** methods.

For this conversion, both the hint and its value are stored in the **Real** object. You can use the following example to perform this conversion, where **outputValue** is a system **float** or **double** to store output:

```
/* perform calculation and assign output to outputValue - may require appropriate float or double casts
 * depending on type of outputValue */
outputValue = real.ToDouble();
```

Code Example 16: Real Conversion to Double/Float

11.2.2.4 Hint Use Case: Converting Double or Float to a Real

To convert a **double** or **float** type to an **Real** type (typically done to prepare for transmission), the user must determine which hint value to use based on the type of value used:

- When converting a decimal value, the chosen hint value must be less than **RealHints.FRACTION_1**.
- When converting a fractional value, the chosen hint value must be greater than or equal to **RealHints.FRACTION_1**.

You can use the following example to perform the conversion, where **inputValue** is the unmodified input **float** or **double** value and **inputHint** is the hint chosen by the user:

```
/* Perform calculation and store output in Real object - may require appropriate float or double casts
   depending on type of inputValue */
real.Value(inputValue, inputHint);
```

Code Example 17: Real Conversion from Double/Float

11.2.3 Date

Date represents the date (i.e., **Day**, **Month**, and **Year**) in a bandwidth-optimized fashion.

11.2.3.1 Date Methods

Date represents the date (i.e., **Day**, **Month**, and **Year**) in a bandwidth-optimized fashion.

If **Day**, **Month**, and **Year** are all set to **0** the **Date** is blank. If any individual member is represented as a blank value (**0**), only that member is blank. This is useful for representing dates which specify **Month** and **Year**, but not **Day**. The **Date** type can be represented as blank when used as a primitive type and a set-defined primitive type.

METHOD	DESCRIPTION
Day	Sets or gets the **Day**. Represents the day of the month, where **0** indicates a blank entry. **Day** allows a range of **0** to **255**, though the value typically does not exceed **31**.
Month	Sets or gets the **Month**. Represents the month of the year, where **0** indicates a blank entry. **Month** allows a range of **0** to **255**, though the value typically does not exceed **12**.
Year	Sets or gets the **Year**.Represents the year, where **0** indicates a blank entry. You can use this member to specify a two- or four-digit year (where specific usage is indicated outside of the Enterprise Transport API). **Year** allows a range of **0** to **65,535**.
Blank	Sets all members in **Date** to **0**. Because **0** represents a blank date value, this performs the same functionality as the **Date.Clear** method.
IsBlank	Returns **true** if **Date** is blank, otherwise **false**.
IsValid	Verifies the contents of the **Date** object. Determines whether the specified **Day** is valid within the specified **Month** (e.g., a day greater than 31 is considered invalid for any month). This method uses the **Year** value to determine leap year validity of day numbers for February. If **Date** is blank or valid, **true** is returned; **false** otherwise.
ToString	Converts the **Date** to a **String** using the "***DD MM YYYY***" format.
Value	Converts a **String** date to **Date** from one of the following formats: • "***DD MMM YYYY***" (e.g., **30 JAN 2018**) • "***MM/DD/YYYY***" (e.g., **01/30/2018**)
Equals	Compares the **Date** to another specified **Date**. Returns **true** if equal, **false** otherwise.
Copy	Performs a deep copy of the **Date** to another specified **Date**.
Encode	Encodes a **Date** into a buffer.
Decode	Decodes a **Date** from a buffer.
Clear	Clears the object for reuse. Because **0** represents a blank date value, this performs the same functionality as the **Date.Blank** method.

Table 60: Date Methods

11.2.4 Time

Time represents time (hour, minute, second, millisecond, microsecond, and nanosecond) in a bandwidth-optimized fashion. This type is represented as Greenwich Mean Time (GMT) unless noted otherwise[1].

11.2.4.1 Time Methods

If all methods are set to their respective blank values, **Time** is blank. If any individual member is set to a blank value, only that member is blank. This is useful for representing times without **Second**, **Millisecond**, **Microsecond**, or **Nanosecond** values. The **Time** type can be represented as blank when it is used as a primitive type and a set-defined primitive type.

METHOD	DESCRIPTION
Hour	Sets or gets the hour of the day (**hour**). Represents the hour of the day using a range of **0** to **255** (**255** represents a blank hour value), though the value does not typically exceed **23**.
Minute	Sets or gets the minute of the hour (**minute**). Represents the minute of the hour using a range of **0** to **255** (**255** represents a blank minute value), though the value does not typically exceed **59**.
Second	Sets or gets the second of the minute (**second**). Represents the second of the minute using a range of **0** to **255** (**255** represents a blank second value), though the value does not typically exceed **59**.
Millisecond	Sets or gets the millisecond of the second (**millisecond**). Represents the millisecond of the second using a range of **0** - **65,535** (**65535** represents a blank millisecond value), though the value does not typically exceed **999**.
Microsecond	Sets or gets the microsecond of the millisecond (**microsecond**). Represents the microsecond of the millisecond using a range of **0** - **2047** (**2047** represents a blank microsecond value), though the value does not typically exceed **999**.
Nanosecond	Sets or gets the nanosecond of the microsecond (**nanosecond**). Represents the nanosecond of the microsecond using a range of **0** - **2047** (where **2047** represents a blank nanosecond value), though the value does not typically exceed **999**.
Blank	Sets all members in **Time** to their blank values.
IsBlank	Returns **true** if all members in **Time** are set to their blank values.
IsValid	Verifies the contents of a populated Time structure. Validates the ranges of the **Hour**, **Minute**, **Second**, **Millisecond**, **Microsecond**, and **Nanosecond** members. If **Time** is blank or valid, **true** is returned; **false** otherwise.
ToString	Converts **Time** to a **String** using the "***hour:minute:second:milli:micro:nano***" format (e.g., ***15:24:54:627:843:143***).
Value	Converts a **String** time to **Time** from one of the following formats: • "***HH*:*MM***" (e.g., **13:01**) • "***HH*:*MM*:*SS***" (e.g., **15:23:54**)
Equals	Compares **Time** to another specified **Time**. If equal, returns **true**; **false** otherwise.
Copy	Performs a deep copy of **Time** to another specified **Time**.
Encode	Encodes a **Time** into a buffer.
Decode	Decodes a **Time** from a buffer.
Clear	Clears the object for reuse.

Table 61: Time Methods

1. The provider's documentation should indicate whether the providing application provides times in another representation.

11.2.5 DateTime

DateTime represents the date (**Date**) and time (**Time**) in a bandwidth-optimized fashion. This time value is represented as Greenwich Mean Time (GMT) unless noted otherwise[2].

11.2.5.1 DateTime Methods

If **Date** and **Time** values are set to their respective blank values, **DateTime** is blank. If any individual member is set to a blank value, only that member is blank. The **DateTime** type can be represented as blank when it is used as a primitive type and a set-defined primitive type.

DateTime contains the following methods:

METHOD	DESCRIPTION
Date	Returns the **Date** portion of the **DateTime** and conforms to the behaviors described in Section 11.2.3.
Time	Returns the **Time** portion of the **DateTime** and conforms to the behaviors described in Section 11.2.4.
Day	Sets or gets the day of the month. The valid range is **0** to **255**, where **0** indicates a blank entry (though the value does not typically exceed **31**).
Month	Sets or gets the month of the year. The valid range is **0** to **255**, where **0** indicates a blank entry (though the value does not typically exceed **12**).
Year	Sets or gets the year. You can use this member to specify a two- or four-digit year (where specific usage is indicated outside of the Enterprise Transport API). The valid range is **0** to **65,535**, where **0** indicates a blank entry.
Hour	Sets or gets the hour of the day. The valid range is **0** to **255**, where **255** represents a blank hour value (though the value does not typically exceed **23**).
Minute	Sets or gets the minute of the hour. The valid range is **0** to **255**, where **255** represents a blank minute value (though the value does not typically exceed **59**).
Second	Sets or gets the second of the minute. The valid range is **0** to **255**, where **255** represents a blank second value (though the value does not typically exceed **59**).
Millisecond	Sets or gets the millisecond of the second. The valid range is **0** to **65535**, where **65535** represents a blank millisecond value (though the value does not typically exceed **999**.
Microsecond	Sets or gets the microsecond of the millisecond. The valid range is **0** to **2047**, where **2047** represents a blank microsecond value (though the value does not typically exceed **999**).
Nanosecond	Sets or gets the nanosecond of the microsecond. The valid range is **0** to **2047**, where **2047** represents a blank nanosecond value (though the value does not typically exceed **999**).
GmtTime	Sets the date time to the present time in GMT zone.
LocalTime	Sets the date time to the present time in the local time zone.
Blank	Sets all members in **DateTime** to their respective blank values.
IsBlank	Returns **true** if all members in **Date** and **Time** are set to the values used to signify blank.
IsValid	Determines whether **Day** is valid for the specified **Month** (e.g., a **Day** greater than 31 is considered invalid for any **Month**) as determined by the specified **Year** (to calculate whether it is a leap year). Also validates the range of **Hour**, **Minute**, **Second**, **Millisecond**, **Microsecond**, and **Nanosecond** members. If **DateTime** is blank or valid, **true** is returned; **false** otherwise.

Table 62: DateTime Methods

2. The provider's documentation should indicate whether the providing application provides times in another representation.

METHOD	DESCRIPTION
MillisSinceEpoch	Returns the date-time value as milliseconds since the January 1, 1970 (midnight UTC/GMT) epoch.
ToString	Converts **DateTime** to a **String** which the string will be "***%d %b %Yhour:minute:second:milli:micro:nano***" (e.g., ***30 JAN 2018 15:24:54:627:843:143***).
Value(String)	Converts a **String** representation of a date and time to a **DateTime**. This method supports: • **Date** values conforming to "***%d %b %Y***" format (e.g., **30 NOV 2010**) or "***%m/%d/%y***" format (e.g., **11/30/2010**). • **Time** values conforming to "***%H:%M***" format (e.g., **15:24**), "***%H:%M:%S***" format (e.g., **15:24:54**), or "***hour:minute:second:milli:micro:nano***" format (e.g., **15:24:54:627:843:143**).
Value(long)	Sets date-time using a number equal to milliseconds since the January 1, 1970 (midnight UTC/GMT) epoch.
Equals	Compares two **DateTime** structures. Returns **true** if equal; **false** otherwise.
Copy	Performs a deep copy of **DateTime** to another specified **DateTime**.
Encode	Encodes a date and time into a buffer.
Decode	Decodes a date and time from a buffer.
Clear	Clears this object, so that you can reuse it. Sets all members to **0**.

Table 62: DateTime Methods (Continued)

11.2.6 Qos

Qos classifies data into two attributes:

- ***Timeliness***: Conveys the age of data.
- ***Rate***: Conveys the rate at which data changes.

Some timeliness or rate values allow you to provide additional time or rate data, for more details refer to Section 11.2.6.1, Section 11.2.6.2, and Section 11.2.6.3.

If present in a data payload, specific handling and interpretation associated with Quality of Service information is provided from outside of the Enterprise Transport API, possibly via the specific Domain Message Model definition.

Several Enterprise Transport API message headers also contain Quality of Service data. When present, this data is typically used to request or convey the Quality of Service associated with a particular stream. For more information about Quality of Service use within a message, refer to Section 12.2.1 and Section 12.2.2. When conflated data is sent, additional conflation data might be included with update messages. For further details on conflation, refer to Section 12.2.3.

11.2.6.1 Qos Methods

Qos contains the following methods:

METHOD	DESCRIPTION
Timeliness	Sets or gets the **Timeliness**. Describes the age of the data (e.g., real time). Timeliness values are described in Section 11.2.6.2.
Rate	Sets or gets the **Rate**. Describes the rate at which the data changes (e.g., tick-by-tick). Rate values are described in Section 11.2.6.3.

Table 63: Qos Methods

METHOD	DESCRIPTION
Dynamic	Describes the changeability of the Quality of Service within the requested range, typically over the life of a data stream. • If set to **false**, the Quality of Service should not change following the initial establishment. • If set to **true**, the Quality of Service can change over time to other values within the requested range. Quality of Service can change due to permissioning information, stream availability, network congestion, or other reasons. Specific information about dynamically changing Quality of Service should be described in documentation for components that support this behavior.
IsDynamic	Returns **true** if the Quality of Service is dynamic. Describes the changeability of the quality of service, typically over the life of a data stream.
TimeInfo	Sets or gets the **`TimeInfo`**. Conveys detailed information about data **`Timeliness`**, typically the amount of time delay. **`TimeInfo`** allows for a range of **0** to **65,535**. This information is present only when **`Timeliness`** is set to **QosTimeliness.DELAYED**.
RateInfo	Sets or gets the **`RateInfo`**. Conveys detailed information about **`Rate`**, typically the interval of time during which data are conflated. ***Conflation*** combines multiple information updates into a single update, usually reducing network traffic. **`RateInfo`** allows for a range of **0** to **65,535**. This information is present only when **`Rate`** is set to **QosRates.TIME_CONFLATED**.
Equals	Compares this **`Qos`** with a specified **`Qos`**. • Returns **true** if the values contained in the structure are identical. • Returns **false** if the values contained in the structure differ.
IsBetter	Compares this **`Qos`** with a specified **`Qos`** to determine which has better overall quality. • Returns **true** if this **`Qos`** is better. • Returns **false** if this **`Qos`** is not better.
IsInRange	Determines whether this **`Qos`** lies within a range from best **`Qos`** to worst **`Qos`**. • Returns **true** if this **`Qos`** falls between best and worst **`Qos`** • Returns **false** if this **`Qos`** falls outside of the best or worst **`Qos`** range.
Blank	Clears this object and sets it to blank.
IsBlank	Returns **true** if **`Qos`** is blank, otherwise **false**.
ToString	Returns a String representation for this **`Qos`**.
Copy	Performs a deep copy of the **`Qos`** to another specified **`Qos`**.
Encode	Encodes **`Qos`** into a buffer.
Decode	Decodes **`Qos`** from a buffer.
Clear	Clears this object, so that you can reuse it. Sets all members in **`Qos`** to an initial value of **0**. This includes setting **`Rate`** and **`Timeliness`** to their unspecified values (not intended to be encoded or decoded).

Table 63: `Qos` Methods (Continued)

11.2.6.2 Qos Timeliness Values

QOS TIMELINESS	DESCRIPTION
QosTimeliness.UNSPECIFIED	**Timeliness** is unspecified. Typically used by Quality of Service initialization methods and not intended to be encoded or decoded.
QosTimeliness.REALTIME	**Timeliness** is real time: data is updated as soon as new data is available. This is the highest-quality **Timeliness** value. In conjunction with a **Rate** of **QosRates.TICK_BY_TICK**, real time is the best overall Quality of Service.
QosTimeliness.DELAYED_UNKNOWN	**Timeliness** is delayed, though the amount of delay is unknown. This is a lower quality than **QosTimeliness.REALTIME** and might be worse than **QosTimeliness.DELAYED** (in which case the delay is known).
QosTimeliness.DELAYED	**Timeliness** is delayed and the amount of delay is provided in **Qos.TimeInfo**. This is lower quality than **QosTimeliness.REALTIME** and might be better than **QosTimeliness.DELAYED_UNKNOWN**.

Table 64: QosTimeliness Values

11.2.6.3 QosRates Values

QOS RATE	DESCRIPTION
QosRates.UNSPECIFIED	**Rate** is unspecified. Typically used by Quality of Service initialization methods and not intended to be encoded or decoded.
QosRates.TICK_BY_TICK	**Rate** is tick-by-tick (i.e., data is sent for every update). This is the highest quality **Rate** value. The best overall Quality of Service is a tick-by-tick **Rate** with a **Timeliness** of **QosTimeliness.REALTIME**.
QosRates.JIT_CONFLATED	**Rate** is ***Just-In-Time (JIT) Conflated***, meaning that quality is typically tick-by-tick, but if a data burst occurs (or if a component cannot keep up with tick-by-tick delivery), multiple updates are combined into a single update to reduce traffic. This value is usually considered a lower quality than **QosRates.TICK_BY_TICK**. Because JIT conflation is triggered by an application's inability to keep up with data rates, the effective rate depends on whether the application can sustain full data rates. Use of this value typically results in a rate similar to **QosRates.TICK_BY_TICK**. However, when the application cannot keep up with data rates, it results in a rate similar to **QosRates.TIME_CONFLATED**, where **RateInfo** is determined by the provider. Specific information about **ConflationTime** or **ConflationCount** might be present in an **IUpdateMsg**. For further details, refer to Section 12.2.3.
QosRates.TIME_CONFLATED	**Rate** is time-conflated. The interval of time (usually in milliseconds) over which data are conflated is provided in **Qos.RateInfo**. This is lower quality than **QosRates.TICK_BY_TICK** and at times even lower than **QosRates.JIT_CONFLATED**. Specific information about the **ConflationTime** or **ConflationCount** might be present in the **IUpdateMsg**. For more details, refer to Section 12.2.3.

Table 65: QosRates Values

11.2.7 State

State conveys data and stream health information. When present in a header, **State** applies to the state of the stream and data. When present in a data payload, the meaning of **State** should be defined by the Domain Message Model.

Several Enterprise Transport API message headers also contain **State** data. When present in a message header, **State** typically conveys the overall data and stream health of messages flowing over a particular stream. For more information on using **State** in a message, refer to Section 12.2.1, Section 12.2.2, and Section 12.2.4. A decision table that provides example behaviors for various state combinations is available in Appendix A, Item and Group State Decision Table.

11.2.7.1 Methods

State contains the following methods:

METHOD	DESCRIPTION
StreamState	Sets or gets the **StreamState**, which conveys data about the stream's health. **StreamState** values are described in Section 11.2.7.2.
DataState	Sets or gets the **DataState**, which conveys data about the health of data flowing within a stream. **DataState** values are described in Section 11.2.7.4.
Code	Sets or gets the **Code**, which is a value that conveys additional information about the current state. Typically indicates more specific information (e.g., pertaining to a condition occurring upstream causing current data and stream states). **Code** is typically used for informational purposes. **StateCode** values are described in Section 11.2.7.6.
	NOTE: An application should not trigger specific behavior based on this content.
Text	Sets or gets the **Text**, which is a Buffer containing specific text regarding the current data and stream state. Typically used for informational purposes. Encoded **Text** has a maximum allowed length of 32,767 bytes.
	NOTE: An application should not trigger specific behavior based on this content.
Equals	Compares the **State** with another specified **State**. • Returns **true** if the values contained in the structure are identical. • Returns **false** if the values contained in the structure differ.
IsBlank	Returns **true** if **State** is blank, otherwise **false**.
IsFinal	• Returns **true** if the **State** represents a final state for a stream (i.e., stream is Closed, Closed Recover, Redirected, or NonStreaming). • Returns **false** if the **State** is not final.
ToString	Returns a Java **String** representing this **State**, including **StreamState**, **DataState**, **Code** and **Text**.
Copy	Perform a deep copy of the **State** to another specified **State**.
Encode	Encodes **State** into a buffer.
Decode	Decodes **State** into a buffer.
Clear	Clears this object for reuse. Sets all members in **State** to an initial value. This includes setting **StreamState** to its unspecified value (not intended to be encoded or decoded).

Table 66: State Methods

11.2.7.2 StreamStates Values

STREAM STATE	DESCRIPTION
StreamStates.UNSPECIFIED	**StreamState** is unspecified. Typically used as a structure initialization value and is not intended to be encoded or decoded.
StreamStates.OPEN	**StreamState** is open. This typically means that data is streaming: as data changes, they are sent on the stream.
StreamStates.NON_STREAMING	**StreamState** is non-streaming. After receiving a final **IRefreshMsg** or **IStatusMsg**, the stream is closed and updated data is not delivered without a subsequent re-request. Update messages might still be received between the first and final part of a multi-part refresh. For further details, refer to Section 13.1.
StreamStates.CLOSED_RECOVER	**StreamState** is closed, however data can be recovered on this service and connection at a later time. This state can occur via either a **IRefreshMsg** or an **IStatusMsg**. Single Open behavior can modify this state (continuing to indicate a stream state of **StreamStates.OPEN**) and attempt to recover data on the user's behalf. For further details on Single Open behavior, refer to Section 13.5.
StreamStates.CLOSED	**StreamState** is closed. Data is not available on this service and connection and is not likely to become available, though the data might be available on another service or connection. This state can result from either an **IRefreshMsg** or an **IStatusMsg**.
StreamStates.REDIRECTED	**StreamState** is redirected. The current stream is closed and has new identifying information. The user can issue a new request for the data using the new message key data from the redirect message. This state can result from either an **IRefreshMsg** or an **IStatusMsg**. For further details, refer to Section 12.1.3.2.

Table 67: StreamStates Values

11.2.7.3 StreamStates Methods

StreamStates includes the following methods:

METHOD	DESCRIPTION
Info	Returns a **String** representation of any information associated with a **StreamStates** value (e.g. "Closed, Recoverable" for **StreamStates.CLOSED_RECOVER**).
ToString	Returns a **String** representation for a **StreamStates** value (e.g. "CLOSED_RECOVER" for **StreamStates.CLOSED_RECOVER**).

Table 68: StreamStates Methods

11.2.7.4 DataStates Values

DATA STATE	DESCRIPTION
DataStates.NO_CHANGE	Indicates there is no change in the current state of the data. When available, it is preferable to send more concrete state information (such as **OK** or **SUSPECT**) instead of **NO_CHANGE**. This typically conveys **Code** or **Text** information associated with an item group, but no change to the group's previous data and stream state has occurred.
DataStates.OK	**DataState** is **OK**. All data associated with the stream is healthy and current.
DataStates.SUSPECT	**DataState** is **SUSPECT** (also known as a stale-data state). A suspect data state means some or all of the data on a stream is out-of-date (or that it cannot be confirmed as current, e.g., the service is down). If an application does not allow suspect data, a stream might change from open to closed or closed recover as a result. For further details, refer to Section 13.5.

Table 69: DataStates Values

11.2.7.5 DataStates Methods

DataStates includes the following methods:

METHOD	DESCRIPTION
Info	Returns a **String** representation of any information associated with a **DataStates** value (e.g. "NoChange" for **DataStates.NO_CHANGE**).
ToString	Returns a **String** representation for a **DataStates** value (e.g. "NO_CHANGE" for **DataStates.NO_CHANGE**).

Table 70: DataStates Methods

11.2.7.6 StateCodes Values

STATE CODE	DESCRIPTION
StateCodes.ALREADY_OPEN	Indicates that a stream is already open on the connection for the requested data.
StateCodes.APP_AUTHORIZATION_FAILED	Indicates that application authorization using the secure token has failed.
StateCodes.DACS_DOWN	Indicates that the connection to the Data Access Control System is down and users are not allowed to connect.
StateCodes.DACS_MAX_LOGINS_REACHED	Indicates that the maximum number of logins has been reached.
StateCodes.DACS_USER_ACCESS_TO_APP_DENIED	Indicates that the application is denied access to the system.
StateCodes.ERROR	Indicates an internal error from the sender.
StateCodes.EXCEEDED_MAX_MOUNTS_PER_USER	Indicates that the login was rejected because the user exceeded their maximum number of allowed mounts.
StateCodes.FAILOVER_COMPLETED	Indicates that recovery from a failover condition has finished.

Table 71: StateCodes Values

STATE CODE	DESCRIPTION
StateCodes.FAILOVER_STARTED	Indicates that a component is recovering due to a failover condition. User is notified when recovery finishes via **StateCodes.FAILOVER_COMPLETED**.
StateCodes.FULL_VIEW_PROVIDED	Indicates that the full view (e.g., all available fields) is being provided, even though only a specific view was requested. discusses views in more detail.
StateCodes.GAP_DETECTED	Indicates that a gap was detected between messages. A gap might be detected via an external reliability mechanism (e.g., transport) or using the **SeqNum** present in Enterprise Transport API messages.
StateCodes.GAP_FILL	Indicates that the received content is meant to fill a recognized gap.
StateCodes.INVALID_ARGUMENT	Indicates that the request includes an invalid or unrecognized parameter. Specific information should be contained in the **Text**.
StateCodes.INVALID_VIEW	Indicates that the requested view is invalid, possibly due to bad formatting. Additional information should be available in the **Text**. discusses views in more detail.
StateCodes.JIT_CONFLATION_STARTED	Indicates that JIT conflation has started on the stream. User is notified when JIT Conflation ends via **StateCodes.REALTIME_RESUMED**.
StateCodes.NO_BATCH_VIEW_SUPPORT_IN_REQ	Indicates that the provider does not support batch and/or view functionality.
StateCodes.NO_RESOURCES	Indicates that no resources are available to accommodate the stream.
StateCodes.NON_UPDATING_ITEM	Indicates that a streaming request was made for non-updating data.
StateCodes.NONE	Indicates that additional state code information is not required, nor present.
StateCodes.NOT_ENTITLED	Indicates that the request was denied due to permissioning. Typically indicates that the requesting user does not have permission to request on the service, to receive requested data, or to receive data at the requested Quality of Service.
StateCodes.NOT_FOUND	Indicates that requested information was not found, though it might be available at a later time or through changing some parameters used in the request.
StateCodes.NOT_OPEN	Indicates that the stream was not opened. Additional information should be available in the **Text**.
StateCodes.PREEMPTED	Indicates the stream was preempted, possibly by a caching device. Typically indicates the user has exceeded an item limit, whether specific to the user or a component in the system. Relevant information should be contained in the **Text**.
StateCodes.REALTIME_RESUMED	Indicates that JIT conflation on the stream has finished.
StateCodes.SOURCE_UNKNOWN	Indicates that the requested service is not known, though the service might be available at a later point in time.
StateCodes.TIMEOUT	Indicates that a timeout occurred somewhere in the system while processing requested data.

Table 71: StateCodes Values(Continued)

STATE CODE	DESCRIPTION
StateCodes.TOO_MANY_ITEMS	Indicates that a request cannot be processed because too many other streams are already open.
StateCodes.UNABLE_TO_REQUEST_AS_BATCH	Indicates that a batch request cannot be used for this request. The user can instead split the batched items into individual requests. discusses batch requesting in more detail.
StateCodes.UNSUPPORTED_VIEW_TYPE	Indicates that the domain on which a request is made does not support the requested **ViewType**. discusses views in more detail.
StateCodes.USAGE_ERROR	Indicates invalid usage within the system. Specific information should be contained in the **Text**.
StateCodes.USER_UNKNOWN_TO_PERM_SYS	Indicates that the user is unknown to the permissioning system and is not allowed to connect.

Table 71: StateCodes Values(Continued)

11.2.7.7 StateCodes Methods

StateCodes includes the following methods:

METHOD	DESCRIPTION
Info	Returns a **String** representation of any information associated with a **StateCodes** value (e.g. “Nonupdating item” for **StateCodes.NON_UPDATING_ITEM**).
ToString	Returns a **String** representation for a **StateCodes** value (e.g. “NON_UPDATING_ITEM” for **StateCodes.NON_UPDATING_ITEM**).

Table 72: StateCodes Methods

11.2.8 Array

The **Array** is a uniform primitive type that can contain multiple simple primitive entries. An **Array** can contain zero to *N* primitive type entries[3], where zero entries indicates an empty **Array**.

Each **Array** can house only simple primitive types such as **Int**, **Real**, or **Date**. An **Array** cannot house any container types or other **Array** types. This is a uniform type, where **Array.PrimitiveType** method indicates the single, simple primitive type of each entry. **Array** uses simple replacement rules for change management. When new entries are added, or any array entry requires a modification, all entries must be sent with the **Array**. This new **Array** entirely replaces any previously stored or displayed data.

An **Array** can be encoded from pre-encoded data or by encoding individual pieces of data as provided. When encoding, the application passes the primitive type (when data is not encoded) or an **Buffer** (containing the pre-encoded primitive).

When decoding, the encoded content of the **ArrayEntry** is available as an **Buffer** by calling the **ArrayEntry.EncodedData** method. Further decoding of the entry’s content can be skipped by invoking the entry decoder to move to the next **ArrayEntry** or the contents can be further decoded by invoking the specifically contained type’s primitive decode function (refer to Section 11.2).

NOTE: Although it can house other primitive types, **Array** is itself considered a primitive type and can be represented as a blank value.

3. An **Array** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of an **Array** entry is bound by the maximum encoded length of the primitive types being contained. These limitations can change in subsequent releases.

11.2.8.1 Array Methods

METHOD	DESCRIPTION
PrimitiveType	Using **DataTypes**, **PrimitiveType** describes the base primitive type of each entry. **Array** can only contain simple primitive types and cannot house container types or other **Array**s.
ItemLength[a]	Sets the expected length of all array entries. • If set to **0**, entries are variable length and each encoded entry can have a different length. • If set to a non-zero value, each entry must be the specified length (e.g. sending **PrimitiveType** of **DataTypes.ASCII_STRING** with **ItemLength** set to **3** indicates that each array entry will be a fixed-length three-byte string). When using a fixed length, the application still passes in the base primitive type when encoding (e.g., if encoding fixed length **DataTypes.INT** types, an **Int** is passed in regardless of **ItemLength**). When encoding buffer types as fixed length: • Any content that exceeds **ItemLength** will be truncated. • Any content that is shorter than **ItemLength** will be padded with the **\0** (NULL) character.
EncodedData	Returns a **Buffer** that contains all encoded primitive types in the contents (if any). This refers to encoded **Array** payload and length information. The length information is available via the **Buffer.Length** method.
EncodeInit	Begins encoding an **Array**. This method expects that the members **Array.PrimitiveType** and **Array.ItemLength** are properly populated. The **EncodeIterator** specifies the **ITransportBuffer** or **Buffer** into which it encodes data. Entries can be encoded after this method returns.
EncodeComplete	Completes encoding of an **Array**. This function expects the same **EncodeIterator** used with **Array.EncodeInit** and **Array.EncodeEntry**. Set the **bool** parameter to: • **true** if encoding was successful, to finish encoding. • **false** if encoding of any entry failed, to roll back encoding to the last successfully-encoded point in the contents. All entries should be encoded before calling **Array.EncodeComplete**.
Decode	Begins decoding an **Array**. This method decodes from the **ITransportBuffer** or **Buffer** specified to the **DecodeIterator**.
IsBlank	Returns **true** if **Array** is blank, otherwise **false**.
Clear	Clears this object, so that you can reuse it. Sets all members to an initial value. **TIP:** When decoding, the **Array** object can be reused without using **Clear**.

Table 73: Array Methods

a. Only specific types are allowed as fixed-length encodings. **DataTypes.INT** and **DataTypes.UINT** can support one-, two-, four-, or eight-byte fixed lengths. **DataTypes.TIME** supports three- or five-byte fixed lengths. **DataTypes.DATETIME** supports seven- or nine-byte fixed lengths. **DataTypes.ENUM** supports one- or two-byte fixed lengths. **DataTypes.BUFFER**, **DataTypes.ASCII_STRING**, **DataTypes.UTF8_STRING**, and **DataTypes.RMTES_STRING** support any legal length value; see those types for allowable lengths.

11.2.8.2 ArrayEntry Methods

ArrayEntry includes the following methods:

METHOD	DESCRIPTION
EncodedData	• When encoding, this method specifies pre-encoded data for an **ArrayEntry**. Populate a **Buffer** with pre-encoded data, then call this method with the **Buffer**. • When decoding, the decode method will populate a **Buffer** with the encoded primitive type (if any). Call this method without a parameter to return the **Buffer** containing the encoded primitive type.
EncodeBlank	Encodes a blank entry.
Encode	Encodes an **ArrayEntry**. This method expects the same **EncodeIterator** used with **Array.EncodeInit**. • If encoding from pre-encoded data, specify the **Buffer** populated with pre-encoded data. • If encoding from a primitive type, specify the primitive type. (e.g. **UInt**). This method should be called for each entry being encoded. The specified type must match the **Array.PrimitiveType**.
Decode	Decodes an **ArrayEntry** and populates an internal **Buffer** (available via **EncodedData** method) with encoded entry contents. This method expects the same **DecodeIterator** used with **Array.Decode**. Any contained primitive type's decode method can be called based on **Array.PrimitiveType** (e.g. **UInt.Decode**) (refer to Section 11.2). Calling **ArrayEntry.Decode** again will decode and provide the next entry in the **Array** until no more entries are available.
Clear	Clears this object, so that you can reuse it. Sets all members to an initial value.
	TIP: When decoding, you can reuse the **Array** object without using **Clear**.

Table 74: ArrayEntry Methods

11.2.8.3 Encoding: Example 1

The following code samples demonstrate how to encode an **Array**. In the first example, the array is set to encode unsigned integer entries, where the entries have a fixed length of two bytes each. The example encodes two array entries:

- The first entry is encoded from a primitive **UInt** type
- The second entry is encoded from an **Buffer** containing a pre-encoded **UInt** type.

The following example includes error handling for the initial encode method only, and omits additional error handling to simplify the sample code.

```
/* EXAMPLE 1 - Array of fixed length unsigned integer values */
/* populate array structure prior call to Array.EncodeInit() */
/* encode unsigned integers in the array */

Array array = new Array();
ArrayEntry arrayEntry = new ArrayEntry();
array.PrimitiveType = DataTypes.UINT;

/* send fixed length values where each uint is 2 bytes */
array.ItemLength = 2;

/* begin encoding of array - assumes that encIter is already populated with buffer and version
information, store return value to determine success or failure */
if ((retCode = Array.EncodeInit(encIter)) < CodecReturnCode.SUCCESS)
{
```

```
    /* error condition - switch our success value to false so we can roll back */
    success = false;
    /* print out message with return value string, value, and text */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with Array.EncodeInit. Error Text:
        {2}\n", error.ErrorId, error.SysError, error.Text);
}
else
{
    UInt uInt = new UInt();
    uInt.Value = 23456;
    /* array encoding was successful */
    /* encode first entry from a UInt from a primitive type */
    retCode = arrayEntry.Encode(encIter, uInt);
    /* encode second entry from a pre-encoded UInt contained in a buffer */
    arrayEntry.EncodedData(encUInt);
    retCode = arrayEntry.Encode(encIter);
}

/* complete array encoding. If success parameter is true, this will finalize encoding. If
success parameter is false, this will roll back encoding prior to encodeInit */
retCode = array.encodeComplete(encIter, success);
```

Code Example 18: Array Encoding Example #1

11.2.8.4 Encoding: Example 2

This example demonstrates encoding an **Array** containing ASCII string values. The example includes error handling for the initial encode function only, and omits additional error handling to simplify the sample code.

```
/* EXAMPLE 2 - Array of variable length ASCII string values */
/* populate array structure prior to call to Array.EncodeInit() */
/* encode ASCII Strings in the array */
Buffer stringBuf = new Buffer();
array.PrimitiveType = DataTypes.ASCII_STRING;
/* itemLength 0 indicates variable length entries */
array.ItemLength = 0;

/* begin encoding of array - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
if ((retCode = array.EncodeInit(encIter)) < CodecReturnCode.SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = false;
    /* print out message with return value string, value, and text */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with Array.EncodeInit. Error Text: {2}",
    error.ErrorId, error.SysError, error.Text);
}
else
{
    stringBuf.Data("ENTRY 1");
```

```
    /* array encoding was successful */
    /* encode first entry from a buffer containing an ASCII_STRING primitive type */
    retCode = arrayEntry.Encode(encIter, stringBuf);
}

/* complete array encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to encodeInit */
retCode = array.EncodeComplete(encIter, success);
```

Code Example 19: Array Encoding Example #2

11.2.8.5 Decoding: Example

The following example decodes an **Array** and each of its entries to the primitive value. This sample code assumes the contained primitive type is an **UInt**. Typically an application invokes the specific primitive decoder for the contained type or uses a switch statement to allow for a more generic array entry decoder. This example uses the same **DecodeIterator** when calling the primitive decoder function. An application could optionally use a new **DecodeIterator** by setting the encoded entry buffer on a new iterator. To simplify the example, some error handling is omitted.

```
/* decode into the array structure header */
if ((retCode = array.Decode(decIter)) >= CodecReturnCode.SUCCESS)
{
   /* decode each array entry */
   while ((retCode = arrayEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
   {
      if (retCode < CodecReturnCode.SUCCESS)
      {
         /* decoding failure tends to be unrecoverable */
         Console.WriteLine("Error ({0}) (errno: {1}) encountered with ArrayEntry.Decode. Error Text:
            {2}", error.ErrorId, error.SysError, error.Text);
      }
      else
      {
         /* Decode array entry into primitive type. We can use the same decode iterator, or set
         the encoded entry buffer onto a new iterator */
         retCode = uInt.Decode(decIter);
      }
   }
}
else
{
   /* decoding failure tends to be unrecoverable */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Array.Decode. Error Text: {2}",
   error.ErrorId, error.SysError, error.Text);
}
```

Code Example 20: Array Decoding Example

11.2.9 Buffer

Buffer represents some type of user-provided content along with the content's length. **Buffer** can:

- Represent various buffer and string types, such as ASCII, RMTES, or UTF8 strings.
- Contain or reference encoded data on both container and message header structures.

No validation or enforcement checks are performed on the contents of an **Buffer**. Any desired validation can be performed by the user depending on the specific type of content represented by **Buffer**. Null termination is not required with this type.

Though **Buffer**s are typically backed by **ByteBuffer**s, they can also be backed by **String**s.

- When decoding, the backing **ByteBuffer** is available via the **Buffer.Data** method. When accessing backing data, use **Buffer.Position** method for the position and **Buffer.Length** method for the length, not the position and limit of the backing **ByteBuffer** returned from **Buffer.Data**.
- When encoding, it is the user's responsibility to provide a **ByteBuffer** of suitable length to the **Buffer.Data** method. LSEG recommends that users pool their **ByteBuffers** for reuse, otherwise it will be garbage collected whenever the reference is lost. Blank buffers are conveyed as a zero-length **Buffer**.
- When decoding, the **Buffer.IsBlank** method will return **true** when the **Buffer.Length** is **0**.
- When encoding, to specify blank, back the **Buffer** with a zero-length **ByteBuffer** (**ByteBuffer**'s position and limit equal) or call **Buffer.Data** method with length of **0**.

11.2.9.1 Methods

Buffer includes the following methods:

METHOD	DESCRIPTION
Length	The length, in bytes, of the content pointed to by **Data**. After encoding, the **Length** method can be used to get the number of bytes encoded.
	NOTE: The backing **ByteBuffer** is initially set along with initial position and length. This method returns the initial length if there was no operation on the backing **ByteBuffer** that would change the position (such as get or put). If the backing **ByteBuffer** position has been changed by reading or writing to the buffer, this method returns the change in position (i.e. difference between current position and initial position).
Position	Returns the position of the buffer.
IsBlank	Returns **true** if the length is **0**, otherwise **false**.
Data	Returns a **ByteBuffer** that contains some type of content, where the specific type description of the content is provided outside of the Enterprise Transport API via an external source (domain model definition, field dictionary, etc.). • Do not use the position and limit from the **ByteBuffer**. • Use **Position** and **Length** from the **Buffer**.
	NOTE: If data is backed by a **String**, garbage is created to return a **ByteBuffer**.
Data(ByteBuffer)	Sets the **Buffer** data to the **ByteBuffer**. Position and length are derived from the **ByteBuffer**'s position and limit.
Data(ByteBuffer, position, length)	Sets the **Buffer** data to the **ByteBuffer**. Position and length will be set to the specified position and length.
Data(String)	Sets the **Buffer** data to the contents of the **String**. This **Buffer**'s position will be set to zero and length will be set to the specified **String**'s length.

Table 75: Buffer Methods

METHOD	DESCRIPTION
Equals	Tests whether the **Buffer** is equal to another specified **Buffer**. The two objects are equal if they have the same length and the two sequence of elements are equal. If one buffer is backed by a **String** and the other buffer is backed by a **ByteBuffer**, the **String** will be compared as 8-bit ASCII.
Copy	Utility to copy this Buffer's data, starting at this Buffer's position, for this Buffer's length, to a destination. The destination can be another **Buffer** or a **ByteBuffer**. The destination must have adequate space.
Encode	Encodes a **Buffer**.
Decode	Decodes a **Buffer**.
ToString	Converts the underlying buffer into a **String**. This should only be called when the **Buffer** is known to contain ASCII data.
	 **WARNING!** Unless the underlying buffer is a **String**, this method creates garbage.
ToHexString	Converts the underlying buffer into a formatted hexadecimal **String**.
	 **WARNING!** This method creates garbage.
Clear	Clears this object, so that you can reuse it. Sets the backing **ByteBuffer**/**String** to null and position and length to **0**.

Table 75: Buffer Methods

11.2.9.2 Example

For performance purposes contents are not copied while decoding **Buffer** type. This may result in the **Buffer.Data** exposing additional encoded contents beyond the **Buffer.Position** and **Buffer.Length** to be exposed. The user can determine appropriate handling to suit their needs. One option is to display the **Buffer** as an ASCII string using **Buffer.ToString** method as illustrated in the following example:

```
/* display only the specified length of Buffer contents */
Console.WriteLine(buffer.ToString());
```

Code Example 21: Displaying Contents of a Buffer

11.2.10 RMTES Decoding

Use special consideration when handling and converting **RmtesBuffer**s that contain RMTES data. This allows for the application of partial content updates, used to efficiently change already received RMTES content by sending only those portions that need to be changed. For a more detailed description of RMTES, refer to the *Multilingual Text Encoding Standard Specification*.

The typical process for handling RMTES content contained in an **RmtesBuffer** involves storing content, applying partial updates, and converting to the desired character set. The Transport API provides several structures and functions to help with this storage and conversion as described in the following sections.

WARNING! RMTES processing is an expensive procedure that incurs multiple content copies. To avoid unnecessary processing, users should confirm that content providers are actually sending RMTES prior to using this function. If the content type is not RMTES, do not use this function[a].

a. Although the type specified in the field dictionary may indicate RMTES, the actual content might not be encoded as such. Unless content uses RMTES encoding, this functionality is not necessary.

11.2.10.1 RmtesCacheBuffer Methods

The **RmtesCacheBuffer** is a simple class used to store initial RMTES content and when applying partial updates. Any character set conversions should be performed on the content stored in the **RmtesCacheBuffer**.

RmtesCacheBuffer includes the following methods:

METHOD	DESCRIPTION
Length	Gets the length of the content pointed to by **Data**; it represents the number of bytes used in the cache. For example, if **Data** refers to 100 bytes and nothing is cached, **Length** should be set to 0. If **Data** refers to 100 bytes, and 50 bytes are currently in cache, **Length** should be set to 50. Sets the length (in bytes) of the actual data that is cached.
Data	Gets the RMTES content as a **ByteBuffer**. The length member should be set to the number of bytes of data in the buffer. Sets the **ByteBuffer** data to that of the input. The length must be set separately for the **RmtesCacheBuffer** using **Length**.
AllocatedLength	Returns the length (in bytes) allocated when creating **Data**. This is typically larger than **Length** to allow for the growth of **Data** when applying future partial updates. Sets the **AllocatedLength** of the data (in bytes).

Table 76: RmtesCacheBuffer Methods

11.2.10.2 RmtesBuffer Methods

The **RmtesBuffer** is a simple structure used to store RMTES content. Any character set conversions should be performed on the content stored in the **RmtesBuffer**.

RmtesBuffer includes the following methods:

METHOD	DESCRIPTION
AllocatedLength	Gets the length (in bytes) allocated when creating **Data**. This is typically larger than **Length** to allow for the growth of **Data** when applying future partial updates. Sets the **AllocatedLength** of the data (in bytes).
Data	Gets the RMTES content as a **ByteBuffer**. The length member should be set to the number of bytes of data in the buffer. Sets the **ByteBuffer** data to that of the input. The length must be set separately for the **RmtesBuffer** using **Length**.
Length	Gets the length integer of the content pointed to by **Data**; it represents the number of bytes of RMTES content. Sets the length (in bytes) of the actual data.
ToString	Converts the underlying buffer into a String. This should only be called when the **RmtesBuffer** is known to contain UTF-16 data.
	WARNING! Unless the underlying buffer is a **String**, this method creates garbage.

Table 77:

11.2.10.3 RmtesDecoder Methods

The **RmtesDecoder** tool manages caching and decoding of data. Its inputs are the **RmtesBuffer** and **RmtesCacheBuffer** to be decoded.

DECODE INTERFACE	DESCRIPTION
RMTESApplyToCache(Buffer, RmtesCacheBuffer)	Applies the buffer's partial update data to the **RmtesCacheBuffer**.
	NOTE: The **RmtesCacheBuffer** must refer to enough memory for storing and modifying the RMTES content.
hasPartialRMTESUpdate(Buffer)	Returns a boolean for whether the buffer contains a partial update command: • **true**: RMTES content in the buffer contains a partial update command. • **false**: RMTES content in the buffer does not contain a partial update command.
RMTESToUCS2(RmtesBuffer, RmtesCacheBuffer)	Converts the given **RmtesCacheBuffer** into UCS2 Unicode and stores the data into the **RmtesBuffer**.
RMTESToUTF8(RmtesBuffer, RmtesCacheBuffer)	Converts the given **RmtesCacheBuffer** into UTF8 Unicode and stores the data into the **RmtesBuffer**

Table 78: RmtesDecoder Methods

11.2.10.4 Example: Converting RMTES to UTF-8

The following example illustrates how to store and convert RMTES content. This example converts from RMTES to UTF-8 and assumes that:

- The input buffer is populated with RMTES content.
- The allocated size of 100 bytes is sufficient for conversion and storage.

To simplify the example, some error handling is omitted.

```
/* create cache buffer for storing RMTES and applying partial updates */
```

```
RmtesCacheBuffer rmtesCache = new RmtesCacheBuffer(100);
/* create RmtesBuffer to convert into */
RmtesBuffer rmtesBuffer = new RmtesBuffer(100);
/* create RmtesDecoder used for the decoding process */
RmtesDecoder decoder = new RmtesDecoder();
/*Our Buffer of data we are converting */
Buffer data = new Buffer();
/* apply RMTES content to cache, if successful convert to UTF-8 */
if ((retVal = decoder.RMTESApplyToCache(data, rmtesCache)) < CodecReturnCode.SUCCESS)
{
   /* error while applying to cache */
   Console.WriteLine("Error encountered while applying buffer to RMTES cache. Error code: "
   + retVal.GetAsString());
}
else if ((retVal = decoder.RMTESToUTF8(rmtesBuffer, rmtesCache)) < CodecReturnCode.SUCCESS)
{
   /* error when converting */
   Console.WriteLine("Error encountered while converting from RMTES to UTF-8. Error code: "
   + retVal.GetAsString());
}
else
{
   /* SUCCESS: Conversion was successful - application can now use converted content stored in
   rmtesBuffer */
}
```

Code Example 22: Converting RMTES to UTF-8 Example

11.2.10.5 Example: Converting RMTES to UCS-2

The following example illustrates storing and converting RMTES content. This example converts from RMTES to UCS-2 and assumes that:

- The input buffer is populated with RMTES content.
- The allocated size of 100 bytes is sufficient for conversion and storage.

To simplify the example, some error handling is omitted.

```
/* create cache buffer for storing RMTES and applying partial updates */
RmtesCacheBuffer rmtesCache = new RmtesCacheBuffer(100);
/* create RmtesBuffer to convert into */
RmtesBuffer rmtesBuffer = new RmtesBuffer(100);
/* create RmtesDecoder used for the decoding process */
RmtesDecoder decoder = new RmtesDecoder();
/*Our Buffer of data we are converting */
Buffer data = new Buffer();
/* apply RMTES content to cache, if successful convert to UCS-2 */
if ((retVal = decoder.RMTESApplyToCache(data, rmtesCache)) < CodecReturnCode.SUCCESS)
{
    /* error while applying to cache */
    Console.WriteLine("Error encountered while applying buffer to RMTES cache. Error code: "
    + retVal.GetAsString());
}
else if ((retVal = decoder.RMTESToUCS2(rmtesBuffer, rmtesCache)) < CodecReturnCode.SUCCESS)
{
    /* error when converting */
    Console.WriteLine("Error encountered while converting from RMTES to UCS-2. Error code: "
    + retVal.GetAsString());
}
else
{
    /* SUCCESS: Conversion was successful - application can now use converted content stored in
    rmtesBuffer */
}
```

Code Example 23: Converting RMTES to UCS-2 Example

11.3 Container Types

Container Types can model more complex data representations and have their contents modified at a more granular level than primitive types. Some container types leverage simple entry replacement when changes occur, while other container types offer entry-specific actions to handle changes to individual entries. The Enterprise Transport API offers several uniform (i.e., homogeneous) container types, meaning that all entries house the same type of data. Additionally, there are several non-uniform (i.e., heterogeneous) container types in which different entries can hold varying types of data.

The **`DataTypes`** enumeration exposes values that define the type of a container. For example, when a **`ContainerType`** is housed in an **`Msg`**, the message would indicate the **`ContainerType`**'s enumerated value. Values ranging from 128 to 224 represent container types. Enterprise Transport API messages and container types can house other Enterprise Transport API container types. Only the **`FieldList`** and **`ElementList`** container types can house both primitive types and other container types.

The following table provides a brief description of each container type and its housed entries.

ENUM TYPE NAME	DESCRIPTION	ENTRY TYPE INFORMATION
DataTypes.FIELD_LIST	**Container Type: `FieldList`** A highly optimized, non-uniform type, that contains field identifier-value paired entries. **`FieldId`** refers to specific name and type information as defined in an external field dictionary (such as **RDMFieldDictionary**). You can further optimize this type by using set-defined data as described in Section 11.6. For more details on this container, refer to Section 11.3.1.	Entry type is **`FieldEntry`**, which can house any **`DataType`**, including set-defined data (Section 11.6), base primitive types (Section 11.2), and container types. • If the information and entry being updated contains a primitive type, previously stored or displayed data is replaced. • If the entry contains another container type, action values associated with that type specify how to update the information.
DataTypes.ELEMENT_LIST	**Container Type: `ElementList`** A self-describing, non-uniform type, with each entry containing **`Name`**, **`DataType`**, and a value. This type is equivalent to **`FieldList`**, but without the optimizations provided through **`FieldId`** use. Use of set-defined data allows for further optimization, as discussed in Section 11.6. For more details on this container, refer to Section 11.3.2.	Entry type is **`ElementEntry`**, which can house any **`DataType`**, including set-defined data (Section 11.6), base primitive types (Section 11.2), and container types. • If the updating information and entry contain a primitive type, any previously stored or displayed data is replaced. • If the entry contains another container type, action values associated with that type specify how to update the information.
DataTypes.MAP	**Container Type: `Map`** A container of key-value paired entries. **`Map`** is a uniform type, where the base primitive type of each entry's key and the **`ContainerType`** of each entry's payload are specified on the **`Map`**. • For more information on base primitive types, refer to Section 11.2. • For more details on this container, refer to Section 11.3.3.	Entry type is **`MapEntry`**, which can include only container types, as specified on the **`Map`**. Each entry's key is a base primitive type, as specified on the **`Map`**. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.

Table 79: Enterprise Transport API Container Types

ENUM TYPE NAME	DESCRIPTION	ENTRY TYPE INFORMATION
DataTypes.SERIES	**Container Type:** **`Series`** A uniform type, where the **`ContainerType`** of each entry is specified on the **`Series`**. This container is often used to represent table-based information, where no explicit indexing is present or required. As entries are received, the user should append them to any previously-received entries. For more details on this container, refer to Section 11.3.4.	Entry type is **`SeriesEntry`**, which can include only container types, as specified on the **`Series`**. **`SeriesEntry`** types do not contain explicit actions; though as entries are received, the user should append them to any previously received entries.
DataTypes.VECTOR	**Container Type:** **`Vector`** A container of position index-value paired entries. This container is a uniform type, where the **`ContainerType`** of each entry's payload is specified on the **`Vector`**. Each entry's **`Index`** is represented by an unsigned integer. For more details on this container, refer to Section 11.3.5.	Entry type is **`VectorEntry`**, which can house only container types, as specified on the **`Vector`**. Each entry's **`Index`** is an unsigned integer. Each entry has an associated action, which informs the user on how to apply the information stored in the entry.
DataTypes.FILTER_LIST	**Container Type:** **`FilterList`** A non-uniform container of **`FilterId`**-value paired entries. A **`FilterId`** corresponds to one of 32 possible bit-value identifiers, typically defined by a domain model specification. **`FilterId`**'s can be used to indicate interest or presence of specific entries through the inclusion of the **`FilterId`** in the message key's **`Filter`** member. • For more information about the message key, refer to Section 12.1.1.3. • For more details on this container, refer to Section 11.3.6.	Entry type is **`FilterEntry`**, which can house only container types. Though the **`FilterList`** can specify a **`ContainerType`**, each entry can override this specification to house a different type. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.
DataTypes.MSG	**Container Type:** **`Msg`** Indicates that the contents are another message. This allows the application to house a message within a message or a message within another container's entries. This type is typically used with posting (described in Section 13.9). For more details on message encoding and decoding, refer to 12, Message Package Detailed View.	None
DataTypes.NO_DATA	**Container Type:** None Indicates there are no contents. • When **`DataTypes.NO_DATA`** is housed in a message, the message has no payload. • If **`DataTypes.NO_DATA`** is housed in a container type, each container entry has no payload.[a]	None
DataTypes.ANSI_PAGE	**Container Type:** None Indicates that contents are ANSI Page format. Though the Enterprise Transport API does not natively support encoding and decoding for the ANSI Page format, the Enterprise Transport API supports the use of a separate ANSI Page encoder/decoder. For further details, refer to the *Enterprise Transport API ANSI Library Manual*. For more details on housing non-LSEG Rssl Wire Format types inside of container types, refer to Section 11.3.7.	None

Table 79: Enterprise Transport API Container Types (Continued)

ENUM TYPE NAME	DESCRIPTION	ENTRY TYPE INFORMATION
DataTypes.XML	**Container Type:** None Indicates that contents are XML-formatted data. Though the Enterprise Transport API does not natively support encoding and decoding XML, the Enterprise Transport API supports the use of a separate XML encoder/decoder. For more details on housing non-LSEG Rssl Wire Format types inside of container types, refer to Section 11.3.7.	None
DataTypes.OPAQUE	**Container Type:** None Indicates that the contents are opaque and additional details are not provided through the Enterprise Transport API. Any specific information about the concrete type housed in the opaque payload should be defined in the specific domain model associated with the message. For more details on housing non-LSEG Rssl Wire Format types inside of container types, refer to Section 11.3.7.	None

Table 79: Enterprise Transport API Container Types (Continued)

a. An `FilterList` can indicate a type of `DataTypes.NO_DATA`, however an individual `FilterEntry` can override using the entry-specific `ContainerType`.

11.3.1 FieldList

The **FieldList** is a container of entries (known as **FieldEntry**s) paired by the values of their field identifiers. A ***field identifier*** (known as a **FieldId**), is a signed, two-byte value that refers to specific name and type information defined by an external field dictionary (e.g., **RDMFieldDictionary**). A field list can contain zero to N^4 entries, where zero indicates an empty field list.

11.3.1.1 Methods

FieldList includes the following methods:

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Fags**) that indicate the presence of optional field list content. For more information about **FieldListFlags** values, refer to Section 11.3.1.2. • You can use the following convenient methods to set specific **FieldListFlags**: **ApplyHasInfo**, **ApplyHasSetData**, **ApplyHasSetId**, **ApplyHasStandardData**. • You can use the following convenient methods to check whether specific **FieldListFlags** are set: **CheckHasInfo**, **CheckHasSetData**, **CheckHasSetId**, **CheckHasStandardData**.
DictionaryId	Sets or gets a two-byte, signed integer (**DictionaryId**) that refers to the external dictionary family for use when interpreting content in this **FieldEntry**. The field dictionary contains specific name and type information which correlates to **FieldId** values present in each **FieldEntry**. An example of this would be the **RDMFieldDictionary**, which has a **DictionaryId** value of **1**. If not present, a value of **1** should be assumed. If using the default dictionary (**RDMFieldDictionary**), **DictionaryId** is not required and is assumed have an id value of **1**. A **DictionaryId** should be provided as part of the initial refresh message on a stream or on the first refresh message after issuing a **CLEAR_CACHE** command. A **DictionaryId** can be changed in two ways. • If a **DictionaryId** is provided on a refresh message (solicited or unsolicited), the specified dictionary is used across all messages on the stream until a new **DictionaryId** is provided in a subsequent refresh. This new dictionary is now used for all messages on the stream until another **DictionaryId** is provided. • If an **FieldEntry** contains a **FieldId** of **0**, this reserved value indicates a temporary dictionary change. In this situation, this entry's value is the new **DictionaryId** (encoded / decoded as an **Int**). When a **DictionaryId** is changed in this manner, the change is only in effect on the remaining entries in the field list or until another **FieldId** of **0** is encountered. Any **ContainerType**s housed inside the remaining entries also adopt this temporary dictionary. When the end of the field list is reached, the **DictionaryId** from the refresh takes precedence once again. **DictionaryId** values have an allowed range of **-16,384** to **16,383**.
FieldListNum	Sets or gets the **FieldListNum**, which is a two-byte, signed integer referring to an external **Fieldlist** template, also known as a ***record template***. The record template contains information about all possible fields in a stream and is typically used by caching implementations to pre-allocate storage. **FieldListNum** values have an allowed range of **-32,768** to **32,767**.
SetId	Sets or gets a two-byte, unsigned integer (**SetId**) corresponding to the set definition used for encoding or decoding the set-defined data in this **FieldList**. • When encoding, this is the set definition used to encode any set-defined content. • When decoding, this is the set definition used for decoding any set-defined content. If a **SetId** value is not present on a message containing set-defined data, a **SetId** of **0** is implied. **SetId** values have an allowed range of **0** to **32,767**. Currently, only values **0** to **15** are used. These indicate locally-defined set definition use. Refer to Section 11.6 for more information.

Table 80: FieldList Methods

4. A field list currently has a maximum entry count of 65,535, where the first 255 entries may contain set-defined types. This type has an approximate maximum encoded length of 5 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of each field entry has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

METHOD	DESCRIPTION
EncodedSetData	Sets or gets **EncodedSetData**, which is a **Buffer** (with position and length) that contains the encoded set-defined data, if any, contained in the message. If populated, contents are described by the set definition associated with the **SetId** member. If this is populated while encoding, this is assumed to be pre-encoded set data. If this is populated while decoding, this represents encoded set data. For more information, refer to Section 11.6.
EncodedEntries	Returns the **EncodedEntries**, which is a **Buffer** (with position and length) that contains the encoded **FieldId**-value pair encoded data, if any, contained in the message. This would refer to encoded **FieldList** payload and length information.
EncodeInit	Begins encoding a **FieldList**. The Enterprise Transport API will encode all content into the **Buffer** to which the passed in **EncodeIterator** refers. Entries can be encoded after this method returns. • If you are encoding set-defined data, pass in the set definition database to this method. The Enterprise Transport API will use the specified definition to validate and optimize content while encoding. • To reserve space for encoding, pass in a maximum length hint value (associated with the expected maximum encoded length of set-defined content in this **FieldList**). If the approximate encoded set data length is not known, you can pass in a value of **0**. For more details on local set definitions, refer to Section 11.6.
EncodeComplete	Completes the encoding of a **FieldList**. This method expects the same **EncodeIterator** that was used with **EncodeInit** and all entries. • If encoding succeeds, a **bool success** parameter setting of **true** finishes the encoding. • If encoding any entry fails, a **bool success** parameter setting of **false** rolls back encoding to the last successfully encoded point in the contents. Encode all field entries prior to this call.
Decode	Begins decoding a **FieldList** from the **Buffer** referenced in the **DecodeIterator**. This method allows the user to pass in local set definitions. If the **FieldList** contains set-defined data (e.g., if the **FieldListFlags.HAS_SET_DATA** flag is present), the Transport API decodes the set-defined entries when definitions are present. Otherwise, set-defined entries are skipped while decoding entries.
Clear	Clears the object so that you can reuse it. When decoding, you can reuse **FieldList** without needing to call **Clear**.

Table 80: FieldList Methods (Continued)

11.3.1.2 FieldListFlags Values

FIELD LIST FLAG	MEANING
FieldListFlags.HAS_FIELD_LIST_INFO	Indicates that **DictionaryId** and **FieldListNum** members are present, which should be provided as part of the initial refresh message on a stream or on the first refresh message after issuance of a **CLEAR_CACHE** command.
FieldListFlags.HAS_STANDARD_DATA	Indicates that the **FieldList** contains standard **FieldId**-value pair encoded data. This value can be set in addition to **FieldListFlags.HAS_SET_DATA** if both standard and set-defined data are present in this **FieldList**. If no entries are present in the **FieldList**, this flag value should not be set.

Table 81: FieldList Flags Values

FIELD LIST FLAG	MEANING
FieldListFlags.HAS_SET_DATA	Indicates that the **FieldList** contains set-defined data. This value can be set in addition to **FieldListFlags.HAS_STANDARD_DATA** if both standard and set-defined data are present in this **FieldList**. If no entries are present in the **FieldList**, this flag value should not be set. For more information, refer to Section 11.6.
FieldListFlags.HAS_SET_ID	Indicates the presence of a **SetId**, used to determine the set definition used for encoding or decoding the set data on this **FieldList**. For more information, refer to Section 11.6.

Table 81: FieldList Flags Values(Continued)

11.3.1.3 FieldEntry Methods

A **FieldList** can contain multiple **FieldEntry**s, and each **FieldEntry** can house any **DataType**. This includes primitive types (as described in), set-defined types (as described in), or container types. If updating information, when the **FieldEntry** contains a primitive type, it replaces any previously stored or displayed data associated with the same **FieldId**. If the **FieldEntry** contains another container type, action values associated with that type indicate how to modify the information.

METHOD	DESCRIPTION
FieldId	Sets or gets the signed two-byte value (**FieldId**) that refers to specific name and type information defined by an external field dictionary, such as the **RDMFieldDictionary**. Negative **FieldId** values typically refer to user-defined values while positive **FieldId** values typically refer to LSEG-defined values. **FieldId** has an allowable range of -32,768 to 32,767 where LSEG defines positive values and the user defines negative values. A **FieldId** value of **0** is reserved to indicate **DictionaryId** changes, where the type of **FieldId 0** is an **Int**.
DataType	Sets or gets the **DataType** of this **FieldEntry**'s contents. • While encoding, **DataType** must be set to the enumerated value of the type being encoded. • While decoding, if **DataType** is **DataTypes.UNKNOWN**, the user must determine the type of contained information from the associated field dictionary. If set-defined data is used, **DataType** will indicate specific **DataType** information as indicated by the set definition.
EncodedData	Sets or gets **EncodedData**, which is a **Buffer** (with position and length) containing the encoded content of this **FieldEntry**. • If populated on encode methods, this indicates that data is pre-encoded and **EncodedData** will be copied while encoding. • If populated on decoding functions, this refers to the encoded **FieldEntry**'s payload and length information.
Encode(w/primitiveType)	Encodes a **FieldEntry** with a primitive data type (e.g. **UInt**). This method expects the same **EncodeIterator** used with **FieldList.EncodeInit**. You must properly populate **FieldEntry.FieldId** and **FieldEntry.DataType**. Call this method for each **PrimitiveType** entry being encoded.
Encode	Encodes a **FieldEntry** with pre-encoded data. This method expects the same **EncodeIterator** used with **FieldList.EncodeInit**. You must properly populate **FieldEntry.FieldId** and **FieldEntry.DataType**. Set **EncodedData** with pre-encoded data before calling this method. Call this method for each pre-encoded entry being encoded.
EncodeBlank	Encodes a blank **FieldEntry**. This method expects the same **EncodeIterator** used with **FieldList.EncodeInit**. Call this method for each blank entry being encoded.

Table 82: FieldEntry Methods

METHOD	DESCRIPTION
EncodeInit	Encodes a **FieldEntry** from a complex type, such as a container type or an array. This method expects the same **EncodeIterator** used with **FieldList.EncodeInit**. You must properly populate **FieldEntry.FieldId** and **FieldEntry.DataType**. To reserve space needed for encoding, you can pass in a maximum-length hint value, associated with the expected maximum-encoded length of this field. If the approximate encoded length is not known, you can pass in a value of **0** which allows the maximum content length. Typical use (e.g. encode an element list as a field entry): 1. Call **FieldEntry.EncodeInit**. 2. Call one or more encoding methods for the complex type using the same buffer. 3. Call **FieldEntry.EncodeComplete**.
EncodeComplete	Completes encoding of a **FieldEntry** for a complex type, such as a container type or an array. This method expects the same **EncodeIterator** used with **FieldList.EncodeInit**, **FieldEntry.EncodeInit**, and all other entry encoding. • If encoding succeeds, set the **bool success** to **true** to finish entry encoding. • If encoding the entry fails, set the **bool success** parameter to **false** to roll back the encoding of this particular **FieldEntry**.
Decode	Decodes a **FieldEntry**, expecting the same **DecodeIterator** used with **FieldList.Decode** and populates **EncodedData** with the entry's encoded contents. • If decoding set-defined entries, the **FieldEntry.DataType** populates with the type from the set definition. • If decoding standard **FieldId**-value data, **FieldEntry.DataType** is set to **DataTypes.UNKNOWN**, indicating that the user must determine the type from a field dictionary. After determining the type, the specific decode method can be called if needed. Calling **FieldEntry.Decode** again will begin decoding the next entry in the **FieldList** until no more entries are available.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, **FieldEntry** can be reused without using **Clear**.

Table 82: FieldEntry Methods (Continued)

11.3.1.4 Rippling

The **FieldList** container supports rippling fields. When ***rippling***, newly received content associated with a **FieldId** replaces previously received content associated with the same **FieldId**. The previously-received content is moved to a new **FieldId** (typically indicated in a field dictionary[5]). Rippling is typically used as a way to reduce bandwidth consumption. Normally, if previously-received data were still relevant, it would need to be sent with subsequent updates even though the value was not changing. Rippling allows this data to be removed from subsequent updates; however the consumer must use the ripple information from a field dictionary to correctly propagate previously received content. Rippling is the responsibility of the consumer application, and the Enterprise Transport API does not perform entry rippling.

11.3.1.5 Encoding Example

The following example illustrates how to encode an **FieldList**. The example encodes four **FieldEntry** values:

- The first encodes an entry from a primitive **Date** type
- The second from a pre-encoded buffer containing an encoded **UInt**
- The third as a blank **Real** value
- The fourth as an **Array** complex type. The pattern followed while encoding the fourth entry can be used for encoding of any container type into an **FieldEntry**.

This example demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted (though it should be performed). This example shows encoding of standard **fieldId**-value data.

```
/* populate field list structure prior to call to FieldList.EncodeInit()
NOTE: the FieldId, DictionaryId and FieldListNum values used for this example do not correspond
to actual id values */

/* indicate that standard data will be encoded and that dictionaryId and fieldListNum are included */
fieldList.ApplyHasStandardData();
fieldList.ApplyHasInfo();
/* populate DictionaryId and FieldListNum with info needed to cross-reference fieldIds and cache */
fieldList.DictionaryId = 2;
fieldList.FieldListNum = 5;

/* begin encoding of field list - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
if ((retCode = fieldList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with FieldList.EncodeInit. Error Text: {2}",
   error.ErrorId, error.SysError, error.Text);
}
else
{
   /* fieldListInit encoding was successful */
   /* create a single FieldEntry and reuse for each entry */
   FieldEntry fieldEntry = new FieldEntry();
   /* stack allocate a date and populate {day, month, year} */
   Date date = new Date();
```

5. In the Domain Model Field Dictionary, the **RIPPLES TO** column defines the **fieldId** information to use when rippling.

```
date.Month = 3;
date.Day = 18;
date.Year = 2013;

/* FIRST Field Entry: encode entry from the Date primitive type. Populate and encode field entry
with FieldId and DataType information for this field */
fieldEntry.FieldId = 16;
fieldEntry.DataType = DataTypes.DATE;
retCode = fieldEntry.Encode(encIter, date);

/* SECOND Field Entry: encode entry from preencoded buffer containing an encoded UInt type */
/* populate and encode field entry with FieldId and DataType information for this field */
/* because we are re-populating all values on FieldEntry, there is no need to clear it */
fieldEntry.FieldId = 1080;
fieldEntry.DataType = DataTypes.UINT;
/* assuming encUInt is a Buffer with length and data properly populated */
fieldEntry.EncodedData = encUInt;
/* no data parameter is passed in because pre-encoded data is set on FieldEntry itself */
retCode = fieldEntry.Encode(encIter);

/* THIRD Field Entry: encode entry as a blank Real primitive type */
/* populate and encode field entry with FieldId and DataType information for this field */
fieldEntry.fieldId(22);
fieldEntry.DataType = DataTypes.REAL;
retCode = fieldEntry.EncodeBlank(encIter);

/* FOURTH Field Entry: encode entry as a complex type, Array primitive */
/* populate and encode field entry with FieldId and DataType information for this field */
/* need to ensure that FieldEntry is appropriately cleared - clearing will ensure that EncodedData
is properly emptied */
fieldEntry.Clear();
fieldEntry.FieldId = 1021;
fieldEntry.DataType = DataTypes.ARRAY;
/* begin complex field entry encoding, we are not sure of the approximate max encoding length */
retCode = fieldEntry.EncodeInit(encIter, 0);
{
   /* now encode nested container using its own specific encode methods */
   /* encode Real values into the array */
   array.PrimitiveType = DataTypes.REAL;
   /* values are variable length */
   array.ItemLength = 0;
   /* begin encoding of array - using same encIterator as field list */
   if ((retCode = array.EncodeInit(encIter)) < CodecReturnCode.SUCCESS)

   /*----- Continue encoding array entries. See example in Section 11.2.8 ---- */

   /* Complete nested container encoding */
   retCode = array.EncodeComplete(encIter, success);
}
/* complete encoding of complex field entry. If any array encoding failed, success is false */
```

```
    retCode = fieldEntry.EncodeComplete(encIter, success);
}
/* complete FieldList encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to EncodeInit */
retCode = fieldList.EncodeComplete(encIter, success);
```

Code Example 24: `FieldList` Encoding Example

11.3.1.6 Decoding Example

The following example demonstrates how to decode an **FieldList** and is structured to decode each entry to the contained value. This example uses a switch statement to invoke the specific decoder for the contained type, however to simplify the example, necessary cases and some error handling are omitted. This example uses the same **DecodeIterator** when calling the primitive decoder function. An application could optionally use a new **DecodeIterator** by setting the **EncodedData** on a new iterator.

```
/* decode into the field list structure */
if ((retCode = fieldList.Decode(decIter, localSetDefs)) >= CodecReturnCode.SUCCESS)
{
   /* decode each field entry */
   while ((retCode = fieldEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
   {
      if (retCode < CodecReturnCode.SUCCESS)
      {
         /* decoding failure tends to be unrecoverable */
         Console.WriteLine("Error ({0}) (errno: {1}) encountered with FieldEntry.Decode. Error Text:
            {2}",
            error.ErrorId, error.SysError, error.Text);
      }
      else
      {
         /* look up type in field dictionary and call correct primitive decode method */
         IDictionaryEntry dictionaryEntry = dictionary.Entry(fieldEntry.FieldId);
         switch (dictionaryEntry.RwfType)
         {
            case DataTypes.REAL:
               retCode = real.Decode(decIter);
            break;
            case DataTypes.DATE:
               retCode = date.Decode(decIter);
            break;
            /* full switch statement omitted to shorten sample code */
         }
      }
   }
}
else
{
   /* decoding failure tends to be unrecoverable */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with FieldList.Decode. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
}
```

Code Example 25: FieldList Decoding Example

11.3.2 ElementList

ElementList is a self-describing container type. Each entry, known as an **ElementEntry**, contains an element **Name**, **DataType** enumeration, and value. An element list is equivalent to **FieldList**, where name and type information is present in each element entry instead of optimized via a field dictionary. An element list can contain zero to N^6 entries, where zero indicates an empty element list.

11.3.2.1 Methods

METHOD	DESCRIPTION
Flags	Sets or gets **Flags**, which is a combination of bit values that indicate whether optional, element-list content is present. For more information about **ElementListFlags** values, refer to Section 11.3.2.2. • You can use the following convenient methods to set specific **ElementListFlags**: A**pplyHasInfo**, A**pplyHasSetData**, A**pplyHasSetId**, A**pplyHasStandardData**. • You can use the following convenient methods to check whether specific **ElementListFlags** are set: **CheckHasInfo**, **CheckHasSetData**, **CheckHasSetId**, **CheckHasStandardData**.
ElementListNum	Sets or gets a two-byte signed integer (**ElementListNum**) that refers to an external element-list template, also known as a ***record template***. A record template contains information about all possible entries contained in the stream and is typically used by caching mechanisms to pre-allocate storage. **ElementListNum** values have a range of **-32,768** to **32,767**.
SetId	Sets or gets a two-byte unsigned integer (**SetId**) that corresponds to the set definition used for encoding or decoding the set-defined data in this **ElementList**. • When encoding, this is the set definition used to encode any set-defined content. • When decoding, this is the set definition used for decoding any set-defined content. **SetId** values have an allowed range of **0** to **32,767**. Currently, only values **0** to **15** are used. These indicate locally-defined set definition use. If a **SetId** value is not present on a message containing set-defined data, a **SetId** of **0** is implied. For more information, refer to Section 11.6.
EncodedSetData	Sets or gets the encoded set-defined data (**EncodedSetData**), which is a **Buffer** (with position and length) containing the encoded set-defined data, if any, contained in the message. If populated, contents are described by the set definition associated with the **SetId** member. • If this is populated while encoding, this is assumed to be pre-encoded set data. • If this is populated while decoding, this represents encoded set data. For more information, refer to Section 11.6.
EncodedEntries	Returns **EncodedEntries**, which is a **Buffer** (with position and length) that contains all encoded element **Name**, **DataType**, value encoded data, if any, contained in the message. This would refer to encoded **ElementList** payload and length information.

Table 83: ElementList Methods

6. An element list currently has a maximum entry count of 65,535, where the first 255 entries may contain set-defined types. This type has an approximate maximum encoded length of 5 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of element entry has a maximum encoded length of 65,535 bytes. These limitations can change in subsequent releases.

METHOD	DESCRIPTION
EncodeInit	Starts encoding an **ElementList**. The Transport API encodes data into the **ITransportBuffer** referred to by the **EncodeIterator**. Entries can be encoded after this method returns. • If encoding set-defined data, pass the set definition database into this method. The Transport API uses the specified definition to validate and optimize content while encoding. • You can reserve space for encoding by passing in a maximum-length hint value (associated with the expected maximum-encoded length of set-defined content in this **ElementList**). If the approximate length of encoded set data is not known, you can pass in a value of 0. For more details on local set definitions, refer to Section 11.6.
EncodeComplete	Completes the encoding of an **ElementList**. This method expects the same **EncodeIterator** used with **ElementList.EncodeInit** and all entries. • If all entries were encoded successfully, a **bool success** parameter setting of true finishes encoding. • If encoding of any entry failed, a **bool success** parameter setting of false rolls back the encoding process to the last successfully encoded point in the contents. Encode any element entries prior to this call.
Decode	Starts decoding an **ElementList**. This method will decode from the **Buffer** referred to by the passed-in **DecodeIterator** and allows the user to pass in local set definitions. If the **ElementList** contains set-defined data (e.g., **ElementListFlags.HAS_SET_DATA** is present), the Transport API will decode set-defined entries when their definitions are present. Otherwise, the Transport API skips set-defined entries when decoding entries.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **ElementList** without needing to call **Clear**.

Table 83: ElementList Methods (Continued)

11.3.2.2 ElementListFlags Values

ELEMENTLISTFLAG VALUES	MEANING
ElementListFlags.HAS_ELEMENT_LIST_INFO	Indicates the presence of the **ElementListNum** member. This member is provided as part of the initial refresh message on a stream or on the first refresh message after a **CLEAR_CACHE** command.
ElementListFlags.HAS_STANDARD_DATA	Indicates that the **ElementList** contains standard element **Name**, **DataType**, value-encoded data. You can set this value in addition to **ElementListFlags.HAS_SET_DATA** if both standard and set-defined data are present in this **ElementList**. If the **ElementList** does not have entries, do not set this flag value.
ElementListFlags.HAS_SET_DATA	Indicates that **ElementList** contains set-defined data. • If both standard and set-defined data are present in this **ElementList**, this value can be set in addition to **ElementListFlags.HAS_STANDARD_DATA**. • If the **ElementList** does not have entries, do not set this flag value. For more information, refer to Section 11.6.
ElementListFlags.HAS_SET_ID	Indicates the presence of a **SetId** and determines the set definition to use when encoding or decoding set data on this **ElementList**. For more information, refer to Section 11.6.

Table 84: ElementList Flags Values

11.3.2.3 ElementEntry Methods

Each **ElementList** can contain multiple **ElementEntry**s and each **ElementEntry** can house any **DataType**, including primitive types (refer to), set-defined types (refer to), or container types. If an **ElementEntry** is a part of updating information and contains a primitive type, any previously stored or displayed data is replaced. If an **ElementEntry** contains another container type, action values associated with that type indicate how to modify data.

METHOD	DESCRIPTION
Name	Sets or gets a **Buffer** containing the **Name** associated with this **ElementEntry**. Element names are defined outside of the Enterprise Transport API, typically as part of a domain model specification or dictionary. A **Name** can be empty; however this provides no identifying information for the element. The **Name** buffer allows for content length ranging from 0 bytes to 32,767 bytes.
DataType	Sets or gets the **DataType**, which defines the **DataType** of this **ElementEntry**'s contents. • While encoding, set this to the enumerated value of the target type. • While decoding, **DataType** describes the type of contained data so that the correct decoder can be used. If set-defined data is used, **DataType** will indicate any specific **DataType** information as defined in the set definition.
EncodedData	Sets or gets the encoded content (**EncodedData**) of this **ElementEntry**. If populated on encode methods, this indicates that data is pre-encoded and **EncodedData**copies while encoding. While decoding, this refers to the encoded **ElementEntry**'s payload and length data.
Encode(w/ primitiveType)	Encodes an **ElementEntry** with a primitive data type (e.g. **UInt**). This method expects the same **EncodeIterator** used with **ElementList.EncodeInit**. **ElementEntry.Name** and **ElementEntry.DataType** must be properly populated. Call this method for each **PrimitiveType** entry that you want to encode.
Encode	Encodes an **ElementEntry** with pre-encoded data. This method expects the same **EncodeIterator** used with **ElementList.EncodeInit**. You must properly populate **ElementEntry.Name** and **ElementEntry.DataType** and also set **EncodedData** with pre-encoded data before calling this method. Call this method for each pre-encoded entry that you want to encode.
EncodeBlank	Encodes a blank **ElementEntry**. This method expects the same **EncodeIterator** used with **ElementList.EncodeInit**. You must properly populate **ElementEntry.Name** and **ElementEntry.DataType**. Call this method for each blank entry that you want to encode.
EncodeInit	Encodes an **ElementEntry** from a complex type, such as a container type or an array. This method expects the same EncodeIterator used with **ElementList.EncodeInit**. You must properly populate **ElementEntry.Name** and **ElementEntry.DataType**. To reserve the appropriate amount of space while encoding, you can pass in a max-length hint value (associated with the expected maximum-encoded length of this element) to this method. If the approximate encoded length is not known, you can pass in a value of **0**. Typical use (e.g. encode an element list as a field entry): **1.** Call **ElementEntry.EncodeInit**. **2.** Call one or more encoding methods for the complex type using the same buffer. **3.** Call **ElementEntry.EncodeComplete**.

Table 85: ElementEntry Methods

METHOD	DESCRIPTION
EncodeComplete	Completes the encoding of an **ElementEntry**. This method expects the same **EncodeIterator** used with **ElementList.EncodeInit**, **ElementEntry.EncodeInit**, and all other entry encoding. • If this specific entry is encoded successfully, a **bool success** parameter setting of true finishes entry encoding. • If this specific entry fails to encode, a **bool success** parameter setting of false rolls back the encoding of only this **ElementEntry**.
Decode	Decodes an **ElementEntry**. This method expects the same **DecodeIterator** used with **ElementList.Decode** and populates **EncodedData** with encoded entry contents. After this method returns, you can use the **ElementEntry.DataType** to invoke the correct contained type's decode methods. Calling **ElementEntry.Decode** again starts decoding the next entry in the **ElementList** until no more entries are available.
Clear	Clears this object, so that you can reuse it.  **TIP:** When decoding, you can reuse **ElementEntry** without using **Clear**.

Table 85: ElementEntry Methods (Continued)

11.3.2.4 ElementList Encoding Example

The following example demonstrates how to encode an **ElementList** and encodes four **ElementEntry** values:

- The first encodes an entry from a primitive **Time** type
- The second encodes from a pre-encoded buffer containing an encoded **UInt**
- The third encodes as a blank **Real** value
- The fourth encodes as an **FieldList** container type

The pattern used to encode the fourth entry can be used to encode any container type into an **ElementEntry**. This example demonstrates error handling for the initial encode function. However, additional error handling is omitted to simplify the example. This example shows the encoding of standard **Name**, **DataType**, and **Value** data.

```
/* populate element list structure prior to call to ElementList.EncodeInit() */
/* NOTE: the element names and elementListNum values used for this example may not correspond to actual
      name values */
/* indicate that standard data will be encoded and that elementListNum is included */
elemList.ApplyHasStandardData();
elemList.CheckHasInfo();

/* populate elementListNum with info needed to cache */
elemList.ElementListNum = 5;

/* begin encoding of element list - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
if ((retCode = elemList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with ElementList.EncodeInit. Error Text:
   {2}",
      error.ErrorId, error.SysError, error.Text);
}
else
{
   /* ElementList.EncodeInit encoding was successful */
   /* create a single ElementEntry and reuse for each entry */
   ElementEntry elemEntry = new ElementEntry();

   Time time = new Time();
   time.Hour(10);
   time.Minute(21);
   time.Second(16);
   time.Millisecond(777);
   Buffer elementEntryName = new Buffer();

   /* FIRST Element Entry: encode entry from the Time primitive type */
   /* populate and encode element entry with name and dataType information for this element */
   elementEntryName.Data("Element1 - Primitive");
   elemEntry.Name = elementEntryName;
   elemEntry.DataType = DataTypes.TIME;
```

```
    retCode = elemEntry.Encode(encIter, time);

    /* SECOND Element Entry: encode entry from preencoded buffer containing an encoded UInt type */
    /* populate and encode element entry with name and dataType information for this element */
    /* because we are re-populating all values on ElementEntry, there is no need to clear it */
    elementEntryName.Data("Element2 - Pre-Encoded");
    elemEntry.Name = elementEntryName;
    elemEntry.DataType = DataTypes.UINT;
    /* assuming encUInt is a Buffer with length and data properly populated */
    elemEntry.EncodedData = encUInt;
    /* no data parameter is passed in because pre-encoded data is set on ElementEntry itself */
    retCode = elemEntry.Encode(encIter);

    /* THIRD Element Entry: encode entry as a blank Real primitive type */
    /* populate and encode element entry with name and dataType information for this element */
    elementEntryName.Data("Element3 - Blank");
    elemEntry.name = elementEntryName;
    elemEntry.DataType = DataTypes.REAL;
    retCode = elemEntry.EncodeBlank(encIter);

    /* FOURTH Element Entry: encode entry as a container type, FieldList */
    /* populate and encode element entry with name and dataType information for this element */
    /* need to ensure that ElementEntry is appropriately cleared - clearing will ensure that encData
           is properly emptied */
    elemEntry.Clear();
    elementEntryName.Data("Element4 - Container");
    elemEntry.Name = elementEntryName;
    elemEntry.DataType = DataTypes.FIELD_LIST;
    /* begin complex element entry encoding, we are not sure of the approximate max encoding length */
    retCode = elemEntry.EncodeInit(encIter, 0);
    {
        /* now encode nested container using its own specific encode methods */
        /* begin encoding of field list - using same encIterator as element list */
        fieldList.ApplyHasStandardData();
        if ((retCode = fieldList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding field entries. See example in Section Section 11.3.1 ---- */

        /* Complete nested container encoding */
        retCode = fieldList.EncodeComplete(encIter, success);
    }
    /* complete encoding of complex element entry. If any field list encoding failed, success is false */
    retCode = elemEntry.EncodeComplete(encIter, success);
}
/* complete elementList encoding. If success parameter is true, this will finalize encoding.
    If success parameter is false, this will roll back encoding prior to encodeInit */
retCode = elemList.EncodeComplete(encIter, success);
```

Code Example 26: `ElementList` Encoding Example

11.3.2.5 ElementList Decoding Example

The following sample demonstrates how to decode an **ElementList** and is structured to decode each entry to its contained value. This example uses a switch statement to invoke the specific decoder for the contained type, however for sample clarity, unnecessary cases have been omitted. This example uses the same **DecodeIterator** when calling the primitive decoder function. An application could optionally use a new **DecodeIterator** by setting the **EncodedData** on a new iterator. For simplification, the example omits some error handling.

```
/* decode into the element list structure */
if ((retCode = elemList.Eecode(decIter, localSetDefs)) >= CodecReturnCode.SUCCESS)
{
   /* decode each element entry */
   while ((retCode = elemEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
   {
      if (retCode < CodecReturnCode.SUCCESS)
      {
         /* decoding failure tends to be unrecoverable */
         Console.WriteLine("Error ({0}) (errno: {1}) encountered with ElementEntry.Decode. Error Text:
            {2}",
         error.ErrorId, error.SysError, error.Text);
      }
      else
      {
         /* use elemEntry.dataType to call correct primitive decode method */
         switch (elemEntry.DataType)
         {
            case DataTypes.REAL:
               retCode = real.Decode(decIter);
            break;
            case DataTypes.TIME:
               retCode = time.Decode(decIter);
            break;
            /* full switch statement omitted to shorten sample code */
         }
      }
   }
   }
}
else
{
   /* decoding failure tends to be unrecoverable */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with ElementList.Decode. Error Text: {2}\n",
      error.ErrorId, error.SysError, error.Text);
}
```

Code Example 27: ElementList Decoding Example

11.3.3 Map

The **Map** is a uniform container type of associated key-value pair entries. Each entry, known as an **MapEntry**, contains an entry key, which is a base primitive type () and value. An **Map** can contain zero to *N*[7] entries, where zero entries indicate an empty **Map**.

11.3.3.1 Map Methods

An **Map** structure contains the following Methods:

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) to indicate the presence of optional **Map** content. For more information about **MapFlags** values, refer to Section 11.3.3.2. • You can use the following convenient methods to set specific **MapFlags**: **ApplyHasKeyFieldId**, **ApplyHasPerEntryPermData**, **ApplyHasSetDefs**, **ApplyHasSummaryData**, **ApplyHasTotalCountHint**. • You can use the following convenient methods to check whether specific **MapFlags** are set: **CheckHasKeyFieldId**, **CheckHasPerEntryPermData**, **CheckHasSetDefs**, **CheckHasSummaryData**, **CheckHasTotalCountHint**.
KeyPrimitiveType	Sets or gets the value (**KeyPrimitiveType**) that describes the base primitive type of each **MapEntry**'s key. **KeyPrimitiveType** accepts primitive DataTypes (values between **1** and **63**), cannot be specified as blank, and cannot be the **DataTypes.ARRAY** or **DataTypes.UNKNOWN** primitive types. For more information about base primitive types, refer to Section 11.2.
KeyFieldId	(Optional) Sets or gets a **FieldId** associated with the entry key information. This is mainly used as an optimization to avoid inclusion of redundant data. In situations where key information is also a member of the entry payload (e.g., Order Id for Market By Order domain type), this allows removal of data from each entry's payload prior to encoding as it is already present via the key and **KeyFieldId**. **KeyFieldId** has an allowable range of -32,768 to 32,767 where positive values are LSEG-defined and negative values are user-defined.
ContainerType	Sets or gets the value (**DataType**) that describes the container type of each **MapEntry**'s payload.
TotalCountHint	Sets or gets a four-byte unsigned integer (**TotalCountHint**) that indicates an approximate total number of entries associated with this stream. This is typically used when multiple **Map** containers are spread across multiple parts of a refresh message (for more information about message fragmentation and multi-part message handling, refer to Section 13.1). **TotalCountHint** provides an approximation of the total number of entries sent across all maps on all parts of the refresh message. This information is useful when determining the amount of resources to allocate for caching or displaying all expected entries. **TotalCountHint** values have a range of 0 to 1,073,741,824.

Table 86: Map Methods

7. A **Map** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 5 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of a **MapEntry** has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

METHOD	DESCRIPTION
EncodedSummaryData	Sets or gets the **EncodedSummaryData**, which is a **ITransportBuffer** (with position and length) that contains the encoded summary data, if any, contained in the message. If populated, summary data contains information that applies to every entry encoded in the **Map** (e.g., currency type). The container type of summary data should match the **ContainerType** specified on the **Map**. If **EncodedSummaryData** is populated while encoding, contents are used as pre-encoded summary data. Encoded summary data has maximum allowed length of 32,767 bytes. For more information, refer to Section 11.5.
EncodedSetDefs	Sets or gets the EncodedSetDefs, which is a **Buffer** (with position and length) that contains the encoded local set definitions, if any, contained in the message. If populated, these definitions correspond to data contained within the **Map**'s entries and are used for encoding or decoding their contents. Encoded local set definitions have a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.6.
EncodedEntries	Returns the **EncodedEntries**, which is a **Buffer** (with position and length) that contains the length and pointer to the all encoded key-value pair data, if any, contained in the message. This would refer to encoded **Map** payload and length information.
EncodeInit	Begins encoding a **Map** which can include summary data (Section 11.5) and Local Set Definitions (Section 11.6). • If summary data and set definitions are pre-encoded, you can populate them on the **EncodedSummaryData** and **EncodedSetDefs** prior to calling **Map.EncodeInit**. Additional work is not needed to complete encoding this content. • If summary data and set definitions are not pre-encoded, **Map.EncodeInit** performs the **Init** for these values. You must call the corresponding **Complete** method after this content is encoded. • You can reserve the appropriate amount of space while encoding by passing in summary data and set definition encoded length hint values to this method. If either is not being encoded or the approximate encoded length is unknown, you can pass in a value of **0**. This is required only when content is not pre-encoded.
EncodeComplete	Completes the encoding of a **Map**. This method expects the same **EncodeIterator** that was used with **Map.EncodeInit**, any summary data, set data, and all entries. • If encoding was successful, the **bool success** parameter should be set to **true** to finish encoding. • If any component failed to encode, the **bool success** parameter should be set to **false** which rolls back the encoding process to the last previously successful encoded point in the contents. Encode all map content prior to this call.
EncodeSummaryDataComplete	Completes encoding of any non-pre-encoded **Map** summary data. If **MapFlags.HAS_SUMMARY_DATA** is set and **EncodeSummaryData** is not populated, summary data is expected after **Map.EncodeInit** or **Map.EncodeSetDefsComplete** returns. This method expects the same **EncodeIterator** used with previous map encoding methods. • If encoding succeeds, the **bool success** parameter should be true to finish encoding. • If encoding fails, the **bool success** parameter should be set to false to roll back to the last previously successful encoded point in the contents. If both **MapFlags.HAS_SUMMARY_DATA** and **MapFlags.HAS_SET_DEFS** are present, then set definitions are expected first, and summary data is encoded after the call to **Map.EncodeSetDefsComplete**.

Table 86: Map Methods (Continued)

METHOD	DESCRIPTION
EncodeSetDefsComplete	Completes encoding of any non pre-encoded local set definition data. If **MapFlags.HAS_SET_DEFS** is set and **EncodedSetDefs** is not populated, local set definition data is expected after **Map.EncodeInit** returns. This method expects the same **EncodeIterator** used with **Map.EncodeInit**. • If encoding succeeds, the **bool success** parameter should be **true** to finish encoding. • If encoding fails, the **bool success** parameter should be set to **false** to roll back to the last previously successful encoded point in the contents. If both **MapFlags.HAS_SUMMARY_DATA** and **MapFlags.HAS_SET_DEFS** are present, set definitions are expected first, while any summary data is encoded after the call to **Map.EncodeSetDefsComplete**.
Decode	Begins decoding a **Map**. This method will decode from the **Buffer** to which the passed-in **DecodeIterator** refers.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **Map** without using **Clear**.

Table 86: Map Methods (Continued)

11.3.3.2 MapFlags Values

FLAG ENUMERATION	MEANING
MapFlags.HAS_KEY_FIELD_ID	Indicates the presence of the **KeyFieldId** member. **KeyFieldId** should be provided if the key information is also a field that would be contained in the entry payload. This optimization allows **KeyFieldId** to be included once instead of in every entry's payload.
MapFlags.HAS_TOTAL_COUNT_HINT	Indicates the presence of the **TotalCountHint** member. This member can provide an approximation of the total number of entries sent across all maps on all parts of the refresh message. This information is useful when determining the amount of resources to allocate for caching or displaying all expected entries.
MapFlags.HAS_PER_ENTRY_PERM_DATA	Indicates that permission information is included with some map entries. The **Map** encoding functionality sets this flag value on the user's behalf if any entry is encoded with its own **PermData**. A decoding application can check this flag to determine if any contained entry has **PermData**, often useful for fan out devices (if an entry does not have **PermData**, the fan out device can likely pass on data and not worry about special permissioning for the entry). Each entry will also indicate the presence of permission data via the use of **MapEntryFlag.HAS_PERM_DATA**.
MapFlags.HAS_SUMMARY_DATA	Indicates that the **Map** contains summary data. If this flag is set while encoding, summary data must be provided by encoding or populating **EncodedSummaryData** with pre-encoded information. If this flag is set while decoding, summary data is contained as part of the **Map** and the user can choose whether to decode it.
MapFlags.HAS_SET_DEFS	Indicates that the **Map** contains local set definition information. Local set definitions correspond to data contained within this **Map**'s entries and are used for encoding or decoding their contents. For more information, refer to Section 11.6.

Table 87: MapFlags Values

11.3.3.3 MapEntry Methods

MapEntrys can house only other container types. **Map** is a uniform type, where the **Map.ContainerType** indicates the single type housed in each entry. Each entry has an associated action which informs the user of how to apply the information contained in the entry.

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values to indicate the presence of any optional **MapEntry** content. For more information about **MapEntryFlags**values, refer to Table 11.3.3.4. • You can use the following convenient methods to set specific **MapEntryFlags**: **ApplyHasPermData**. • You can use the following convenient methods to check whether specific **MapEntryFlags** are set: **CheckHasPermData**.
Action	Sets or gets the entry **Action** which helps to manage change processing rules and tells the consumer how to apply the information contained in the entry. For specific information about possible **Action**'s associated with an **MapEntry**, refer to Table 11.3.3.5.
EncodedKey	Sets or gets the **EncodedKey**, which is a **Buffer** (with position and length) that contains the encoded map entry key information. The encoded type of the key corresponds to the **Map**'s **KeyPrimitiveType**. The key value must be a base primitive type and cannot be blank, **DataTypes.ARRAY**, or **DataTypes.UNKNOWN** primitive types. If populated on encode functions, this indicates that the key is pre-encoded and **EncodedKey** will be copied while encoding. While decoding, this would contain only this encoded **MapEntry** key's payload and length information.
PermData	(Optional) Sets or gets authorization information for this specific entry. If present, **MapEntryFlag.HAS_PERM_DATA** should be set. **PermData** has a maximum allowed length of 32,767 bytes. • For more information on permissioning, refer to Section 11.4. • For more information about **MapEntryFlags** values, refer to Table 11.3.3.4.
EncodedData	Sets or gets **EncodedData**, which is a **Buffer** (with position and length) that contains the encoded content of this **MapEntry**. If populated on encode methods, this indicates that data is pre-encoded, and **EncodedData** will be copied while encoding. While decoding, this would refer to this encoded **MapEntry**'s payload and length information.**MapEntryFlag.**
Encode(w/PrimitiveType)	Encodes a **MapEntry** with a primitive data type (e.g. **UInt**). This method expects the same **EncodeIterator** used with **Map.EncodeInit** and is called after **Map.EncodeInit** and after completing any summary data and local set definition data encoding. Call this method for each **PrimitiveType** entry you want to encode.
Encode	Encodes a **MapEntry** from pre-encoded data. This method expects the same **EncodeIterator** used with **Map.EncodeInit**. You must set the pre-encoded map entry payload via the **MapEntry.EncodedData** method prior to calling this method. This method is called after **Map.EncodeInit** and after completing any summary data and local set definition data encoding. Call this method for each pre-encoded entry you want to encode.

Table 88: MapEntry Methods

METHOD	DESCRIPTION
EncodeInit(w/KeyPrimitiveType)	Encodes a **MapEntry** from a container type. This method expects the same **EncodeIterator** used with **Map.EncodeInit**. After this call, you can use housed-type encode methods to encode contained types. The **KeyPrimitiveType** accepts primitive **DataType** (values between **1** and **63**), cannot be specified as blank and cannot be the **DataType.ARRAY** or **DataType.UNKNOWN** primitive types. For more information about base primitive types, refer to Section 11.2. You call this method after **Map.EncodeInit** and after encoding any summary data and local set definition data. To reserve the appropriate amount of space for encoding, you can pass in a max-length hint value, associated with the expected maximum encoded length of this entry. If the approximate encoded length is unknown, you can pass in a value of **0**.
EncodeInit	Encodes a **MapEntry** with pre-encoded primitive key. This method expects the same **EncodeIterator** used with **Map.EncodeInit**. After this call, you can use housed-type encode methods to encode contained types. Call this method after **Map.EncodeInit** and after encoding any summary data and local set definition data. To reserve the appropriate amount of space for encoding, you can pass in a max-length hint value, associated with the expected maximum encoded length of this entry. If the approximate encoded length is unknown, you can pass in a value of **0**. Set **Map.EncodedKey** with pre-encoded data before calling this method.
EncodeComplete	Completes the encoding of a **MapEntry**. This method expects the same **EncodeIterator** used with **Map.EncodeInit**, **MapEntry.EncodeInit**, and all other encoding for this container. • If this specific map entry is encoded successfully, the **bool success** parameter should be set to **true** to finish entry encoding. • If this specific entry fails to encode, the **bool success** parameter should be set to **false** to roll back the encoding of only this **MapEntry**.
Decode(keyData)	Decodes a **MapEntry** and can optionally decode the **MapEntry.EncodedKey**. This method expects the same **DecodeIterator** used with **Map.Decode**. This populates **EncodedData** with encoded entry contents and **EncodedKey** with the encoded entry key. After this method returns, you can use the **Map.ContainerType** to invoke the correct contained-type's decode methods. Calling **MapEntry.Decode** again continues the decoding of the next entry in the **Map** until no more entries are available. **KeyData** can be any valid **KeyPrimitiveType** primitive (e.g. **UInt**). If **KeyData** is non NULL, the entry key will also be decoded into the specified **KeyData**. As entries are received, the action dictates how to apply contents.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **MapEntry** without using **Clear**.

Table 88: MapEntry Methods (Continued)

11.3.3.4 MapEntry Flag Enumeration Value

FLAG ENUMERATION	MEANING
MapEntryFlag.HAS_PERM_DATA	Indicates that the container entry includes a **PermData** member and also specifies any authorization information for this entry. For more information, refer to .

Table 89: MapEntryFlags Values

11.3.3.5 MapEntry Action Enumeration Values

ACTION ENUMERATION	MEANING
ADD	Indicates that the consumer should add the entry. An add action typically occurs when an entry is initially provided. It is possible for multiple add actions to occur for the same entry. If this occurs, any previously received data associated with the entry should be replaced with the newly added information.
UPDATE	Indicates that the consumer should update any previously stored or displayed information with the contents of this entry. An update action typically occurs when an entry has already been added and changes to the contents need to be conveyed. If an update action occurs prior to the add action for the same entry, the update action should be ignored.
DELETE	Indicates that the consumer should remove any stored or displayed information associated with the entry. No map entry payload is included when the action is delete.

Table 90: `MapEntryActions` Values

11.3.3.6 MapEntry Encoding Example

The following sample illustrates the encoding of a **`Map`** containing **`FieldList`** values. The example encodes three **`MapEntry`** values as well as summary data:

- The first entry is encoded with an update action type and a passed in key value.
- The second entry is encoded with an add action type, pre-encoded data, and pre-encoded key.
- The third entry is encoded with a delete action type.

This example also demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted, though it should be performed.

```
/* populate map structure prior to call to Map.EncodeInit() */
/* NOTE: the key names used for this example may not correspond to actual name values */

/* indicate that summary data and a total count hint will be encoded */
map.ApplyHasSummaryData();
map.ApplyHasTotalCountHint();
/* populate maps KeyPrimitiveType and containerType */
map.ContainerType = DataTypes.FIELD_LIST;
map.KeyPrimitiveType = DataTypes.UINT;
/* populate total count hint with approximate expected entry count */
map.TotalCountHint = 3;

/* begin encoding of map - assumes that encIter is already populated with buffer and version information,
store return value to determine success or failure */
/* expect summary data of approx. 100 bytes, no set definition data */
if ((retCode = map.EncodeInit(encIter, 100, 0 )) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Map.EncodeInit. Error Text: {2}\n",
      error.ErrorId, error.SysError, error.Text);
}
```

```
else
{
    /* mapInit encoding was successful */
    /* create a single MapEntry and FieldList and reuse for each entry */
    UInt entryKeyUInt = new UInt();
    /* encode expected summary data, init for this was done by Map.EncodeInit - this type should
    match map.ContainerType */
    {
        /* now encode nested container using its own specific encode methods */
        /* begin encoding of field list - using same encIterator as map list */
        fieldList.ApplyHasStandardData();
        if ((retCode = fieldList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding field entries. See example in Section 11.3.1.5 ---- */

        /* Complete nested container encoding */
        retCode = fieldList.EncodeComplete(encIter, success);
    }

    /* complete encoding of summary data. If any field list encoding failed, success is false */
    retCode = map.EncodeSummaryDataComplete(encIter, success);

    /* FIRST Map Entry: encode entry from non pre-encoded data and key. Approx. encoded length unknown */
    mapEntry.Action = MapEntryActions.UPDATE;
    entryKeyUInt.Value(1);
    retCode = mapEntry.EncodeInit(encIter, entryKeyUInt, 0);
    /* encode contained field list - this type should match map.ContainerType */
    {
        /* now encode nested container using its own specific encode methods */
        /* clear, then begin encoding of field list - using same encIterator as map */
        fieldList.Clear();
        fieldList.ApplyHasStandardData();
        if ((retCode = fieldList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding field entries. See example in Section 11.3.1.5 ---- */\

        /* Complete nested container encoding */
        retCode = fieldList.EncodeComplete(encIter, success);
    }

    retCode = mapEntry.EncodeComplete(encIter, success);

    /* SECOND Map Entry: encode entry from pre-encoded buffer containing an encoded FieldList */
    /* because we are re-populating all values on MapEntry, there is no need to clear it */
    mapEntry.action = MapEntryActions.ADD;
    /* assuming encUInt Buffer contains the pre-encoded key with length and data properly populated */
    mapEntry.EncodedKey = encUInt;
    /* assuming encFieldList Buffer contains the pre-encoded payload with data and length populated */
    mapEntry.EncodedData = encFieldList;
    /* no keyData parameter is passed in because pre-encoded key is set on MapEntry itself */
    retCode = mapEntry.Encode(encIter);
    /* THIRD Map Entry: encode entry with delete action. Delete actions have no payload */
    /* need to ensure that MapEntry is appropriately cleared - clearing will ensure that EncodedData and
    encKey are properly emptied */
    mapEntry.Clear();
    mapEntry.Action = MapEntryActions.DELETE;
    entryKeyUInt.Value(3);
    /* entryKeyUInt parameter is passed in for key primitive value. EncodedData is empty for delete */
    retCode = mapEntry.Encode(encIter, entryKeyUInt);
}
```

```
/* complete map encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to EncodeInit */
retCode = map.EncodeComplete(encIter, success);
```

Code Example 28: Map Encoding Example

11.3.3.7 MapEntry Decoding Example

The following sample demonstrates the decoding of a **Map** and is structured to decode each entry to the contained value. This sample assumes that the housed container type is an **FieldList** and that the **KeyPrimitiveType** is **DataTypes.INT**. This sample also uses the **MapEntry.Decode** function to perform key decoding. Typically an application would invoke the specific container-type decoder for the housed type or use a switch statement to allow for a more generic map entry decoder. This example uses the same **DecodeIterator** when calling the content's decoder function. An application could optionally use a new **DecodeIterator** by setting the **EncodedData** on a new iterator. To simplify the sample, some error handling is omitted.

```
/* decode contents into the map structure */
if ((retCode = map.Decode(decIter)) >= CodecReturnCode.SUCCESS)
{
    /* create primitive value to have key decoded into and a single map entry to reuse */
    Int tempInt = new Int();
    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry)
    indicates to ETA that user wants to decode summary data */
    if (map.CheckHasSummaryData())
    {
        /* summary data is present. Its type should be that of map.ContainerType */
        retCode = fieldList.Decode(decIter, null);

        /* Continue decoding field entries. See example in Section 11.3.1.6 */
    }

    /* decode each map entry, passing KeyPrimitiveType decodes mapEntry key as well */
    while ((retCode = mapEntry.Decode(decIter, tempInt)) != CodecReturnCode.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCode.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            Console.WriteLine("Error ({0}) (errno: {1}) encountered with MapEntry.Decode. Error Text:
    {2}",
                error.ErrorId, error.SysError, error.Text);
        }
        else
        {
            retCode = fieldList.Decode(decIter, null);

            /* Continue decoding field entries. See example in Section 11.3.1.6 */
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with Map.Decode. Error Text: {2}",
        error.ErrorId, error.SysError, error.Text);
}
```

Code Example 29: Map Decoding Example

11.3.4 Series

The **Series** is a uniform container type. Each entry, known as an **SeriesEntry**, contains only encoded data. This container is often used to represent table-based information, where no explicit indexing is present or required. A **Series** can contain zero to *N*[8] entries, where zero entries indicates an empty **Series**.

11.3.4.1 Series Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicates the presence of optional **Series** content. For more information about flag values, refer to Section 11.3.4.2. • You can use the following convenient methods to set specific **SeriesFlags**: **ApplyHasSetDefs**, **ApplyHasSummaryData**, **ApplyHasTotalCountHint**. • You can use the following convenient methods to check whether specific **SeriesFlags** are set: **CheckHasSetDefs**, **CheckHasSummaryData**, **CheckHasTotalCountHint**.
ContainerType	Sets or gets **ContainerType**, which is a **DataType** value that describes the container type of each **SeriesEntry**'s payload.
TotalCountHint	Sets or gets a four-byte unsigned integer (**TotalCountHint**) that indicates an approximate total number of entries associated with this stream. This is typically used when multiple **Series** containers are spread across multiple parts of a refresh message (For more information about message fragmentation and multi-part message handling, refer to Section 13.1). The **TotalCountHint** provides an approximation of the total number of entries sent across all series on all parts of the refresh message. This information is useful when determining the amount of resources to allocate for caching or displaying all expected entries. **TotalCountHint** values have a range of 0 to 1,073,741,824.
EncodedSummaryData	Sets or gets **EncodedSummaryData**, which is a **ITransportBuffer** (with position and length) that contains the encoded summary data, if any, contained in the message. If populated, summary data contains information that applies to every entry encoded in the **Series** (e.g., currency type). The container type of summary data should match the **ContainerType** specified on the **Series**. If **EncodedSummaryData** is populated while encoding, the contents will be used as pre-encoded summary data. For more information, refer to Section 11.5. Encoded summary data a maximum allowed length of 32,767 bytes.
EncodedSetDefs	Sets or gets **EncodedSetDefs**, which is a **ITransportBuffer** (with position and length) that contains the encoded local set definitions, if any, contained in the message. If populated, these definitions correspond to data contained within this **Series**'s entries and are used to encode or decode their contents. For more information, refer to Section 11.6. Encoded local set definitions have a maximum allowed length of 32,767 bytes.
EncodedEntries	Returns **EncodedEntries**, which is a **Buffer** (with position and length) that contains all encoded key-value pair encoded data, if any, contained in the message. This refers to encoded **Series** payload and length data.

Table 91: Series Methods

8. A **Series** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 4 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of an **SeriesEntry** has a maximum encoded length of 65,535 bytes. These limitations can change in subsequent releases.

METHOD	DESCRIPTION
EncodeInit	Starts encoding a **Series** and allows for the encoding of summary data (for details, refer to Section 11.5) and Local Set Definitions (for details, refer to Section 11.6). You can encode additional summary data, set definitions, or entries after this method returns. • If summary data or set definitions are pre-encoded, you populate them on the **EncodedSummaryData** and **EncodedSetDefs** prior to calling **EncodeInit**. No additional work is needed to complete the encoding of this content. • If summary data or set definitions are not pre-encoded, **EncodeInit** will perform the **Init** for these components. After this content is encoded, you must call the corresponding **Complete** methods. To reserve space while encoding, you can pass in summary data and set definition encoded length hint values to this method. If either is not being encoded or the approximate encoded length is unknown, a value of 0 can be passed in. This is only needed when the content is not pre-encoded.
EncodeComplete	Completes the encoding of a **Series**. This method expects the same **EncodeIterator** used with **Series.EncodeInit**, any summary data, set data, and all entries. • If encoding succeeds, the **bool success** parameter should be **true** to finish encoding. • If encoding fails, the **bool success** parameter should be **false** to roll back the encoding to the last previously successful encoded point in the contents. Encode all series content prior to this call.
EncodeSummaryDataComplete	Completes the encoding of any non-pre-encoded **Series** summary data. If the **SeriesFlags.HAS_SUMMARY_DATA** flag is set and **EncodedSummaryData** is not populated, summary data is expected after **Series.EncodeInit** or **Series.EncodeSetDefsComplete** returns. This method expects the same **EncodeIterator** used with previous series encoding methods. • If encoding succeeds, the **bool success** parameter should be **true** to finish encoding. • If encoding fails, the **bool success** parameter should be **false** to roll back the encoding prior to summary data. If both **SeriesFlags.HAS_SUMMARY_DATA** and **SeriesFlags.HAS_SET_DEFS** are present, set definitions are expected first, while any summary data is encoded after the call to **EncodeSetDefsComplete**.
EncodeSetDefsComplete	Completes the encoding of any non pre-encoded local set definition data. If the **SeriesFlags.HAS_SET_DEFS** flag is set and **EncodedSetDefs** is not populated, local set definition data is expected after **Series.EncodeInit** returns. This method expects the same **EncodeIterator** used with **Series.EncodeInit**. • If encoding succeeds, the **boolean success** parameter should be **true** to finish encoding. • If encoding fails, the **boolean success** parameter should be **false** to roll back the encoding prior to the set definition data. If both **SeriesFlags.HAS_SUMMARY_DATA** and **SeriesFlags.HAS_SET_DEFS** are present, set definitions are expected first, while any summary data is encoded after the call to **Series.EncodeSetDefsComplete**.
Decode	Begins decoding a **Series** from the **ITransportBuffer** specified by **DecodeIterator**.
Clear	Clears the object, so that you can reuse it.  **TIP:** When decoding, you can reuse **Series** without using **Clear**.

Table 91: Series Methods (Continued)

11.3.4.2 Series Flag Enumeration Values

FLAG	MEANING
HAS_TOTAL_COUNT_HINT	Indicates the presence of the **TotalCountHint** member, which can provide an approximation of the total number of entries sent across maps on all parts of the refresh message. Such information is useful when determining resource allocation for caching or displaying all expected entries.
HAS_SUMMARY_DATA	Indicates that the **Series** contains summary data. • If set while encoding, summary data must be provided by encoding or populating **EncodedSummaryData** with pre-encoded information. • If set while decoding, summary data is contained as part of **Series** and the user can choose to decode it.
HAS_SET_DEFS	Indicates that the **Series** contains local set definition information. Local set definitions correspond to data contained in this **Series**'s entries and encode or decode their contents. For more information, refer to Section 11.6.

Table 92: Series Flag Enumeration Values

11.3.4.3 SeriesEntry Methods

Each **SeriesEntry** can house other Container Types only. **Series** is a uniform type, where **Series.ContainerType** indicates the single type housed in each entry. As entries are received, they are appended to any previously received entries.

METHOD	DESCRIPTION
EncodedData	Sets or gets **EncodedData**, which is a **Buffer** (with position and length) that contains the encoded content of this **SeriesEntry**. • If populated on encode methods, this indicates that data is pre-encoded and **EncodedData** will be copied while encoding. • If populated while decoding, this refers to this encoded **SeriesEntry**'s payload and length data.
Encode	Encodes a **SeriesEntry** from pre-encoded data. This method expects the same **EncodeIterator** used with **Series.EncodeInit**. You can pass in the pre-encoded series entry payload via **SeriesEntry.**EncodedData. **SeriesEntry.Encode** is called after **Series.EncodeInit** and any summary data and local set definition data is encoded.
EncodeInit	Encodes a **SeriesEntry** from a container type. **SeriesEntry.EncodeInit** expects the same **EncodeIterator** used with **Series.EncodeInit**. After this call, you can use housed-type encode methods to encode the contained type. The contained type's encode method would be called after **Series.EncodeInit** and any summary data and local set definition data encoding has been completed. To reserve space while encoding, you can pass in a max-length hint value to this method. If the approximate encoded length is unknown, You can pass in a value of **0**.
EncodeComplete	Completes the encoding of a **SeriesEntry**. This method expects the same **EncodeIterator** used with **Series.EncodeInit**, **SeriesEntry.EncodeInit**, and all other encoding for this container. • If encoding succeeds, the **bool success** parameter should be **true** to finish entry encoding. • If encoding of this specific entry fails, the **bool success** parameter should be **false** to roll back the encoding of only this **SeriesEntry**.

Table 93: SeriesEntry Methods

METHOD	DESCRIPTION
Decode	Decodes a **SeriesEntry**. This method expects the same **DecodeIterator** used with **Series.Decode** and populates **EncodedData** with encoded entry. After **SeriesEntry.Decode** returns, you can use **Series.ContainerType** to invoke the correct contained type's decode methods. Calling **SeriesEntry.Decode** again decodes the next entry in the **Series** until no more entries are available. As entries are received, they are appended to previously received entries.
Clear	Clears this object, so that you can reuse it.  **TIP:** When decoding, you can reuse **SeriesEntry** without using **Clear**.

Table 93: SeriesEntry Methods (Continued)

11.3.4.4 Series Encoding Example

The following sample illustrates how to encode an **Series** containing **ElementList** values. The example encodes two **SeriesEntry** values as well as summary data.

- The first entry is encoded from an unencoded element list.
- The second entry is encoded from a buffer containing a pre-encoded element list.

The example demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted, though it should be performed.

```
/* populate series structure prior to call to Series.EncodeInit() */
/* indicate that summary data and a total count hint will be encoded */
series.ApplyHasSummaryData();
series.ApplyHasTotalCountHint();
/* populate containerType and total count hint */
series.ContainerType = DataTypes.ELEMENT_LIST;
series.TotalCountHint = 2;

/* begin encoding of series - assumes that encIter is already populated with buffer and version
information, store return value to determine success or failure */
/* summary data approximate encoded length is unknown, pass in 0 */
if ((retCode = series.EncodeInit(encIter, 0, 0)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Series.EncodeInit. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
}
else
{
   /* series init encoding was successful */
   /* create a single SeriesEntry and ElementList and reuse for each entry */
   SeriesEntry seriesEntry = new SeriesEntry();
   ElementList elementList = new ElementList();
   /* encode expected summary data, init for this was done by Series.encodeInit - this type should match
   series.ContainerType */
   {
      /* now encode nested container using its own specific encode methods */
      /* begin encoding of element list - using same encIterator as series */
      elementList.ApplyHasStandardData();
      if ((retCode = elementList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

      /*----- Continue encoding element entries. See example in Section 11.3.2 ---- */

      /* Complete nested container encoding */
      retCode = elementList.EncodeComplete(encIter, success);
   }
   /* complete encoding of summary data. If any element list encoding failed, success is false */
   retCode = series.EncodeSummaryDataComplete(encIter, success);
   /* FIRST Series Entry: encode entry from unencoded data. Approx. encoded length unknown */
   retCode = seriesEntry.EncodeInit(encIter, 0);
   /* encode contained element list - this type should match series.ContainerType */
```

```
    {
        /* now encode nested container using its own specific encode methods */
        /* clear, then begin encoding of element list - using same encIterator as series */
        elementList.Clear();
        elementList.ApplyHasStandardData();
        if ((retCode = elementList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding element entries. See example in Section 11.3.2 ---- */

        /* Complete nested container encoding */
        retCode = elementList.EncodeComplete(encIter, success);
    }
    retCode = seriesEntry.EncodeComplete(encIter, success);
    /* SECOND Series Entry: encode entry from pre-encoded buffer containing an encoded ElementList */
    /* assuming encElementList Buffer contains the pre-encoded payload with data and length populated */
    seriesEntry.EncodedData = encElementList;
    retCode = seriesEntry.Encode(encIter);
}
/* complete series encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to encodeInit */
retCode = series.EncodeComplete(encIter, success);
```

Code Example 30: `Series` Encoding Example

11.3.4.5 Series Decoding Example

The following sample illustrates how to decode an **`Series`** and is structured to decode each entry to the contained value. The sample code assumes the housed container type is an **`ElementList`**. Typically an application invokes the specific container type decoder for the housed type or uses a switch statement to allow for a more generic series entry decoder. This example uses the same **`DecodeIterator`** when calling the content's decoder function. An application could optionally use a new **`DecodeIterator`** by setting **`EncodedData`** on a new iterator. To simplify the sample, some error handling is omitted.

```
/* decode contents into the series structure */
if ((retCode = series.Decode(decIter)) >= CodecReturnCode.SUCCESS)
{
    /* create single series entry and reuse while decoding each entry */
    SeriesEntry seriesEntry = new SeriesEntry();
    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry)
    indicates to the Transport API that user wants to decode summary data */
    if (series.CheckHasSummaryData())
    {
        /* summary data is present. Its type should be that of series.ContainerType */
        ElementList elementList = new ElementList();
        retCode = elementList.Decode(decIter, null);

        /* Continue decoding element entries. See example in Section 11.3.2 */

    }
    /* decode each series entry until there are no more left */
    while ((retCode = seriesEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCode.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            Console.WriteLine("Error ({0}) (errno: {1}) encountered with SeriesEntry.Decode. Error Text:
    {2}",
                error.ErrorId, error.SysError, error.Text);
```

```
        }
        else
        {
            ElementList elementList = new ElementList();
            retCode = elementList.Decode(decIter, null);

            /* Continue decoding element entries. See example in Section 11.3.2 */
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with Series.Decode. Error Text: {2}\n",
        error.ErrorId, error.SysError, error.Text);
}
```

Code Example 31: `Series` Decoding Example

11.3.5 Vector

The **Vector** is a uniform container type of **index**-value pair entries. Each entry, known as an **VectorEntry**, contains an index that correlates to the entry's position in the information stream and value. A **Vector** can contain zero to *N*[9] entries (zero entries indicates an empty **Vector**).

11.3.5.1 Vector Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicate special behaviors and whether optional **Vector** content is present. For more information about flag values, refer to Section 11.3.5.2. • You can use the following convenient methods to set specific **VectorFlags**: **ApplyHasPerEntryPermData**, **ApplyHasSetDefs**, **ApplyHasSummaryData**, **ApplyHasTotalCountHint**, **ApplySupportsSorting**. • You can use the following convenient methods to check whether specific **VectorFlags** are set: **CheckHasPerEntryPermData**, **CheckHasSetDefs**, **CheckHasSummaryData**, **CheckHasTotalCountHint**, **CheckSupportsSorting**.
ContainerType	Sets or gets the container type (**ContainerType**; a **DataType** value) of each **VectorEntry**'s payload.
TotalCountHint	Sets or gets a four-byte, unsigned integer (**TotalCountHint**) that indicates the approximate total number of entries sent across all vectors on all parts of the refresh message. **TotalCountHint** is typically used when multiple **Vector** containers are spread across multiple parts of a refresh message (for more information about message fragmentation and multi-part message handling, refer to Section 13.1). Such information helps in determining the amount of resources to allocate for caching or displaying all expected entries. **TotalCountHint** values have a range of 0 to 1,073,741,824.
EncodedSummaryData	Sets or gets the EncodedSummaryData, which is a **Buffer** (with position and length) containing the encoded summary data contained in the message. If populated, summary data contains information that applies to every entry encoded in the **Vector** (e.g. currency type). The container type of summary data must match the **ContainerType** specified on the **Vector**. If **EncodedSummaryData** is populated while encoding, contents are used as pre-encoded summary data. Encoded summary data a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.5.
EncodedSetDefs	Sets or gets the **EncodedSetDefs**, which is a **Buffer** (with position and length) containing the encoded local set definitions contained in the message. If populated, these definitions correspond to data contained within this **Vector**'s entries and are used to encode or decode their contents. Encoded local set definitions have a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.6.
EncodedEntries	Returns the **EncodedEntries**, which is a **Buffer** (with position and length) containing the encoded **index**-value pair encoded data contained in the message. This would refer to encoded **Vector** payload and length information.

Table 94: Vector Methods

9. A **Vector** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 4 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of a **VectorEntry** has a maximum encoded length of 65,535 bytes. These limitations can change in future releases.

METHOD	DESCRIPTION
EncodeInit	Begins encoding a **Vector**. Using this method, you can encode summary data (Section 11.5) and local set definitions (Section 11.6). Further summary data, set definitions, and/or entries can be encoded after this method returns. • If summary data and set definitions are pre-encoded, they can be populated on the **EncodedSummaryData** and **EncodedSetDefs** prior to calling **Vector.EncodeInit**. No additional work is needed to complete the encoding of this content. • If summary data and set definitions are not pre-encoded, **Vector.EncodeInit** will perform the **Init** for these components. After encoding this content, the corresponding **Complete** methods must be called. • To allow extra space while encoding, you can pass in summary data and set definition encoded length hint values to this method. If either is not being encoded or the approximate encoded length is unknown, a value of **0** can be passed in. This is only needed when the content is not pre-encoded.
EncodeComplete	Completes the encoding of a **Vector**. This method expects the same **EncodeIterator** used with **Vector.EncodeInit,** any summary data, set data, and all entries. • If encoding succeeds, the **bool success** parameter should be **true** to finish encoding. • If any component fails to encode, the **bool success** parameter should be **false** to roll back encoding to the last successfully-encoded point in the contents. Vector content should be encoded prior to this call.
EncodeSummaryDataComplete	Completes the encoding of **Vector** summary data. If **VectorFlags.HAS_SUMMARY_DATA** is set and **EncodedSummaryData** is not populated, summary data is expected after **Vector.EncodeInit** or **Vector.EncodeSetDefsComplete** returns. This method expects the same **EncodeIterator** used with previous vector encoding methods. • If encoding succeeds, the **bool success** parameter should be **true** to finish encoding. • If any data fail to encode, the **bool success** parameter should be **false** to roll back to the last successfully-encoded point prior to summary data. • If both **VectorFlags.HAS_SUMMARY_DATA** and **VectorFlags.HAS_SET_DEFS** are present, set definitions are expected first, while summary data is encoded after the call to **Vector.EncodeSetDefsComplete**.
EncodeSetDefsComplete	Completes the encoding of local set definition data. If **VectorFlags.HAS_SET_DEFS** is set and **EncodedSetDefs** is not populated, local set definition data is expected after **Vector.EncodeInit** returns. This method expects the same **EncodeIterator** used with **Vector.EncodeInit**. • If set definition data encodes successfully, the **boolean success** parameter should be **true** to finish encoding. • If set definition data fails to encode, the **boolean success** parameter should be **false** to roll back to the last successfully-encoded point prior to set definition data. • If both **VectorFlags.HAS_SUMMARY_DATA** and **VectorFlags.HAS_SET_DEFS** are present, set definitions are expected first, and then any summary data is encoded after the call to **Vector.EncodeSetDefsComplete**.
Decode	Begins decoding a **Vector**. This method decodes from the **ITransportBuffer** to which the passed-in **DecodeIterator** refers.
Clear	Clears this object, so that you can reuse it.  **TIP:** When decoding, you can reuse **Vector** without using **Clear**.

Table 94: Vector Methods (Continued)

11.3.5.2 Vector Flag Enumeration Values

FLAG	MEANING
VectorFlags.HAS_TOTAL_COUNT_HINT	Indicates that the **TotalCountHint** member is present. **TotalCountHint** can provide an approximation of the total number of entries sent across all vectors on all parts of the refresh message. Such information is useful in determining the amount of resources to allocate for caching or displaying all expected entries.
VectorFlags.HAS_PER_ENTRY_PERM_DATA	Indicates that permission information is included with some vector entries. The **Vector** encoding functionality sets this flag value on the user's behalf if an entry is encoded with its own **PermData**. A decoding application can check this flag to determine whether a contained entry has **PermData** and is often useful for fan out devices (if an entry does not have **PermData**, the fan out device can likely pass on data and not worry about special permissioning for the entry). Each entry also indicates the presence of permission data via the use of **VectorEntryFlags.HAS_PERM_DATA**. Refer to Section 11.3.5.4.
VectorFlags.HAS_SUMMARY_DATA	Indicates that the **Vector** contains summary data. • If this flag is set while encoding, summary data must be provided by encoding or populating **EncodedSummaryData** with pre-encoded data. • If this flag is set while decoding, summary data is contained as part of **Vector** and the user can choose whether to decode it.
VectorFlags.HAS_SET_DEFS	Indicates that the **Vector** contains local set definition information. Local set definitions correspond to data contained in this **Vector**'s entries and are used for encoding or decoding their contents. For more information, refer to Section 11.6.
VectorFlags.SUPPORTS_SORTING	Indicates that the **Vector** may leverage sortable action types. If an **Vector** is sortable, all components must properly handle changing index values based on insert and delete actions. If a component does not properly handle these action types, it can result in the corruption of the **Vector**'s contents. For more information on proper handling, refer to Section 11.3.5.5.

Table 95: VectorFlags Values

11.3.5.3 VectorEntry Methods

Each **VectorEntry** can house other Container Types only. **Vector** is a uniform type, whereas **Vector.ContainerType** indicates the single-type housed in each entry. Each entry has an associated action which informs the user of how to apply the data contained in the entry.

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicate whether optional **VectorEntry** content is present. • For more information about **VectorEntryFlags** values, refer to Section 11.3.5.4.You can use the convenient method **ApplyHasPermData** to set specific **VectorEntryFlags**. • You can use the convenient method **CheckHasPermData** to check whether specific **VectorEntryFlags** are set.
Action	Sets or gets **Action**, which helps to manage change processing rules and informs the consumer of how to apply the entry's data. For specific information about possible **Action**'s associated with an **VectorEntry**, refer to Section 11.3.5.5.
Index	Sets or gets the entry's position (**Index**) in the **Vector**. This value can change over time based on other **VectorEntryActions**. **Index** has an allowable range of 0 to 1,073,741,823.
PermData	(Optional) Sets or gets **PermData**, which is a **Buffer** (with position and length) that specifies authorization information for this specific entry. If present, the **VectorEntryFlags.HAS_PERM_DATA** flag should be set. • For more information, refer to Section 11.4. • For more information about **VectorEntryFlags**, refer to Section 11.3.5.4. **PermData** has a maximum allowed length of 32,767 bytes.
EncodedData	Sets or gets **EncodedData**, which is a **Buffer** (with position and length) that contains this **VectorEntry**'s encoded content. • If populated using encode methods, this indicates that data is pre-encoded and **EncodedData** is copied while encoding. • If populated while decoding, this refers to this encoded **VectorEntry**'s payload and length information.
Encode	Encodes a **VectorEntry** from pre-encoded data. This method expects the same **EncodeIterator** used with **Vector.EncodeInit**. The pre-encoded vector entry payload can be passed in via **VectorEntry.EncodedData**. This method is called after **Vector.EncodeInit** and after encoding any summary data and local set definition data.
EncodeInit	Encodes a **VectorEntry** from a container type. This method expects the same **EncodeIterator** used with **Vector.EncodeInit**. After this call, housed-type encode methods can encode the contained type. This method is called after **Vector.EncodeInit** and after encoding any summary and local set definition data. To reserve space for encoding, pass in a maximum length hint value (associated with the expected maximum encoded length of this entry). If you do not know the approximate encoded set data length, you can pass in a value of **0**.
EncodeComplete	Completes the encoding of a **VectorEntry**. This method expects the same **EncodeIterator** used with **Vector.EncodeInit**, **VectorEntry.EncodeInit** and all other encoding for this container. • If encoding succeeds, the **bool success** parameter should be **true** to finish entry encoding. • If encoding fails, the **bool success** parameter should be **false** to roll back the encoding of this **VectorEntry** only.

Table 96: VectorEntry Methods

METHOD	DESCRIPTION
Decode	Decodes a **VectorEntry**. This method expects the same **DecodeIterator** used with **Vector.Decode** and populates **EncodedData** with an encoded entry. After this method returns, you can use the **Vector.ContainerType** to invoke the correct contained type's decode methods. Calling **VectorEntry.Decode** again will continue to decode subsequent entries in **Vector** until no more entries are available. As entries are received, the action will indicate how to apply their contents.
Clear	Clears this object, so that you can reuse it.  **TIP:** When decoding, you can reuse **VectorEntry** without using **Clear**.

Table 96: VectorEntry Methods (Continued)

11.3.5.4 VectorEntry Flag Enumeration Value

FLAG	MEANING
VectorEntryFlags.HAS_PERM_DATA	Indicates the presence of the **PermData** member in this container entry and indicates authorization information for this entry. For more information, refer to Section 11.4.

Table 97: VectorEntryFlags Values

11.3.5.5 VectorEntry Action Enumeration Values

ACTION	MEANING
VectorEntryActions.SET	Indicates that the consumer should set the entry at this index position. A set action typically occurs when an entry is initially provided. It is possible for multiple set actions to target the same entry. If this occurs, any previously received data associated with the entry should be replaced with the newly-added information. **VectorEntryActions.SET_ENTRY** can apply to both sortable and non-sortable vectors.
VectorEntryActions.UPDATE	Indicates that the consumer should update any previously stored or displayed information with the contents of this entry. An update action typically occurs when an entry is already set or inserted and changes to the contents are required. If an update action occurs prior to the set or insert action for the same entry, the update action should be ignored. **VectorEntryActions.UPDATE_ENTRY** can apply to both sortable and non-sortable vectors.
VectorEntryActions.CLEAR	Indicates that the consumer should remove any stored or displayed information associated with this entry's index position. **VectorEntryActions.CLEAR_ENTRY** can apply to both sortable and non-sortable vectors. No entry payload is included when the action is a 'clear.'
VectorEntryActions.INSERT	Applies only to a sortable vector. The consumer should insert this entry at the index position. Any higher order index positions are incremented by one (e.g., if inserting at index position 5 the existing position 5 becomes 6, existing position 6 becomes 7, and so forth).
VectorEntryActions.DELETE	Applies only to a sortable vector. The consumer should remove any stored or displayed data associated with this entry's index position. Any higher order index positions are decremented by one (e.g., if deleting at index position 5 the existing position 5 is removed, position 6 becomes 5, position 7 becomes 6, and so forth). No entry payload is included when the action is a 'delete.'

Table 98: VectorEntryActions Values

11.3.5.6 Vector Encoding Example

The following sample demonstrates how to encode an **Vector** containing **Series** values. The example encodes three **VectorEntry** values as well as summary data:

- The first entry is encoded from an unencoded series
- The second entry is encoded from a buffer containing a pre-encoded series and has perm data
- The third is a clear action type with no payload.

This example demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted (though it should be performed).

```
/* populate vector structure prior to call to Vector.EncodeInit() */

/* indicate that summary data and a total count hint will be encoded */
vector.ApplyHasSummaryData();
vector.ApplyHasTotalCountHint();
vector.ApplyHasPerEntryPermData();
/* populate containerType and total count hint */
vector.ContainerType = DataTypes.SERIES;
vector.TotalCountHint = 3;

/* begin encoding of vector - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
/* summary data approximate encoded length is 50 bytes */
if ((retCode = vector.EncodeInit(encIter, 50, 0 )) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Vector.EncodeInit. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
}
else
{
   /* vector init encoding was successful */
   /* create a single VectorEntry and Series and reuse for each entry */
   VectorEntry vectorEntry = new VectorEntry();
   Series series = new Series();
   /* encode expected summary data, init for this was done by Vector.EncodeInit
   - this type should match vector.ContainerType */
   {
      /* now encode nested container using its own specific encode methods */
      /* begin encoding of series - using same encIterator as vector */
      if ((retCode = series.EncodeInit(encIter, 0, 0)) < CodecReturnCode.SUCCESS)

      /*----- Continue encoding series entries. See example in Section 11.3.4.4 */

      /* Complete nested container encoding */
      retCode = series.EncodeComplete(encIter, success);
   }
   /* complete encoding of summary data. If any series entry encoding failed, success is false */
   retCode = vector.EncodeSummaryDataComplete(encIter, success);

   /* FIRST Vector Entry: encode entry from unencoded data. Approx. encoded length 90 bytes */
   /* populate index and action, no perm data on this entry */
   vectorEntry.Index = 1;
```

```
    vectorEntry.Action = VectorEntryActions.UPDATE;
    retCode = vectorEntry.EncodeInit(encIter, 90);
    /* encode contained series - this type should match vector.ContainerType */
    {
        /* now encode nested container using its own specific encode methods */
        /* clear, then begin encoding of series - using same encIterator as vector */
        series.Clear();
        if ((retCode = series.EncodeInit(encIter, 0, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding series entries. See example in Section 11.3.4.4 ---- */

        /* Complete nested container encoding */
        retCode = series.EncodeComplete(encIter, success);
    }
    retCode = vectorEntry.EncodeComplete(encIter, success);

    /* SECOND Vector Entry: encode entry from pre-encoded buffer containing an encoded Series */
    /* assuming encSeries Buffer contains the pre-encoded payload with data and length populated
    and permData contains permission data information */
    vectorEntry.Index = 2;
    /* by passing permData on an entry, the map encoding functionality will implicitly set the
    VectorFlags.HAS_PER_ENTRY_PERM_DATA flag */
    vectorEntry.ApplyHasPermData();
    vectorEntry.Action = VectorEntryActions.SET;
    vectorEntry.PermData = permData;
    vectorEntry.EncodedData = encSeries;
    retCode = vectorEntry.Encode(encIter);

    /* THIRD Vector Entry: encode entry with clear action, no payload on clear */
    /* Should clear entry for safety, this will set flags to NONE */
    vectorEntry.Clear();
    vectorEntry.Index = 3;
    vectorEntry.Action = VectorEntryActions.CLEAR;
    retCode = vectorEntry.Encode(encIter);
}

/* complete vector encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to EncodeInit */
retCode = vector.EncodeComplete(encIter, success);
```

Code Example 32: `Vector` Encoding Example

11.3.5.7 Vector Decoding Example

The following sample illustrates how to decode an **Vector** and is structured to decode each entry to the contained value. This sample code assumes the housed container type is an **Series**. Typically an application would invoke the specific container type decoder for the housed type or use a switch statement to allow a more generic series entry decoder. This example uses the same **DecodeIterator** when calling the content's decoder function. Optionally, an application could use a new **DecodeIterator** by setting the **EncodedData** on a new iterator. To simplify the sample, some error handling is omitted.

```
/* decode contents into the vector structure */
if ((retCode = vector.decode(decIter)) >= CodecReturnCode.SUCCESS)
{
    /* create single vector entry and reuse while decoding each entry */
    VectorEntry vectorEntry = new VectorEntry();
    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry)
    indicates to the Transport API that the user wants to decode summary data */
    if (vector.CheckHasSummaryData())
    {
        /* summary data is present. Its type should be that of vector.ContainerType */
        retCode = series.Decode(decIter);

        /* Continue decoding series entries. See the example in Section 11.3.4.5 */
    }
    /* decode each vector entry until there are no more left */
    while ((retCode = vectorEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCode.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            Console.WriteLine("Error ({0}) (errno: {1}) encountered with VectorEntry.Decode. Error Text:
    {2}",
                error.ErrorId, error.SysError, error.Text);
        }
        else
        {
            retCode = series.Decode(decIter);
            /* Continue decoding series entries. See example in Section 11.3.4 */
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with Vector.Decode. Error Text: {2}",
        error.ErrorId, error.SysError, error.Text);
}
```

Code Example 33: Vector Decoding Example

11.3.6 FilterList

The **FilterList** is a non-uniform container type of **FilterId**-value pair entries. Each entry, known as a **FilterEntry**, contains an **id** corresponding to one of 32 possible bit-value identifiers. These identifiers are typically defined by a domain model specification and can indicate interest in or the presence of specific entries through the inclusion of the **FilterId** in the message key's **Filter** member. A **FilterList** can contain zero to *N*[10] entries, where zero indicates an empty **FilterList**, though this type is typically limited by the number of available of **FilterId** values.

11.3.6.1 FilterList Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) to indicate presence of optional **FilterList** content. For more information about **FilterListFlags** values, refer to Section 11.3.6.2. • You can use the following convenient methods to set specific **FilterListFlags**: **ApplyHasPerEntryPermData**, **ApplyHasTotalCountHint**. • You can use the following convenient methods to check whether specific **FilterListFlags** are set: **CheckHasPerEntryPermData**, **CheckHasTotalHintCount**.
ContainerType	Sets or gets a **ContainerType**, which is a **DataType** enumeration value that, for most efficient bandwidth use, should describe the most common container type across all housed filter entries. All housed entries may match this type, though one or more entries may differ. If an entry differs, the entry specifies its own type via the **FilterEntry.ContainerType** member.
TotalCountHint	Sets or gets a four-byte unsigned integer (**TotalCountHint**) that indicates an approximate total number of entries associated with this stream. **TotalCountHint** is used typically when multiple **FilterList** containers are spread across multiple parts of a refresh message (for more information about message fragmentation and multi-part message handling, refer to Section 13.1). **TotalCountHint** is useful in determining the amount of resources to allocate for caching or displaying all expected entries. **TotalCountHint** values have a range of 0 to 1,073,741,824, though the **FilterList** is typically limited by available **filterId** values.
EncodedEntries	Returns the **EncodedEntries**, which is a Buffer (with position and length) that contains the **filterId**-value pair encoded data, if any, contained in the message. This would refer to the encoded **FilterList** payload and length information.
EncodeInit	Begins encoding a **FilterList**. **ContainerType** should define the most common entry type.
EncodeComplete	Completes the encoding of a **FilterList**. This method expects the same **EncodeIterator** used with **FilterList.EncodeInit**. • If encoding succeeds, the **bool success** parameter should be set to **true** to finish encoding. • If any entry fails to encode, the **bool success** parameter should be set to **false** to roll back to the last successfully encoded point in the contents. Encode all entries prior to this call.
Decode	Begins decoding a **FilterList**. This method decodes from the **Buffer** specified in **DecodeIterator**.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **FilterList** without using **Clear**.

Table 99: FilterList Methods

10. A **FilterList** currently has a maximum entry count of 65,535, though due to the allowable range of id values, this typically does not exceed 32. If all entry count values are allowed, this type has an approximate maximum encoded length of 4 GB but may be limited to 65,535 bytes if housed inside a container entry. The content of an **FilterEntry** has a maximum encoded length of 65,535 bytes. These limitations can change in future releases.

11.3.6.2 FilterList Flag Enumeration Values

FLAG ENUMERATION VALUES	MEANING
FilterListFlags.HAS_TOTAL_COUNT_HINT	Indicates the presence of the **TotalCountHint** member. **TotalCountHint** provides an approximation of the total number of entries sent across all filter lists on all parts of the refresh message. This information is useful in determining the amount of resources to allocate for caching or displaying all expected entries.
FilterListFlags.HAS_PER_ENTRY_PERM_DATA	Indicates some filter entries include permission information. The **FilterList** encoding functionality sets this flag value on the user's behalf if any entry is encoded with its own **PermData**. A decoding application can check this flag to determine whether any contained entry has **PermData**, often useful for fan out devices (if entries do not have **PermData**, the fan out device can pass along the data and not worry about special permissioning for an entry). Each entry will also indicate permission data presence via the use of the **FilterEntryFlags.HAS_PERM_DATA** flag. Refer to Section 11.3.6.4.

Table 100: FilterListFlags Values

11.3.6.3 FilterEntry Methods

Each **FilterEntry** can house only other container types. **FilterList** is a non-uniform type, where the **FilterList.ContainerType** should indicate the most common type housed in each entry. Entries that differ from this type must specify their own type via **FilterEntry.ContainerType**.

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicate the presence of optional **FilterEntry** content. For more information about **FilterEntryFlags**values, refer to Section 11.3.6.4. • You can use the following convenient methods to set specific **FilterEntryFlags**: **ApplyHasContainerType**, **ApplyHasPermData**. • You can use the following convenient methods to check whether specific **FilterEntryFlags** are set: **CheckHasContainerType**, **CheckHasPermData**.
Action	Sets or gets **Action**, which helps manage change processing rules and informs the consumer how to apply the information contained in the entry. For specific information about possible **Action**'s associated with an **FilterEntry**, refer to Section 11.3.6.5.
Id	Sets or gets the ID (**Id**) associated with the entry. Each possible **Id** corresponds to a bit-value that can be used with the message key's **Filter** member. This bit-value can be specified on the **Filter** to indicate interest in the **Id** when present in an **IRequestMsg** or to indicate presence of the **Id** when present in other messages. For additional information about the filter, refer to Section 12.1.2. **Id** has a range of **1** to **32**. A value of **0** is not valid as it cannot correlate to a bit-value for use with the message key filter.
ContainerType	Sets or gets **ContainerType**; a **DataType** value describing the type of this **FilterEntry**. If **ContainerType** is present, the user should set the **FilterEntry** flag (**FilterEntryFlags.HAS_CONTAINER_TYPE**). For more information about **FilterEntry** flag values, refer to Section 11.3.6.4.

Table 101: FilterEntry Methods

METHOD	DESCRIPTION
PermData	(Optional) Sets or gets **PermData**, which is a **Buffer** (with position and length) that specifies authorization information for this entry. If **PermData** is present, the user should set the **FilterEntryFlags.HAS_PERM_DATA** flag. **PermData** has a maximum allowed length of 32,767 bytes. • For more information about **FilterEntry** flag values, refer to Section 11.3.6.4. • For more information, refer to Section 11.4.
EncodedData	Sets or gets **EncodedData**, which is a **Buffer** (with position and length) containing the **FilterEntry**'s encoded content. • If populated on encode functions, **EncodedData** indicates that data is pre-encoded, and **EncodedData** will be copied while encoding. • If populated while decoding, this refers to this encoded **FilterEntry**'s payload and length information.
Encode	Encodes a **FilterEntry** from pre-encoded data. **Encode** expects the same **EncodeIterator** used with **FilterList.EncodeInit**. The pre-encoded filter entry payload can be passed in via **FilterEntry.EncodeData**. This method can be called after **FilterList.EncodeInit** completes. If this filter entry houses a type other than what is specified in **FilterList.ContainerType**, the entry's **ContainerType** should be populated to indicate the difference.
EncodeInit	Encodes a **FilterEntry** from a container type. **EncodeInit** expects the same **EncodeIterator** used with **FilterList.EncodeInit**. After this call, the housed type encode Method can begin to encode the contained type. This method can be called after **FilterList.EncodeInit** is completed. • To reserve space for encoding, pass in a maximum length hint value (associated with the expected maximum encoded length of this entry) to **FilterEntry.EncodeInit**. If you do not know the approximate encoded length, you can pass in a value of **0**. • If this filter entry houses a type other than that specified in **FilterList.ContainerType**, the entry's **ContainerType** value must indicate the difference.
EncodeComplete	Completes the encoding of a **FilterEntry**. **EncodeComplete** expects the same **EncodeIterator** used with **FilterList.EncodeInit**, **FilterEntry.EncodeInit** and all other encoding for this container. • If encoding succeeds, the **bool success** parameter should be set to **true** to finish entry encoding. • If encoding fails, the **bool success** parameter should be set to **false** to roll back the encoding of this **FilterEntry**.
Decode	Decodes a **FilterEntry**. **Decode** expects the same **DecodeIterator** used with **FilterList.Decode**. This populates **EncodedData** with an encoded entry. As an entry is received, its action indicates how to apply contents. After this method returns, the **FilterList.ContainerType** (or **FilterEntry.ContainerType** if present) can invoke the correct contained type's decode methods. Calling **FilterEntry.Decode** again decodes the remaining entries in the **FilterList**.
Clear	Clears this object, so that you can reuse it.  **TIP:** When decoding, you can reuse **FilterEntry** without using **Clear**.

Table 101: FilterEntry Methods (Continued)

11.3.6.4 FilterEntry Flag Enumeration Values

FLAG	MEANING
FilterEntryFlags.HAS_PERM_DATA	Indicates the presence of **`PermData`** in this container entry and indicates authorization information for this entry. For more information, refer to Section 11.4.
FilterEntryFlags.HAS_CONTAINER_TYPE	Indicates the presence of **`ContainerType`** in this entry. This flag is used when the entry's **`ContainerType`** differs from the specified **`FilterList.ContainerType`**.

Table 102: `FilterEntryFlags` Values

11.3.6.5 FilterEntry Action Flag Values

Each entry has an associated **`Action`** which informs the user of how to apply the entry's contents.

ACTION ENUMERATION	MEANING
FilterEntryActions.SET	Indicates that the consumer should set the entry corresponding to this **`Id`**. A set action typically occurs when an entry is initially provided. Multiple set actions can occur for the same entry **`Id`**, in which case, any previously received data associated with the entry **`Id`** should be replaced with the newly-added information.
FilterEntryActions.UPDATE	Indicates that the consumer should update any previously stored or displayed information with the contents of this entry. An update action typically occurs when an entry is set and changes to the contents need to be conveyed. An update action can occur prior to the set action for the same entry **`Id`**, in which case, the update action should be ignored.
FilterEntryActions.CLEAR	Indicates that the consumer should remove any stored or displayed information associated with this entry's **`Id`**. No entry payload is included when the action is a clear.

Table 103: `FilterEntryActions` Values

11.3.6.6 FilterList Encoding Example

The following sample illustrates how to encode an **FilterList** containing a mixture of housed types. The example encodes three **FilterEntry** values:

- The first is encoded from an unencoded element list.
- The second is encoded from a buffer containing a pre-encoded element list.
- The third is encoded from an unencoded map value.

This example demonstrates error handling only for the initial encode function, and to simplify the example, omits additional error handling (though it should be performed).

```
/* populate filterList structure prior to call to FilterList.EncodeInit() */

/* populate containerType. Because two element lists exist, this is most common so specify that type */
filterList.ContainerType = DataTypes.ELEMENT_LIST;

/* begin encoding of filterList - assumes that encIter is already populated with buffer and version
information, store return value to determine success or failure */
if ((retCode = filterList.EncodeInit(encIter)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with FilterList.EncodeInit. Error Text:
   {2}",
      error.ErrorId, error.SysError, error.Text);
}
else
{
   /* filterList init encoding was successful */
   /* create a single FilterEntry and reuse for each entry */
   FilterEntry filterEntry = new FilterEntry();

   /* FIRST Filter Entry: encode entry from unencoded data. Approx. encoded length 350 bytes */
   /* populate id and action */
   filterEntry.Id = 1;
   filterEntry.Action = FilterEntryActions.SET;
   retCode = filterEntry.EncodeInit(encIter, 350);
   /* encode contained element list */
   {
      ElementList elementList = new ElementList();
      elementList.ApplyHasStandardData();
      /* now encode nested container using its own specific encode methods */
      if ((retCode = elementList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

      /*----- Continue encoding element entries. See example in Section 11.3.2.4---- */

      /* Complete nested container encoding */
      retCode = elementList.EncodeComplete(encIter, success);
   }
   retCode = filterEntry.EncodeComplete(encIter, success);

   /* SECOND Filter Entry: encode entry from pre-encoded buffer containing an encoded element list */
   /* assuming encElemList Buffer contains the pre-encoded payload with data and length populated */
   filterEntry.Id = 2;
```

```
    filterEntry.Action = FilterEntryActions.UPDATE;
    filterEntry.EncodedData = encElemList;
    retCode = filterEntry.Encode(encIter);

    /* THIRD Filter Entry: encode entry from an unencoded map */
    filterEntry.Id = 3;
    filterEntry.Action = FilterEntryActions.UPDATE;
    /* because type is different from filterList.ContainerType, we need to specify on entry */
    filterEntry.ApplyHasContainerType();
    filterEntry.ContainerType = DataTypes.MAP;
    retCode = filterEntry.EncodeInit(encIter, 0);
    /* encode contained map */
    {
        map.KeyPrimitiveType = DataTypes.ASCII_STRING;
        map.ContainerType = DataTypes.FIELD_LIST;
        /* now encode nested container using its own specific encode methods */
        if ((retCode = map.EncodeInit(encIter, 0, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding map entries. See example in Section 11.3.3.6 ---- */

        /* Complete nested container encoding */
        retCode = map.EncodeComplete(encIter, success);
    }
    retCode = filterEntry.EncodeComplete(encIter, success);
}
/* complete filterList encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to EncodeInit */
retCode = filterList.EncodeComplete(encIter, success);
```

Code Example 34: `FilterList` Encoding Example

11.3.6.7 FilterEntry Decoding Example

The following sample illustrates how to decode an **`FilterList`** and is structured to decode each entry to its contained value. The sample code uses a switch statement to decode the contents of each filter entry. Typically an application invokes the specific container type decoder for the housed type or uses a switch statement to use a more generic series entry decoder. This example uses the same **`DecodeIterator`** when calling the content's decoder function. Optionally, an application could use a new **`DecodeIterator`** by setting the **`EncodedData`** on a new iterator. To simplify the example, some error handling is omitted.

```
/* decode contents into the filter list structure */
if ((retCode = filterList.Decode(decIter)) >= CodecReturnCode.SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    FilterEntry filterEntry = new FilterEntry();
    /* decode each filter entry until there are no more left */
    while ((retCode = filterEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCode.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            Console.WriteLine("Error ({0}) (errno: {1}) encountered with FilterEntry.Decode. Error Text:
    {2}",
                error.ErrorId, error.SysError, error.Text);
        }
        else
        {
            /* if filterEntry.containerType is present, switch on that,
```

```
          Otherwise switch on filterList.ContainerType */
          int cType;
          if (filterEntry.CheckHasContainerType())
             cType = filterEntry.ContainerType;
          else
             cType = filterList.ContainerType;
          switch (cType)
          {
             case DataTypes.MAP:
                retCode = map.Decode(decIter);

                /* Continue decoding map entries. See example in Section 11.3.3.7 */

             break;
             case DataTypes.ELEMENT_LIST:
                retCode = elemList.Decode(decIter, null);

                /* Continue decoding element entries. See example in Section 11.3.2.5 */

             break;
                /* full switch statement omitted to shorten sample code */
          }
       }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with FilterList.Decode. Error Text: {2}",
       error.ErrorId, error.SysError, error.Text);
}
```

Code Example 35: `FilterList` Decoding Example

11.3.7 Non-RWF Container Types

Enterprise Transport API messages and container entries allow non-LSEG Rssl Wire Format content. Non-LSEG Rssl Wire Format content can be:

- A specific type of formatted data such as ANSI Page or XML, where a **DataType** value aids in identifying the type.
- A type of customized, user-defined information. You can use **DataType**'s range of **225** - **255** to define custom types.

11.3.7.1 Non-RWF Encode Methods

The Transport API provides utility methods to help encode non-LSEG Rssl Wire Format types. These methods work in conjunction with **EncodeIterator** to provide appropriate encoding position and length data to the user, which can then be used with specific methods for the non-LSEG Rssl Wire Format type being encoded.

METHOD	DESCRIPTION
EncodeIterator.EncodeNonRWFInit	Uses the **EncodeIterator** to populate an **Buffer** with encoding information for the user. **Buffer.Data** contains the backing **ByteBuffer**. **Buffer.Position** contains the position where encoding begins. **Buffer.Length** contains the number of available bytes for encoding. After this method returns successfully, you can populate this buffer using non-LSEG Rssl Wire Format encode methods.
EncodeIterator.EncodeNonRWFComplete	Integrates content encoded into **Buffer** with other pre-encoded information. **Buffer.Data.Position** should be set to theposition of the last byte encoded prior to this method being called.

Table 104: Non-LSEG Rssl Wire Format Type Encode Methods

11.3.7.2 Non-RWF Encoding Example

NOTE: Do not change the value of **Buffer.Data** between calls to **EncodeIterator.EncodeNonRWFInit** and **EncodeIterator.EncodeNonRWFComplete**.

The following sample demonstrates how to encode an **Series** containing a non-RWF type of ANSI Page. This example demonstrates error handling for the initial encode method while omitting additional error handling (though it should be performed).

```
/* populate ContainerType with the ANSI dataType enumerated value; this could be any non-RWF type enum */
series.ContainerType = DataTypes.ANSI_PAGE;
/* begin encoding of series - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
if ((retCode = series.EncodeInit(encIter, 0, 0)) < CodecReturnCode.SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = false;
    /* print out message with return value string, value, and text */
    Console.WriteLine("Error ({0}) (errno: {1}) encountered with Series.EncodeInit. Error Text: {2}",
        error.ErrorId, error.SysError, error.Text);
}
else
{
    /* series init encoding was successful */
    /* begin our series entry and then nest ANSI Page inside of it using non-RWF encode methods */
    SeriesEntry seriesEntry = new SeriesEntry();
    /* create an empty buffer for information to be populated into */
    Buffer nonRWFBuffer = new Buffer();
    retCode = seriesEntry.EncodeInit(encIter, 0);
    /* encode contained non-RWF type using non-RWF encode methods */
    {
        retCode = encIter.EncodeNonRWFInit(nonRWFBuffer);
        /* now encode nested container using its own specific encode methods -
        Ensure that we do not exceed nonRWFBuffer.Length */
        /* we could copy into the nonRWFBuffer or use it with other encode methods */
        /* The encAnsiBuffer shown here is expected to be populated with data from an
        external ANSI encoder. The native ANSI encode methods could be called, instead
        of a copy with pre-encoded ANSI content, to directly encode into the nonRWFBuffer */
        nonRWFBuffer.Data().Put(encAnsiBuffer.Data());
        retCode = encIter.EncodeNonRWFComplete(nonRWFBuffer, success);
    }
    retCode = seriesEntry.EncodeComplete(encIter, success);
}
/* complete series encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to EncodeInit */
retCode = series.EncodeComplete(encIter, success);
```

Code Example 36: Non-RWF Type Encoding Example

11.3.7.3 Decoding Non-RWF Types

When decoding, the user can obtain non-RWF data via the **EncodedData** member and use this with methods specific to the non-RWF type being decoded.

11.4 Permission Data

Permission Data is optional authorization information. The DACS Lock API provides functionality for creating and manipulating permissioning information. For more information on Data Access Control System usage and permission data creation, refer to the *Enterprise Transport API DACS LOCK Library Reference Manual*.

Permission data can be specified in some messages. When permission data is included in an **`IRefreshMsg`** or an **`IStatusMsg`**, this generally defines authorization information associated with all content on the stream. You can change permission data on an existing stream by sending a subsequent **`IStatusMsg`** or **`IRefreshMsg`** which contains the new permission data. When permission data is included in an **`IUpdateMsg`**, this generally defines authorization information that applies only to that specific **`IUpdateMsg`**.

Permission data can also be specified in some container entries. When a container entry includes permission data, it generally defines authorization information that applies only to that specific container entry. Specific usage and inclusion of permissioning information can be further defined within a domain model specification.

Permission data typically ensures that only entitled parties can access restricted content. On LSEG Real-Time Distribution System, all content is restricted (or filtered) based on user permissions.

When content is contributed, permission data in an **`IPostMsg`** is used to permission the user who posts the information. If the payload of the **`IPostMsg`** is another message type with permission data (i.e., **`IRefreshMsg`**), the nested message's permissions can change the permission expression associated with the posted item. If permission data for the nested message is the same as permission data on the **`IPostMsg`**, the nested message does not need permission data.

11.5 Summary Data

Some Enterprise Transport APIcontainer types allow summary data. ***Summary data*** conveys information that applies to every entry housed in the container. Using summary data ensures data is sent only once, instead of repetitively including data in each entry. An example of summary data is the currency type because it is likely that all entries in the container share the same currency. Summary data is optional and applications can determine when to employ it.

Specific domain model definitions typically indicate whether summary data should be present, along with information on its content. When included, the **`ContainerType`** of the summary data is expected to match the **`ContainerType`** of the payload information (e.g., if summary data is present on a **`Vector`**, the **`Vector.ContainerType`** defines the type of summary data and **`VectorEntry`** payload).

11.6 Set Definitions and Set-Defined Data

A ***Set-Defined Primitive Type*** is similar to a primitive type (described in) with several key differences. While primitive types can be encoded as a variable number of bytes, most set-defined primitive types use a fixed-length encoding. Fixed-length encoding can help reduce the number of bytes required to contain the encoded primitive type. **`DataType`** values between **64** and **127** are set-defined primitive types and set fixed-length encodings for many base primitive types (e.g., **`DataTypes.INT_1`** is a one-byte fixed-length encoding of **`DataTypes.INT`**). Whereas all primitive types can represent blank data, only several set-defined primitive types can do so. All encoding and decoding continues to use primitive type definitions and should continue to function in the same manner as described in the previous sections. The **`DataTypes`** enumeration exposes values that define each set-defined primitive, though these values are only used inside of a set definition. When using set-defined primitive types, a set definition is required to encode or decode content.

A ***Set Definition*** can define the contents of an **`FieldList`** or an **`ElementList`** and allow additional optimizations. Use of a set definition can reduce overall encoded content by eliminating repetitive type and length information.

- A set definition describing an **`FieldList`** contains **`FieldId`** and type information specified in the same order as the contents are arranged in the encoded field list.
- A set definition describing an **`ElementList`** contains element **`Name`** and type information specified in the same order as the contents are arranged in the encoded element list.

When encoding, in addition to providing set definition information, an application encodes the field list or element list content. Internally the encoder uses the provided set definition to perform type encoding specific to the definition and omit redundant information needed only in the definition.

When decoding, in addition to providing set definition information, an application decodes the field list or element list content. Internally, the decoder uses the provided set definition to decode any type-specific optimizations and to reintroduce redundant information omitted during the encoding.

Instead of including multiple instances of the same content, you can use a set definition (i.e., an **`Map`** containing **`FieldList`** content in each entry). In this case, a set definition can be provided once as part of the **`Map`** to define the layout of repetitive field list information contained in the **`MapEntry`** (i.e., **`FieldId`**). When encoding each **`FieldList`**, this content will be omitted because it is included in the set definition.

A set definition can contain primitive type enumerations (), set-defined primitive type enumerations, and container type enumerations (). Encoding and decoding occurs exactly the same as primitive type and container type encoding or decoding.

11.6.1 Set-Defined Primitive Types

Set primitive types do not use separate interface methods for encoding or decoding. Decoding uses the same primitive type decoder used when decoding the primitive type. Because these types can only be contained in an **`FieldList`** or **`ElementList`**, encoding occurs as usual by calling **`FieldEntry.Encode`** or **`ElementEntry.Encode`**. When calling these methods, populate the field or element entry using the base primitive type. The table below provides a brief description of each set-defined primitive type, along with its corresponding base primitive type enumeration and its respective decode interface.

	BASE PRIMITIVE TYPE	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
DataTypes.INT_1	DataTypes.INT	Int	Int.Decode	A signed, one-byte integer type that represents a value up to 7 bits with a one-bit sign (positive or negative). Allowable range is (-2^7) to $(2^7 - 1)$. This type cannot be represented as blank.
DataTypes.INT_2	DataTypes.INT	Int	Int.Decode	A signed, two-byte integer type that represents a value up to 15 bits with a one-bit sign (positive or negative). Allowable range is (-2^{15}) to $(2^{15} - 1)$. This type cannot be represented as blank.
DataTypes.INT_4	DataTypes.INT	Int	Int.Decode	A signed, four-byte integer type that represents a value up to 31 bits with a one-bit sign (positive or negative). Allowable range is (-2^{31}) to $(2^{31} - 1)$. This type cannot be represented as blank.
DataTypes.INT_8	DataTypes.INT	Int	Int.Decode	A signed, eight-byte integer type that represents a value up to 63 bits with a one-bit sign (positive or negative). Allowable range is (-2^{63}) to $(2^{63} - 1)$. This type cannot be represented as blank.
DataTypes.UINT_1	DataTypes.UINT	UInt	UInt.Decode	An unsigned, one-byte integer type that represents an unsigned value with precision of up to 8 bits. Allowable range is 0 to $(2^8 - 1)$. This type cannot be represented as blank.
DataTypes.UINT_2	DataTypes.UINT	UInt	UInt.Decode	An unsigned, two-byte integer type that represents an unsigned value with precision of up to 16 bits. Allowable range is 0 to $(2^{16} - 1)$. This type cannot be represented as blank.
DataTypes.UINT_4	DataTypes.UINT	UInt	UInt.Decode	An unsigned, four-byte integer type that represents an unsigned value with precision of up to 32 bits. Allowable range is 0 to $(2^{32} - 1)$. This type cannot be represented as blank.

Table 105: Set-Defined Primitive Types

	BASE PRIMITIVE TYPE	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
DataTypes.UINT_8	DataTypes.UINT	UInt	UInt.Decode	An unsigned, eight-byte integer type that represents an unsigned value with precision of up to 64 bits. Allowable range is 0 to (2^{64} - 1). This set-defined primitive type cannot be represented as blank.
DataTypes.FLOAT_4	DataTypes.FLOAT	Float	Float.Decode	A four-byte, floating point type that represents the same range of values allowed by the system **`float`** type. Follows the IEEE 754 specification. This type cannot be represented as blank.
DataTypes.DOUBLE_8	DataTypes.DOUBLE	Double	Double.Decode	An eight-byte, floating point type that represents the same range of values allowed by the system **`double`** type. Follows the IEEE 754 specification. This type cannot be represented as blank.
DataTypes.REAL_4RB	DataTypes.REAL	Real	Real.Decode	An optimized Rssl Wire Format representation of a decimal or fractional value which typically requires less bytes on the wire than **`float`** or **`double`** types. This type allows up to a four-byte value, with a hint value (for converting to decimal or fractional representation), which can add or remove up to seven trailing zeros, ten decimal places, or fractional denominators up to 256. Allowable range is (-2^{31}) to (2^{31} - 1) This type can be represented as blank. For more details on this type, refer to Section 11.2.2.
DataTypes.REAL_8RB	DataTypes.REAL	Real	Real.Decode	An optimized Rssl Wire Format representation of a decimal or fractional value which typically requires less bytes on the wire than float or double types. This type allows up to an eight byte value, with a hint value (for converting to decimal or fractional representation), which can add or remove up to seven trailing zeros, 14 decimal places, or fractional denominators up to 256. Allowable range is (-2^{63}) to (2^{63} - 1) This type can be represented as blank. For more details on this type, refer to Section 11.2.2.
DataTypes.DATE_4	DataTypes.DATE	Date	Date.Decode	Representation of a date containing month, day, and year values. This value can be represented as blank. For more details on this type, refer to Section 11.2.3.
DataTypes.TIME_3	DataTypes.TIME	Time	Time.Decode	Representation of a time containing hour, minute, and second values. This value can be represented as blank. For more details on this type, refer to Section 11.2.4.

Table 105: Set-Defined Primitive Types (Continued)

	BASE PRIMITIVE TYPE	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
DataTypes.TIME_5	DataTypes.TIME	Time	Time.Decode	Representation of a time containing hour, minute, second, and millisecond values. This value can be represented as blank. For more details on this type, refer to Section 11.2.4.
DataTypes.DATETIME_7	DataTypes.DATETIME	DateTime	DateTime.Decode	Combined representation of date and time. Contains all members of **`DataTypes.DATE`** and hour, minute, and second from **`DataTypes.TIME.`** This value can be represented as blank. For more details on this type, refer to Section 11.2.5.
DataTypes.DATETIME_9	DataTypes.DATETIME	DateTime	DateTime.Decode	Combined representation of date and time. Contains all members of **`DataTypes.DATE`** and all members of **`DataTypes.TIME`**. This value can be represented as blank. For more details on this type, refer to Section 11.2.5.

Table 105: Set-Defined Primitive Types (Continued)

11.6.2 Set Definition Use

In the Enterprise Transport API, an application can leverage local set definitions. A ***local set definition*** is a set definition sent along with the content it defines. Local set definitions are valid only within the scope of the container of which they are a part and apply only to the information in the container on which they are specified (e.g., a **`Map`**'s set definition content applies only to the payload within the map's entries). Set definitions are divided into two concrete types

- ***Field set definition***: A set definition that defines **`FieldList`** content
- ***Element set definition***: A set definition that defines **`ElementList`** content

Set definitions can contain multiple entries, each defining a specific encoding type for a **`FieldEntry`** or **`ElementEntry`**.

11.6.2.1 FieldSetDef Methods

The following table defines **`FieldSetDef`** Methods. **`FieldSetDef`** represents a single field set definition and can define the contents of multiple entries in an **`FieldList`**.

METHOD	DESCRIPTION
SetId	Sets or gets the field set definition's identifier value (**`SetId`**). Any field list content that leverages this definition should have **`FieldList.SetId`** match this identifier. **`SetId`** values have an allowed range of 0 to 32,767. However, only values **0** to **15** are valid for local set definition content. For more information, refer to Section 11.6. For more details on how **`FieldList`** indicates the use of a set definition, refer to Section 11.3.1
Count	Sets or gets the number (**`Count`**) of **`FieldSetDefEntry`**s contained in this definition. Each entry defines how a **`FieldEntry`** is encoded or decoded. A set definition is limited to 255 entries. For more information, refer to Section 11.6.2.2
Entries	Sets or gets entries, which is an array of **`FieldSetDefEntry`**s. Each entry defines how an **`FieldEntry`** is encoded or decoded. For more information, refer to Section 11.6.2.2.
Clear	Clears this object, so that you can reuse it.  **TIP:** When decoding, you can reuse **`FieldSetDef`** without using **`Clear`**.

Table 106: `FieldSetDef` Methods

11.6.2.2 FieldSetDefEntry Methods

METHOD	DESCRIPTION
FieldId	Set or get the **FieldId** value that corresponds to this entry in the set-defined **FieldList** content. **FieldId** is a signed, two-byte value that refers to specific name and type information defined by an external field dictionary, such as the **RDMFieldDictionary**. Negative **FieldId** values typically refer to user-defined values while positive **FieldId** values typically refer to LSEG-defined values. When encoding, the **FieldEntry.FieldId** should match the value that the set definition expects. When decoding, the **FieldEntry.FieldId** is populated with the **FieldId** value indicated in the set definition. **FieldId** has an allowable range of -32,768 to 32,767 where positive values are LSEG-defined and negative values are user-defined. The **FieldId** value of **0** is reserved to indicate **DictionaryId** changes, where the type of **FieldId 0** is an **Int**.
DataType	Set or get the **DataType**, which defines the **DataType** value of the entry as it encodes or decodes when using this set definition. This can be a base primitive type, a set-defined primitive type, or a container type. • While encoding, populate the **FieldEntry.DataType** with the base primitive type or container type value that corresponds to the type contained in this definition. • While decoding, **FieldEntry.DataType** is populated with the specific **DataType** information as indicated by the Set Definition, where any set-defined primitive type is converted to the corresponding base primitive type. For a map of set-defined primitive types and their corresponding base primitive types, refer to Section 11.6.1.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **FieldSetDefEntry** without using **Clear**.

Table 107: FieldSetDefEntry Methods

11.6.2.3 ElementSetDef Methods

The following table defines **ElementSetDef** Methods. **ElementSetDef** represents a single element set definition, and can define content for multiple entries in an **ElementList**.

METHOD	DESCRIPTION
SetId	Sets or gets the field set definition's identifier value (**SetId**). Any element list content that leverages this definition should have the **ElementList.SetId** matching this identifier. Though **SetId** values have an allowed range of 0 to 32,767, the only values valid for local set definition content are **0** - **15**. These indicate locally defined set definition use. For more information, refer to Section 11.6. For more information about how an **ElementList** indicates use of a set definition, refer to Section 11.3.2.
Count	Sets or gets the **count**, which is the number of **ElementSetDefEntry**s contained in this definition. Each entry defines how to encode or decode an **ElementEntry**. A set definition is limited to 255 entries. For more information, refer to Section 11.6.2.4.
Entries	Set or get entries, which is an array of **ElementSetDefEntry**s. Each entry defines how to encode or decode an **ElementEntry**. For more information, refer to Section 11.6.2.4.

Table 108: ElementSetDef Methods

METHOD	DESCRIPTION
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **ElementSetDef** without using **Clear**.

Table 108: ElementSetDef Methods (Continued)

11.6.2.4 ElementSetDefEntry Methods

METHOD	DESCRIPTION
Name	Sets or gets the **Name**, which is a **Buffer** (with position and length) that corresponds to this set-defined element. Element names are defined outside of the Transport API, typically as part of a domain model specification or dictionary. When encoding, you can optionally populate **ElementEntry.Name** with the **Name** expected in the set definition. If **Name** is not used, validation checking is not provided and information might be encoded that does not properly correspond to the definition. When decoding, **ElementEntry.Name** is populated with the information indicated in the set definition. The **Name** buffer allows content length ranging from 0 bytes to 32,767 bytes.
DataType	Sets or gets **DataType**. When encoding or decoding an entry using this set definition, **DataType** defines the entry's **DataTypes**. This can be a base primitive type, a set-defined primitive type, or a container type. • While encoding, populate **ElementEntry.DataType** with the base primitive type or container type value that corresponds to the type contained in this definition. • While decoding, populate **ElementEntry.DataType** with the specific **DataType** information as indicated by set definition, where any set-defined primitive type is converted to the corresponding base primitive type. For a map of set-defined primitive types and their corresponding base primitive types, refer to Section 11.6.1.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, you can reuse **ElementSetDefEntry** without using **Clear**.

Table 109: ElementSetDefEntry Methods

11.6.3 Set Definition Database

A ***set definition database*** can group definitions together. Using a database can be helpful when the content leverages multiple definitions; the database provides an easy way to pass around all set definitions necessary to encode or decode information. For instance, a **Vector** can contain multiple set definitions via a set definition database with the contents of each **VectorEntry** requiring a different definition from the database.

11.6.3.1 LocalFieldSetDefDb Methods

LocalFieldSetDefDb represents multiple local field set definitions and uses the following Methods.

METHOD	DESCRIPTION
Definitions	Returns an array containing up to fifteen **FieldSetDef**s. Each contained field set definition defines a unique **SetId** for use in the container. This memory is created by the Transport API and should not be overwritten otherwise a garbage collection (GC) will occur. For suggested use, refer to the encoding example in Section 11.6.3.5
Entries[a]	A **FieldSetDefEntry** that helps manage memory associated with set definition entries for each **FieldSetDef**. This memory is created by the Enterprise Transport API and should not be overwritten or a GC will occur. • When decoding, the Transport API assigns entries to definitions, according to the Set Definitions being decoded. • When encoding, you can assign entries to definitions, according to the Set Definitions being encoded. Refer to the encoding example in Section 11.6.3.5 for suggested use.
Clear	Clears this object, so that you can reuse it.

Table 110: LocalFieldSetDefDb Methods

a. If an application uses multiple **DecodeIterator** structures in the same thread, where each decode iterator requires the use of a local set definition database, the application must provide the memory into which entries decode.

11.6.3.2 LocalElementSetDefDb Methods

LocalElementSetDefDb (which represents multiple local element set definitions) has the following methods:

METHOD	DESCRIPTION
Definitions	An array containing up to fifteen **ElementSetDef**s. Each contained element set definition defines a unique **SetId** for use within the container on which this is present. Refer to the encoding example in Section 11.6.3.7 for suggested use. **WARNING!** This memory is created by the Enterprise Transport API. Do not overwrite this memory or a GC will occur.
Entries[a]	An **ElementSetDefEntry** that helps manage memory associated with set definition entries for each **ElementSetDef**. • When decoding, the Enterprise Transport API assigns entries to definitions, according to the Set Definitions being decoded. • When encoding, the user can assign entries to definitions, according to the Set Definitions being encoded. Refer to the encoding example in Section 11.6.3.7 for suggested use. **WARNING!** This memory is created by the Enterprise Transport API. Do not overwrite this memory or a GC will occur.
Clear	Clears this object, so that you can reuse it.

Table 111: LocalElementSetDefDb Methods

a. Within the same thread, if an application is using multiple **DecodeIterator** structures, where each decode iterator requires the use of a local set definition database, the application must provide entries memory for decoding into.

11.6.3.3 Local Set Definition Database Encoding Interfaces

Applications can send or receive local set definitions while using the **Map**, **Vector**, or **Series** container types. To provide local set definition information, an application can use the **EncodedSetDefs** method with a pre-encoded set definition database, or encode this using the Enterprise Transport API-provided methods described in this section.

The following table describes all available encoding methods required to provide set definition database content on a **Map**, **Vector**, or **Series**. When present, this information should apply to any **FieldList** or **ElementList** content within the types' entries. When encoding set-defined field or element list content, the application must pass **LocalFieldSetDefDb** or **LocalElementSetDefDb** into the **FieldList.EncodeInit** and **ElementList.EncodeInit** methods.

ENCODE INTERFACE	DESCRIPTION
LocalFieldSetDefDb.Encode	Encodes a non-pre-encoded local field set definition database into its own buffer for use with **EncodedSetDefs** or directly into a **Map**, **Vector**, or **Series**. After the container's **EncodeInit** method, local set definition encoding is expected prior to any summary data or container entries.
LocalElementSetDefDb.Encode	Encodes a non-pre-encoded local element set definition database into its own buffer for use with **EncodedSetDefs** or directly into a **Map**, **Vector**, or **Series**. After the containers **EncodeInit** method, local set definition encoding is expected prior to any summary data or container entries.
Map.EncodeSetDefsComplete	Completes encoding non-pre-encoded element or field set definition database content. This applies to local set definition database content on a **Map**, refer to Section 11.3.3.
Series.EncodeSetDefsComplete	Completes encoding non-pre-encoded element or field set definition database content. This applies to local set definition database content on a **Series**, refer to Section 11.3.4.
Vector.EncodeSetDefsComplete	Completes encoding non-pre-encoded element or field set definition database content. This applies to local set definition database content on a **Vector**, refer to Section 11.3.5.

Table 112: Local Set Definition Database Encode Methods

11.6.3.4 Local Set Definition Database Decoding Interfaces

The following table describes decoding methods for use with a local set definition database. When decoding set-defined content, the application can pass the **LocalFieldSetDefDb** or **LocalElementSetDefDb** into the **FieldList.Decode** and **ElementList.Decode** methods. If this information is not provided, Enterprise Transport API skips decoding set-defined content.

DECODE INTERFACE	DESCRIPTION
LocalFieldSetDefDb.Decode	Decodes **EncodedSetDefs** into a local field set definition database for use when decoding contained **FieldList** information.
LocalElementSetDefDb.Decode	Decodes **EncodedSetDefs** into a local field set definition database for use when decoding contained **ElementList** information.

Table 113: Local Set Definition Database Decode Methods

11.6.3.5 Field Set Definition Database Encoding Example

The following example demonstrates encoding of a field set definition database into an **Map**. The field set definition database contains one definition, made up of three field set definition entries. After set-defined content encoding is completed, an additional standard data field entry is encoded.

```
/* Create the fieldSetDefDb */
LocalFieldSetDefDb fieldSetDefDb = new LocalFieldSetDefDb();

/* create entries arrays */
FieldSetDefEntry[] fieldSetDefEntries = new FieldSetDefEntry[3];

/* Contains BID as Real */
fieldSetDefEntries[0] = new FieldSetDefEntry();
fieldSetDefEntries[0].DataType = DataTypes.REAL;
fieldSetDefEntries[0].FieldId = 22;

/* Contains ASK as an optimized Real */
fieldSetDefEntries[1] = new FieldSetDefEntry();
fieldSetDefEntries[1].DataType = DataTypes.REAL_8RB;
fieldSetDefEntries[1].FieldId = 25;

/* Contains TRADE TIME as an optimized Time */
fieldSetDefEntries[2] = new FieldSetDefEntry();
fieldSetDefEntries[2].DataType = DataTypes.TIME_3;
fieldSetDefEntries[2].FieldId = 18;

/* Now populate the entries into the set definition Db. If there were more than one definition, all
   required defs would be populated into the same Db */
/* Structure must be cleared first */
fieldSetDefDb.Clear();
/* set the definition into the slot that corresponds to its ID */
/* since this definition is ID 5, it goes into definitions array position 5 */
fieldSetDefDb.Definitions[5].SetId = 5;
fieldSetDefDb.Definitions[5].Count = 3;
fieldSetDefDb.Definitions[5].Entries = fieldSetDefEntries;

/* begin encoding of map that will contain set def DB - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
map.ApplyHasSetDefs();
map.ContainerType = DataTypes.FIELD_LIST;
map.KeyPrimitiveType = DataTypes.UINT;
if ((retCode = map.EncodeInit(encIter, 0, 0 )) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Map.EncodeInit. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
}
else
{
   /* map init encoding was successful */

   /* It expects the local set definition database to be encoded next */
   /* because we are encoding a local field set definition database, we have to call the correct method
   */
   retCode = fieldSetDefDb.Encode(encIter);
   /* Our set definition db is now encoded into the map, we must complete the map portion of this
```

```
    encoding and then begin encoding entries */
    retCode = map.EncodeSetDefsComplete(encIter, true);
    /* begin encoding of map entry - this contains a field list using the set definition encoded above */
    mapEntry.Action = MapEntryActions.ADD;
    uInt.Value(100212); /* populate map entry key */
    retCode = mapEntry.EncodeInit(encIter, uInt, 0);
    /* set field list flags - this has a setId and set defined data - we can also have standard data after
       set defined data is encoded */
    fieldList.ApplyHasSetId();
    fieldList.ApplyHasSetData();
    fieldList.ApplyHasStandardData();
    fieldList.SetId = 5; /* this field list will use the set definition from above */
    /* when encoding set defined data, the database containing the necessary definitions must be passed
       in */
    retCode = fieldList.EncodeInit(encIter, fieldSetDefDb, 0);
    /* for each field entry we encode that is set defined, the Transport API encoder verifies that the
    correct fieldId and content type are passed in. Order must match definition */

    /* Encode FIRST field in set definition */
    fieldEntry.FieldId = 22; /* fieldId of the first set definition entry */
    fieldEntry.DataType = DataTypes.REAL; /* base primitive type of the first set definition entry */
    real.Value(227, RealHints.EXPONENT_2);
    /* encode the first entry - this matches the fieldId and type specified in the first definition entry
       */
    retCode = fieldEntry.Encode(encIter, real);

    /* Encode SECOND field in set definition */
    fieldEntry.FieldId = 25; /* fieldId of the second set definition entry */
    fieldEntry.DataType = DataTypes.REAL; /* base primitive type of the second set definition entry */
    real.Value(22801, RealHints.EXPONENT_4);
    /* encode the second entry - this matches the fieldId and type specified in the first definition
       entry */
    retCode = fieldEntry.Encode(encIter, real);

    /* Encode THIRD field in set definition */
    fieldEntry.FieldId = 18; /* fieldId of the third set definition entry */
    fieldEntry.DataType = DataTypes.TIME; /* base primitive type of the third set definition entry */
    time.Hour(8);
    time.Minute(39);
    time.Second(24);
    /* encode the third entry - this matches the fieldId and type specified in the first definition entry
       */
    retCode = fieldEntry.Encode(encIter, time);

    /* Encode standard data after field set definition is complete */
    fieldEntry.FieldId = 2; /* fieldId of the first standard data entry after set definition is
       complete*/
    fieldEntry.DataType = DataTypes.UINT; /* base primitive type of the first set definition entry */
    /* encode the standard data in the message after set data is complete */

    retCode = fieldEntry.Encode(encIter, uInt);
    /* complete encoding of the content */
    retCode = fieldList.EncodeComplete(encIter, true);
    retCode = mapEntry.EncodeComplete(encIter, true);
    retCode = map.EncodeComplete(encIter, true);
}
```

Code Example 37: Field Set Definition Database Encoding Example

11.6.3.6 Field Set Definition Database Decoding Example

The following example illustrates how to decode a field set definition database from a **Map**. After decoding the database, it can be passed in while decoding **FieldList** content.

```
/* Decode the map */
retCode = map.Decode(decIter);
/* If the map flags indicate that set definition content is present, decode the set def db */
if (map.CheckHasSetDefs())
{
    /* must ensure it is the correct type - if map contents are field list, this is a field set definition
    db */
    if (map.ContainerType == DataTypes.FIELD_LIST)
    {
        fieldSetDefDb.Clear();
        retCode = fieldSetDefDb.Decode(decIter);
    }
    /* If map contents are an element list, this is an element set definition db */
    if (map.ContainerType == DataTypes.ELEMENT_LIST)
    {
        /* this is an element list set definition db */
    }
}
/* decode map entries */
while ((retCode = mapEntry.Decode(decIter, uInt)) != CodecReturnCode.END_OF_CONTAINER)
{
    if (retCode < CodecReturnCode.SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        Console.WriteLine("Error ({0}) (errno: {1}) encountered with MapEntry.Decode. Error Text: {2}",
            error.ErrorId, error.SysError, error.Text);
    }
    else
    {
        /* entries contain field lists - since there were definitions provided they should be passed
        in for field list decoding. Any set defined content will use the definition when
        decoding. If set definition db is not passed in, any set content will not be decoded */
        retCode = fieldList.Decode(decIter, fieldSetDefDb);
        /* Continue decoding field entries. See example in Section 11.3.1.6 */
    }
}
```

Code Example 38: Field Set Definition Database Decoding Example

11.6.3.7 Element Set Definition Database Encoding Example

The following example illustrates how to encode an element set definition database into an **Series**. The database contains one element set definition with three element set definition entries. After encoding is completed, the sample encodes an additional standard data element entry.

```
/* Create the elementSetDefDb and element set definition */
LocalElementSetDefDb elementSetDefDb = new LocalElementSetDefDb();
/* create entries arrays */
ElementSetDefEntry[] elementSetDefEntries = new ElementSetDefEntry[3];

/* Contains BID as a Real */
elementSetDefEntries[0] = new ElementSetDefEntry();
elementSetDefEntries[0].DataType = DataTypes.REAL;
Buffer bidBuffer = new Buffer();
bidBuffer.Data("BID");
elementSetDefEntries[0].Name = bidBuffer;

/* Contains ASK as an optimized Real */
elementSetDefEntries[1] = new ElementSetDefEntry();
elementSetDefEntries[1].DataType DataTypes.REAL_8RB;
Buffer askBuffer = new Buffer();
askBuffer.Data("ASK");
elementSetDefEntries[1].Name = askBuffer;

/* Contains TRADE TIME as an optimized Time */
elementSetDefEntries[2] = new ElementSetDefEntry();
elementSetDefEntries[2].DataType = DataTypes.TIME_3;
Buffer tradeTimeBuffer = new Buffer();
tradeTimeBuffer.Data("TRADE TIME");
elementSetDefEntries[2].Name = tradeTimeBuffer;

/* Now populate the entries into the set definition Db. If there were more than one definition,
all required defs would be populated into the same Db */
/* Structure must be cleared first */
elementSetDefDb.Clear();
/* set the definition into the slot that corresponds to its ID */
/* since this definition is ID 10, it goes into definitions array position 10 */
elementSetDefDb.Definitions[10].SetId = 10;
elementSetDefDb.Definitions[10].Count = 3;
elementSetDefDb.Definitions[10].Entries = elementSetDefEntries;

/* begin encoding of series that will contain set def DB - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
series.ApplyHasSetDefs();
series.ContainerType = DataTypes.ELEMENT_LIST;
if ((retCode = series.EncodeInit(encIter, 0, 0)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error ({0}) (errno: {1}) encountered with Series.EncodeInit. Error Text: {2}",
      error.ErrorId, error.SysError, error.Text);
}
else
```

```
{
    /* series init encoding was successful */
    SeriesEntry seriesEntry = new SeriesEntry();
    ElementList elementList = new ElementList();
    ElementEntry elementEntry = new ElementEntry();

    /* It expects the local set definition database to be encoded next */
    /* because we are encoding a local element set definition database, we have to call the correct
    method */
    retCode = elementSetDefDb.Encode(encIter);
    /* Our set definition db is now encoded into the series, we must complete the series portion of this
    encoding and then begin encoding entries */
    retCode = series.EncodeSetDefsComplete(encIter, true);
    /* begin encoding of series entry - this contains an element list using the set definition encoded
    above */
    retCode = seriesEntry.EncodeInit(encIter, 0);
    /* set element list flags - this has a setId and set defined data - we can also have standard data
    after set defined data is encoded */
    elementList.ApplyHasSetId();
    elementList.ApplyHasSetData();
    elementList.ApplyHasStandardData();
    elementList.SetId = 10; /* this element list will use the set definition from above */
    /* when encoding set defined data, the database containing the necessary definitions must be passed
    in */
    retCode = elementList.EncodeInit(encIter, elementSetDefDb, 0);
    /* for each element entry we encode that is set defined, the Transport API encoder verifies that the
    correct element name and content type are passed in. Order must match definition */

    /* Encode FIRST element in set definition */
    elementEntry.Name = bidBuffer; /* name of the first set definition entry */
    elementEntry.DataType = DataTypes.REAL; /* base primitive type of the first set definition entry */
    real.Value(227, RealHints.EXPONENT_2);
    /* encode the first entry - this matches the name and type specified in the first definition entry */
    retCode = elementEntry.Encode(encIter, real);

    /* Encode SECOND element in set definition */
    elementEntry.Name = askBuffer; /* name of the second set definition entry */
    elementEntry.DataType = DataTypes.REAL; /* base primitive type of the second set definition entry */
    real.value(22801, RealHints.EXPONENT_4);
    /* encode the second entry - this matches the name and type specified in the second definition entry
    */
    retCode = elementEntry.encode(encIter, real);

    /* Encode THIRD field in set definition */
    elementEntry.Name = tradeTimeBuffer; /* name of the third set definition entry */
    elementEntry.DataType = DataTypes.TIME; /* base primitive type of the third set definition entry */
    time.Hour(8);
    time.Minute(39);
    time.Second(24);
    /* encode the third entry - this matches the name and type specified in the third definition entry */
    retCode = elementEntry.Encode(encIter, time);
```

```
    /* Encode standard data after element set definition is complete */
    Buffer displayTemplateBuffer = new Buffer();
    displayTemplateBuffer.Data("DISPLAYTEMPLATE");
    elementEntry.Name = displayTemplateBuffer; /* name of the first standard data entry after
    set definition is complete*/
    elementEntry.DataType = DataTypes.UINT; /* base primitive type of the first set definition entry */
    uInt.Value(2112);
    /* encode the standard data in the message after set data is complete */
    retCode = elementEntry.Encode(encIter, uInt);

    /* complete encoding of the content */
    retCode = elementList.EncodeComplete(encIter, true);
    retCode = seriesEntry.EncodeComplete(encIter, true);
    retCode = series.EncodeComplete(encIter, true);
}
```

Code Example 39: Element Set Definition Database Encoding Example

11.6.3.8 Element Set Definition Database Decoding Example

The following example illustrates how to decode an element set definition database from an **Series**. After decoding the database, it can be passed in while decoding **ElementList** content.

```
/* Decode the series */
retCode = series.Decode(decIter);
/* If the series flags indicate that set definition content is present, decode the set def db */
if (series.CheckHasSetDefs())
{
    /* must ensure it is the correct type - if series contents are element list,
    this is an element set definition db */
    if (series.ContainerType == DataTypes.ELEMENT_LIST)
    {
        elementSetDefDb.Clear();
        retCode = elementSetDefDb.Decode(decIter);
    }
    /* If map contents are an field list, this is a field set definition db */
    if (series.ContainerType == DataTypes.FIELD_LIST)
    {
        /* this is a field list set definition db */
    }
}

/* decode series entries */
while ((retCode = seriesEntry.Decode(decIter)) != CodecReturnCode.END_OF_CONTAINER)
{
    if (retCode < CodecReturnCode.SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        Console.WriteLine("Error ({0}) (errno: {1}) encountered with SeriesEntry.Decode. Error Text:
    {2}",
            error.ErrorId, error.SysError, error.Text);
    }
    else
    {
        /* entries contain element lists - since there were definitions provided they should be passed
        in for element list decoding. Any set defined content will use the definition when
        decoding. If set definition db is not passed in, any set content will not be decoded */
        retCode = elementList.Decode(decIter, elementSetDefDb);
        /* Continue decoding element entries. See example in Section 11.3.2.5 */
    }
}
```

Code Example 40: Element Set Definition Database Decoding Example

12 Message Package Detailed View

12.1 Concepts

Messages communicate data between system components: to exchange information, indicate status, permission users and access, and for a variety of other purposes. Many messages have associated semantics for efficient use in market data systems to request information, respond to information, or provide updated information. Other messages have relatively loose semantics, allowing for a more dynamic use either inside or outside market data systems.

An individual flow of related messages within a connection is typically referred to as a ***stream***, and the message package allows multiple simultaneous streams to coexist in a connection. An information stream is instantiated between a consuming application and a providing application when the consumer issues an **`IRequestMsg`** followed by the provider responding with an **`IRefreshMsg`** or **`IStatusMsg`**. At this point the stream is established and allows other messages to flow within the stream. The remainder of this chapter discusses streams, stream identification, and stream uniqueness.

The Message Package offers a suite of message header definitions; each optimized to communicate a specific set of information. There are constructs to allow for communication stream identification and to determine uniqueness of streams within a connection. The following sections describe the various constructs, concepts, and processes involved with use of the Message Package.

12.1.1 Common Message Interface

Each Enterprise Transport API message consists of both unique members and common message members. The common members form the **`Msg`** portion of the message structure.

12.1.1.1 Msg Methods

METHOD	DESCRIPTION
MsgClass	Required on all messages. Sets or gets the **`MsgClass`**, which identifies the specific type of a message (e.g. **`IUpdateMsg`**, **`IRequestMsg`**). **`MsgClass`** allows a range from **0** to **31**, with all values reserved for use by LSEG. For more details about the various message classes, refer to Section 12.1.1.2.
DomainType	Required on all messages. Sets or gets the **`DomainType`**, which identifies the specific domain message model type. **`DomainType`** allows a range from **0** to **255**, where LSEG-defined values are between **0** and **127** and user-defined values are between **128** and **255**. The domain model definition is decoupled from the API and domain models are typically defined in a specification document. Domain models defined by LSEG are specified in the *Transport API LSEG Domain Model Usage Guide*, and Domain types are defined in **`LSEG.Eta.Rdm.DomainType`**.
ContainerType	Required on all messages. Sets or gets the **`ContainerType`**, which identifies the type of message payload content and indicates the presence of a container type (value **129** - **224**), some type of customer-defined, or non-LSEG Rssl Wire Format container type (**225** - **255**), or no message payload (**128**). For more details about container type definitions and use, refer to Section 11.3.
MsgKey	Required on an **`IRequestMsg`** and optional on **`IRefreshMsg`**, **`IStatusMsg`**, **`IUpdateMsg`**, **`IGenericMsg`**, **`IPostMsg`**, and **`IAckMsg`**. Returns the **`MsgKey`**, which houses various attributes that help identify contents flowing within a stream. The **`MsgKey`**, in conjunction with QoS and **`DomainType`**, uniquely identifies the stream. The key typically includes naming and service-related information. For more information about the message key and stream identification, refer to Section 12.1.2andSection 12.1.3.

Table 114: `Msg` Methods

METHOD	DESCRIPTION
StreamId	Required on all messages. Sets or gets **StreamId**, which specifies a unique, signed-integer identifier associated with all messages flowing within a stream. **StreamId** allows a range from -2,147,483,648 to 2,147,483,647, where: • Positive values indicate a consumer-instantiated stream (typically via **IRequestMsg**). • Negative values indicate a provider-instantiated stream (often associated with non-interactive providers). For more information about stream identification and **StreamId** use, refer toSection 12.1.3.
EncodedDataBody	Set or get the **EncodedDataBody**, which is a **Buffer** (with position and length) containing any encoded data contained in the message. If populated, the content type is described by **ContainerType**. **EncodedDataBody** would contain only encoded message payload and length information. **EncodedDataBody** can represent up to 4,294,967,295 bytes of payload. This payload length is typically limited by the contained type's specification. • When encoding, **EncodedDataBody** refers to any pre-encoded message payload. • When decoding, **EncodedDataBody** refers to any encoded message payload.
EncodedMsgBuffer	Returns the **EncodedMsgBuffer**, which is a **Buffer** (with position and length) containing the entire encoding of the message. **EncodedMsgBuffer** would contain both encoded message header and encoded message payload. **EncodedMsgBuffer** is typically populated only while decoding, and refers to the entire encoded message header and payload.
ExtendedHeader	Available for domain-specific user-specified header information. Contents and formatting are defined by the domain model specification. This data is not used in determining stream uniqueness and may not pass through all components. To determine support, refer to the relevant component documentation.
ValidateMsg	Performs a basic validation on the populated **Msg** structure (useful when encoding), ensuring that optional members indicated as present are correctly populated (e.g., that length and data are both populated).
IsFinalMsg	Returns **true** if the message is the last message received on a stream, such as: • The final response to non-streaming requests. • A message with a **StreamState** indicating a closed stream (refer to Section 11.2.7). • A message that explicitly closes the stream (e.g. closed with a **ICloseMsg**). Returns **false** if data is to continue streaming.
Copy	Performs a deep copy of a **Msg** structure. Expects all memory to be owned and managed by the user. • If the memory for the **Buffer**s (i.e. name, attrib, ect.) is not provided, it will be created. • If memory is passed in by the user, the user is responsible for managing the memory.
Clear	Clears this object, so that you can reuse it.

Table 114: Msg Methods (Continued)

12.1.1.2 MsgClasses Values

MSG CLASS VALUE	MESSAGE INTERFACE NAME	DESCRIPTION
REQUEST	IRequestMsg	Consumers use **`IRequestMsg`** to express interest in a new stream or modify some parameters on an existing stream; typically results in the delivery of an **`IRefreshMsg`** or **`IStatusMsg`**. For more information, refer to Section 12.2.1.
REFRESH	IRefreshMsg	The Interactive Provider can use this class to respond to a consumer's request for information (solicited) or provide a data resynchronization point (unsolicited). The non-interactive provider can use this class to initiate a data flow on a new item stream. Conveys state information, QoS, stream permissioning information, and group information in addition to payload. For more information, refer to Section 12.2.2.
UPDATE	IUpdateMsg	Providers (of either type) use the **`IUpdateMsg`** to convey changes to information on a stream. Update messages typically flow on a stream after delivery of a refresh. For more information, refer to Section 12.2.3.
STATUS	IStatusMsg	Indicates changes to the stream or data properties. A provider uses **`IStatusMsg`** to close streams and to indicate successful establishment of a stream when there is no data to convey. For more information, refer to Section 12.2.4. This message can indicate changes: • In **`StreamState`** or **`DataState`** • In a stream's permissioning information • To the item group to which the stream belongs
CLOSE	ICloseMsg	A consumer uses **`ICloseMsg`** to indicate no further interest in a stream. As a result, the stream should be closed. For more information, refer to Section 12.2.5.
GENERIC	IGenericMsg	A bi-directional message that does not have any implicit interaction semantics associated with it, thus the name generic. For more information, refer to Section 12.2.6. After a stream is established via a request-refresh/status interaction: • A consumer can send this message to a provider. • A provider can send this message to a consumer. • A non-interactive provider can send this message to the LSEG Real-Time Advanced Distribution Hub.
POST	IPostMsg	A consumer uses **`IPostMsg`** to push content upstream. This information can be applied to an LSEG Real-Time Distribution System cache or routed further upstream to a data source. After receiving posted data, upstream components can republish it to downstream consumers. For more information, refer to Section 12.2.7.
ACK	IAckMsg	A provider uses **`IAckMsg`** to inform a consumer of success or failure for a specific **`IPostMsg`** or **`ICloseMsg`**. For more information, refer to Section 12.2.8.

Table 115: MsgClasses Values

12.1.2 MsgClasses Methods

METHOD	DESCRIPTION
ToString	Returns a representation of a message interface.
	WARNING! This method creates garbage.

Table 116: MsgClasses Methods

12.1.3 Message Key

The ***Message Key*** (**`MsgKey`**) houses a variety of attributes that help identify content that flows in a particular stream. A data stream is uniquely identified by the **`DomainType`**, Quality of Service data, and message key.

12.1.3.1 MsgKey Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**`Flags`**) to indicate the presence of optional **`MsgKey`** members. For more information about flag values, refer to Section 12.1.2.2. • You can use the following convenient methods to set specific **`MsgKeyFlags`**: **`ApplyHasAttrib`**, **`ApplyHasFilter`**, **`ApplyHasIdentifier`**, **`ApplyHasName`**, **`ApplyHasNameType`**, **`ApplyHasServiceId`**. • You can use the following convenient methods to check whether specific **`MsgKeyFlags`** are set: **`CheckHasAttrib`**, **`CheckHasFilter`**, **`CheckHasIdentifier`**, **`CheckHasName`**, **`CheckHasNameType`**, **`CheckHasServiceId`**.
Id	Sets or gets a service's two-byte, unsigned integer identifier (**`ServiceId`**); a logical mechanism that provides or enables access to a set of capabilities. **`ServiceId`** allows a range from **0** to **65,535**, with **0** being reserved. This value should correspond to the service content being requested or provided. In the Enterprise Transport API, a service corresponds to a subset of content provided by a component, where the Source Directory domain defines specific attributes associated with each service. These attributes include information such as Quality of Service, the specific domain types available, and any dictionaries required to consume information from the service. The Source Directory domain model can obtain this and other types of information. For details, refer to the *Transport API LSEG Domain Model Usage Guide*.
NameType	Sets or gets a numeric value (**`NameType`**), typically enumerated, that indicates the type of the **`Name`** member. Examples are **User Name** or **RIC** (i.e., the Instrument Code). **`NameType`**s are defined on a per-domain model basis. **`NameType`** allows a range from **0** to **255**. Name type values and rules are defined within domain message model specifications. Values associated with LSEG domain models can be found in **`LSEG.Eta.Rdm.InstrumentNameTypes`**.
Name	Sets or gets the **`name`**, which is a **`Buffer`** (with position and length) containing the name associated with the contents of the stream. Specific **`Name`** type and contents should comply with the rules associated with the **`NameType`** member. **`Name`** is a **`Buffer`** type that allows for a name of up to 255 bytes.

Table 117: `MsgKey` Methods

METHOD	DESCRIPTION
Filter	Sets or gets a filter; a combination of up to 32 unique **FilterId** bit-values (where each **FilterId** corresponds to a filter bit-value) that describe content for domain model types with an **FilterList** payload. Filter identifier values are defined by the corresponding domain model specification. • When specified in an **IRequestMsg**, **Filter** conveys information which entries to include in responses. • When specified on a message housing an **FilterList** payload, **Filter** conveys information about which filter entries are present. For more information, refer to Section 11.3.6.
AddFilterId	Converts a **FilterId** value into the bit-value representation and adds bit-value to the **MsgKey.Filter** member. Used with **FilterList** container types. For more information, refer to Section 11.3.6.
CheckFilterId	Converts a **FilterId** value into the bit-value representation and checks for the bit-value presence in the **MsgKey.Filter** member. Used with **FilterList** container types. For more information, refer to Section 11.3.6.
Identifier	User-specified numeric identifier defined on a per-domain model basis. **Identifier** allows a range from -2,147,483,648 to 2,147,483,647. **NOTE:** More information should be present as part of the specific domain model definition.
AttribContainerType	Sets or gets the content type (**AttribContainerType**) of the **MsgKey.EncodedAttrib** information. Can indicate the presence of a container type (value **129** - **224**) or some type of customer-defined container type (**225** - **255**). For more details about container type definitions and use, refer to Section 11.3.
EncodedAttrib	Sets or gets **Name**, which is a **Buffer** (with position and length) containing additional, encoded, message key attribute information. If populated, contents are described by the **AttribContainerType** member. Additional attribute information typically allows for further uniqueness in the identification of a stream. **EncodedAttrib** is an **Buffer** that can represent up to 32,767 bytes of information.
Equals	Compares this **MsgKey** to another **MsgKey**, to determine whether they are the same. Returns **true** if the keys match; **false** otherwise.
Copy	Performs a deep copy of a **MsgKey**. Expects all memory to be owned and managed by user. If the memory for the **Buffer**s (i.e. **Name**, **Attrib**) are not provided, they will be created.
Clear	Clears this object, so that you can reuse it. **TIP:** When decoding, the **MsgKey** object can be reused without using **Clear**.

Table 117: MsgKey Methods (Continued)

12.1.3.2 Message Key Flag Enumeration Values

MSG KEY FLAG	MEANING
MsgKeyFlags.HAS_SERVICE_ID	Indicates the presence of the **ServiceId** member.
MsgKeyFlags.HAS_NAME	Indicates the presence of the **Name** member.
MsgKeyFlags.HAS_NAME_TYPE	Indicates the presence of the **NameType** member.
MsgKeyFlags.HAS_FILTER	Indicates the presence of the **Filter** member.

Table 118: MsgKeyFlags Values

MSG KEY FLAG	MEANING
MsgKeyFlags.HAS_IDENTIFIER	Indicates the presence of the **`Identifier`** member.
MsgKeyFlags.HAS_ATTRIB	Indicates the presence of the **`AttribContainerType`** and **`EncodedAttrib`** members.

Table 118: `MsgKeyFlags` Values (Continued)

12.1.4 Stream Identification

The Enterprise Transport API allows users to simultaneously interact across multiple, independent data streams within a single network connection. Each data stream can be uniquely identified by the specified **DomainType**[1], QoS, and **MsgKey** contents. The **MsgKey** contains a variety of attributes used in defining a stream. To avoid repeatedly sending **MsgKey** and QoS on all messages in a stream[2], a signed integer (referred to as a ***StreamId*** or stream identifier) is used. This **StreamId** can convey all of the same stream identification information, but consumes only a small, fixed-size (four bytes). A positive value **StreamId** indicates a consumer-instantiated stream while a negative value **StreamId** indicates a provider-instantiated stream, usually, but not always, associated with a non-interactive provider application.

For a consumer application, a positive value **StreamId** should be specified on any **IRequestMsg**, along with the **DomainType**, **MsgKey** and additional key attributes, and desired QoS information. An interactive provider application should provide a response, typically a **IRefreshMsg**, which contains the same **StreamId**, **DomainType**, and message key information. If the request specified a QoS range, this response will also contain the concrete or actual QoS being provided for the stream. For more information about QoS, refer to Section 11.2.6.

For a non-interactive provider, the initial **IRefreshMsg** published for each item should contain **DomainType**, message key information, and the QoS being provided for the stream. In addition, the non-interactive provider should specify a negative value **StreamId** to be associated with the stream for the remainder of the run-time.

12.1.4.1 Stream Comparison

To most efficiently use a connection's bandwidth, LSEG recommends that you combine like streams when possible. Two streams are identical when all identifying aspects match - that is the two streams have the same **DomainType**, provided QoS, and all **MsgKey** members. When these message members match, a new stream should not be established, rather the existing stream and **StreamId** should be leveraged to consume or provide this content.

A consumer application can issue a subsequent **IRequestMsg** using the existing **StreamId**, referred to as a ***reissue***. This allows the consumer application to obtain an additional refresh, if desired, and to indicate a change in the priority of the stream. The additional solicited **IRefreshMsg** can satisfy the additional request, and any **IStatusMsg**, **IUpdateMsg**, and **IGenericMsg** content can be provided to both requestors, if different. This behavior is called fan-out and is the responsibility of the consumer application when combining multiple like-streams into a single stream.

A provider application can choose to allow multiple like-streams to be simultaneously established or, more commonly, it can reject any subsequent requests on a different **StreamId** using an **IStatusMsg**. In this case, the **IStatusMsg** would contain a **StreamState** of **StreamStates.CLOSED_RECOVER**, a **DataState** of **DataStates.SUSPECT**, and a state **Code** of **StateCodes.ALREADY_OPEN**. This status message informs the consumer that they already have a stream open for this information and that they should use the existing **StreamId** when re-requesting this content. For more details about the state information, refer to Section 11.2.7.

1. When off-stream posting, it is possible for the post messages sent on the Login stream to contain a different **domainType**. This is a specialized use case and more information is available in .
2. **domainType** is present on all messages and cannot be optimized out like quality of service and **msgKey** information.

12.1.4.2 Private Streams

The Enterprise Transport API provides ***private stream*** functionality, an easy way to ensure delivery of content only between a stream's two endpoints. Private streams behave in a manner similar to standard streams, with the following exceptions:

- All data on a private stream flow between the end provider and the end consumer of the stream.
- Intermediate components do not fan out content (i.e., do not distribute it to other consumers).
- Intermediate components should not cache content.
- In the event of connection or data loss, intermediate components do not recover content. All private stream recovery is the responsibility of the consumer application.

These behaviors ensure that only the two endpoints of the private stream send or receive content associated with the stream. As a result, a private stream can exchange identifying information so the provider can validate the consumer, even through multiple intermediate components (such as might exist in an LSEG Real-Time Distribution System deployment). After a private stream is established, content can flow freely within the stream, following either existing market data semantics (i.e., private Market Price domain) or any other user-defined semantics (i.e., bidirectional exchange of **`IGenericMsg`**s).

For more information about private stream instantiation, refer to Section 13.12.

12.1.4.3 Changeable Stream Attributes

A select number of attributes may change during the life of a stream. A consumer can change attributes via a subsequent **`IRequestMsg`** that uses the same **`StreamId`** as previous requests. A provider of either type can change attributes via a subsequent solicited or unsolicited **`IRefreshMsg`**.

The message key's **`Filter`** member, though not typical, can change between the consumer request and provider response. A change is likely due to a difference between the filter entries for which the consumer asks and the filter entries that the provider can provide. If this behavior is allowed within a domain, it is defined on a per-domain model basis. More information should be present as part of the specific domain model definition.

Contents of the message key's **`EncodedAttrib`** may change. If this behavior is allowed within a domain, it is defined on a per-domain model basis. More information should be present as part of the specific domain model definition.

A consumer can change the **`PriorityClass`** or **`PriorityCount`** via a subsequent **`IRequestMsg`** to indicate more or less interest in a stream. For more information, refer to Section 13.2.

If a Quality of Service range is requested, the provided **`IRefreshMsg`** includes only the concrete Quality of Service, which may be different from the best and worst specified. If a **`Dynamic`** Quality of Service is supported, Quality of Service may occasionally change over the life of the stream, however this should stay within the range requested in **`IRequestMsg`**.

An item's identification might also change, which can result in changes to multiple **`MsgKey`** members. Such a case can occur via a ***redirect***, an **`IRefreshMsg`** or **`IStatusMsg`** with a **`StreamState`** of **`StreamStates.REDIRECTED`** (for more information on the redirected state value, refer to see Section 11.2.7.2). The user can determine the original item identification from the **`MsgKey`** information previously associated with the **`StreamId`** contained in the redirect message. The new item identification that should be requested is provided via the redirect's **`MsgKey`** member. When a redirect occurs, the stream closes. At this point, the user can open a new stream and continue the flow of data by issuing a new **`IRequestMsg`**, containing the redirected **`MsgKey`**.

Some **`IRequestMsg.Flag`** values are allowed to change over the life of a stream. These values include the **`RequestMsgFlags.PAUSE`** and **`RequestMsgFlags.STREAMING`** flags, used when pausing or resuming content flow on a stream. For more details, refer to Section 13.6. Additionally, the **`RequestMsgFlags.NO_REFRESH`** flag can be changed. This allows subsequent reissue requests to be performed where the user does not require a response - this can be useful for a reissue to change the priority of a stream.

12.2 Messages

12.2.1 Request Message Interface

The **IRequestMsg** interface extends the **Msg** interface. A consumer uses an **IRequestMsg** to express interest in a particular information stream. The request's **MsgKey** members help identify the stream and priority information can be used to indicate the stream's importance to the consumer. QoS information can be used to express either a specific desired QoS or a range of acceptable qualities of service that can satisfy the request (refer to Section 13.3).

When an **IRequestMsg** is issued with a new **StreamId**, this is considered a request to open the stream. If requested information is available and the consumer is entitled to receive the information, this typically results in an **IRefreshMsg** being delivered to the consumer, though an **IStatusMsg** is also possible - either message can be used to indicate a stream is open. If information is not available or the user is not entitled, an **IStatusMsg** is typically delivered to provide more detailed information to the consumer.

Issuing an **IRequestMsg** on an existing stream allows a consumer to modify some parameters associated with the stream (also refer to Section 12.1.3.2). Also known as a ***reissue***, this can be used to pause or resume a stream (also refer to Section 13.6), change a Dynamic View (also refer to Section 13.8), increase or decrease the stream's priority (also refer to Section 13.2) or request a new refresh.

12.2.1.1 IRequestMsg Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) to indicate special behaviors and the presence of optional **IRequestMsg** content. For more information about flag values, refer to Section 12.2.1.2. • You can use the following convenient methods to set specific **RequestMsgFlags**: **ApplyConfInfoInUpdates**, **ApplyHasBatch**, **ApplyHasExtendedHdr**, **ApplyHasPriority**, **ApplyHasQos**, **ApplyHasView**, **ApplyHasWorstQos**, **ApplyMsgKeyInUpdates**, **ApplyNoRefresh**, **ApplyPause**, **ApplyPrivateStream**, **ApplyStreaming**. • You can use the following convenient methods to check whether specific **RequestMsgFlags** are set: **CheckConfInfoInUpdates**, **CheckHasBatch**, **CheckHasExtendedHdr**, **CheckHasPriority**, **CheckHasQos**, **CheckHasView**, **CheckHasWorstQos**, **CheckMsgKeyInUpdates**, **CheckNoRefresh**, **CheckPause**, **CheckPrivateStream**, **CheckStreaming**.
Priority	Returns a **Priority** object which you can use to set or get the **PriorityClass** and **PriorityCount**. • **Priority.PriorityClass** can contain values ranging from 0 to 255. • **Priority.PriorityCount** can contain values ranging from 0 to 65,535. For more information about **Priority** and its uses, refer to Section 13.2.

Table 119: IRequestMsg Methods

METHOD	DESCRIPTION
Qos	Returns a **Qos** object which you can use to set or get the allowable QoS for the requested stream. • When specified without a **WorstQos** member, this is the only allowable QoS for the requested stream. If this Qos is unavailable, the stream is not opened. • When specified with a **WorstQos**, this is the best in the range of allowable QoSs. When a QoS range is specified, any QoS within the range is acceptable for servicing the stream. • If neither **Qos** nor **WorstQos** are present on the request, this indicates that any available QoS will satisfy the request. Some components may require **Qos** on initial request and reissue messages. See specific component documentation for details. • For more information, refer to Section 11.2.6. • For specific handling information, refer to Section 13.3.
WorstQos	Returns a **Qos** object which you can use to set or get the least acceptable QoS for the requested stream. When specified with a **Qos** value, this is the worst in the range of allowable QoSs. When a QoS range is specified, any QoS within the range is acceptable for servicing the stream. • For more information, refer to Section 11.2.6. • For specific handling information, refer to Section 13.3.

Table 119: IRequestMsg Methods (Continued)

12.2.1.2 RequestMsgFlags Values

REQUEST MSG FLAG	MEANING
RequestMsgFlags.STREAMING	Indicates whether the request is for streaming data. • If present, the consumer wants to continue to receive changes to information after the initial refresh is complete. • If absent, the consumer wants to receive only the refresh, after which the provider should close the stream. Such a request is typically referred to as a ***non-streaming*** or ***snapshot*** data request. Because a refresh can be split into multiple parts, it is possible for updates to occur between the first and last part of the refresh, even as part of a non-streaming request. For more information about multi-part message handling, refer to Section 13.1.
RequestMsgFlags.NO_REFRESH	Indicates that the consumer application does not require a refresh for this request. This typically occurs after an initial request handshake is completed, usually to change stream attributes (e.g., priority). In some instances, a provider might still deliver a refresh message (but if the consumer does not explicitly ask for it, the message is unsolicited).
RequestMsgFlags.PAUSE	Indicates that the consumer would like to pause the stream, though this does not guarantee that the stream will pause. To resume data flow, the consumer must send a subsequent request message with the **RequestMsgFlags.STREAMING** flag set. For more information, refer to Section 13.6.
RequestMsgFlags.HAS_PRIORITY	Indicates the presence of the **Priority** member, which contains **PriorityClass** and **PriorityCount** members. For more information about using priority, refer to Section 13.2.

Table 120: RequestMsgFlags Values

REQUEST MSG FLAG	MEANING
RequestMsgFlags.HAS_QOS	Indicates the presence of the **Qos** member. • For more information, refer to Section 12.2.1.1 and Section 11.2.6. • For specific handling information, refer to Section 13.3.
RequestMsgFlags.HAS_WORST_QOS	Indicates the presence of the **WorstQos** member. • For more information, refer to Section 12.2.1.1 and Section 11.2.6. • For specific handling information, refer to Section 13.3.
RequestMsgFlags.HAS_VIEW	Indicates that the request message payload might contain a dynamic view, specifying information the application wishes to receive (or that the application wishes to continue receiving a previously specified view). If this flag is not present, any previously specified view is discarded and a full view is provided. For more information about using dynamic views, refer to Section 13.8.
RequestMsgFlags.HAS_BATCH	Indicates that the request message payload contains a list of items of interest, all with matching **MsgKey** information. For more information on using batch requests, refer to Section 13.7.
RequestMsgFlags.HAS_EXTENDED_HEADER	Indicates that the **ExtendedHeader** member is present. Information in the **ExtendedHeader** is defined outside of the scope of the Enterprise Transport API.
RequestMsgFlags.MSG_KEY_IN_UPDATES	Indicates that the consumer wants to receive the full **MsgKey** in update messages. This flag does not guarantee that the **MsgKey** is present in an update message. Instead, the provider application determines whether this information is present (the consumer should be written to handle either the presence or absence of **MsgKey** in any **IUpdateMsg**). When specified on a request to an LSEG Real-Time Advanced Distribution Server, the server fulfills the request.
RequestMsgFlags.CONF_INFO_IN_UPDATES	Indicates that the consumer wants to receive conflation information in update messages delivered on this stream. This flag does not guarantee that conflation information is present in update messages. Instead, the provider application determines whether this information is present (the consumer should be capable of handling conflation information in any **IUpdateMsg**). For details about conflation information on update messages, refer to Section 12.2.3.
RequestMsgFlags.PRIVATE_STREAM	Requests that the stream be opened as private. For details, refer to Section 13.13.

Table 120: RequestMsgFlags Values (Continued)

12.2.2 Refresh Message Interface

The **IRefreshMsg** interface extends the **Msg** interface. **IRefreshMsg** is often provided as an initial response or when an upstream source requires a data resynchronization point. An **IRefreshMsg** contains payload information along with state, Quality of Service, permissioning, and group information.

- If provided as a response to an **IRequestMsg**, the refresh is a ***solicited refresh***. Typically, solicited refresh messages are delivered only to the requesting consumer application
- If some kind of information change occurs (e.g., some kind of error is detected on a stream), an upstream provider can push out an **IRefreshMsg** to downstream consumers. This type of refresh is an ***unsolicited refresh***. Typically, unsolicited refresh messages are delivered to all consumers using each consumer's respective stream.

When an OMM interactive provider sends an **IRefreshMsg**, the **StreamId** should match the **StreamId** on the corresponding **IRequestMsg**. The **MsgKey** should be populated with the appropriate stream identifying information, and often matches the **MsgKey** of the request. When a non-interactive provider sends an **IRefreshMsg**, the provider should assign a negative **StreamId** (when establishing a new stream, the **StreamId** should be unique). In this scenario, the **MsgKey** should define the information that the stream provides.

Using **IRefreshMsg**, an application can fragment the contents of a message payload and deliver the content across multiple messages, with the final message indicating that the refresh is complete. This is useful when providing large sets of content that may require multiple cache look-ups or be too large for an underlying transport layer. Additionally, an application receiving multiple parts of a response can potentially begin processing received portions of data before all content has been received. For more details on multi-part message handling, refer to Section 13.1.

12.2.2.1 IRefreshMsg Methods

METHOD	DESCRIPTION
Flags	Set or get **Flags**, which is a combination of bit values that indicate special behaviors and the presence of optional **IRefreshMsg** content. For more information about flag values, refer to Section 12.2.2.2. • You can use the following convenient methods to set specific **RefreshMsgFlags**: **ApplyClearCache**, **ApplyDoNotCache**, **ApplyHasExtendedHdr**, **ApplyHasMsgKey**, **ApplyHasPartNum**, **ApplyHasPermData**, **ApplyHasPostUserInfo**, **ApplyHasQos**, **ApplyHasSeqNum**, **ApplyPrivateStream**, **ApplyRefreshComplete**, **ApplySolicited**. • You can use the following convenient methods to check whether specific **RefreshMsgFlags** are set: **CheckClearCache**, **CheckDoNotCache**, **CheckHasExtendedHdr**, **CheckHasMsgKey**, **CheckHasPartNum**, **CheckHasPermData**, **CheckHasPostUserInfo**, **CheckHasQos**, **CheckHasSeqNum**, **CheckPrivateStream**, **CheckRefreshComplete**, **CheckSolicited**.
PartNum	Sets or gets the part number (**PartNum**) of this refresh. **PartNum** can contain values ranging from 0 to 32,767 where a value of **0** indicates the initial part of a refresh. • On multi-part refresh messages, **PartNum** should start at **0** (to indicate the initial part) and increment by **1** for each subsequent message in the multi-part message. • If sent on a single-part refresh, a **PartNum** of **0** should be used.
SeqNum	Sets or gets a user-defined sequence number (**SeqNum**), which allows for values ranging from 0 to 4,294,967,295. **SeqNum** should typically increase to help with temporal ordering, but may have gaps depending on the sequencing algorithm in use. Details about sequence number use should be defined within the domain model specification or any documentation for products which require the use of **SeqNum**.

Table 121: IRefreshMsg Methods

METHOD	DESCRIPTION
State	Returns a **State** object which you can use to set or get stream and data state information, which can change over time via subsequent refresh, status messages, or group status notifications. • For details about state information, refer to Section 11.2.7. • For a decision table that provides example behavior for various state combinations, refer to Appendix A.
Qos	Returns a Quality of Service object which you can use to set or get the concrete Quality of Service of the stream. If a range was requested by the **IRequestMsg**, the **Qos** should fall somewhere in this range, otherwise **Qos** should exactly match what was requested. • For more details on Quality of Service, refer to Section 11.2.6. • For specific handling information, refer to Section 13.3.
PermData	**Optional**. Sets or gets **PermData**, which is a **Buffer** (with position and length) that specifies authorization information for this stream. **PermData** has a maximum allowed length of 32,767 bytes. When **PermData** is specified on an **IRefreshMsg**, this indicates authorization information for all content on the stream, unless additional permission information is provided with specific content (e.g., **MapEntry.PermData**). For more information, refer to Section 11.4.
GroupId	Sets or gets **GroupId**, which is a **Buffer** (with position and length) containing information about the item group to which this stream belongs. The **GroupId Buffer** has a maximum allowed length of 255 bytes. You can change the associated **GroupId** via a subsequent **IStatusMsg** or **IRefreshMsg**. Group status notifications can change the state of an entire group of items. For more information about item groups, refer to Section 13.4.
PostUserInfo	**Optional**. Returns a **PostUserInfo** object which can be used to set or get information that identifies the user posting this information. If present on an **IRefreshMsg**, this implies that the refresh was posted to the system by the user described in **PostUserInfo**. • For more information about posting, refer to Section 13.9. • For more information about the ***Visible Publisher Identifier***, refer to Section 13.11.

Table 121: IRefreshMsg Methods (Continued)

12.2.2.2 RefreshMsgFlags Values

REFRESH MSG FLAG	MEANING
RequestMsgFlags.REFRESH_COMPLETE	Indicates that the message is the final part of the **IRefreshMsg**. This flag value should be set when: • The message is a single-part refresh (i.e., atomic refresh). • The message is the final part of a multi-part refresh. For more information about multi-part message handling, refer to Section 13.1.
RequestMsgFlags.SOLICITED	Indicates that the refresh is sent as a response to a request, referred to as a ***solicited*** refresh. A refresh sent to inform a consumer of an upstream change in information (i.e., an ***unsolicited*** refresh) must not include this flag.
RequestMsgFlags.DO_NOT_CACHE	Indicates that the message's payload information should not be cached. This flag value applies only to the message on which it is present.
RequestMsgFlags.CLEAR_CACHE	Indicates that the stream's stored payload information should be cleared. This is typically set by providers when: • Sending the initial solicited **IRefreshMsg**. • Sending the first part of a multi-part **IRefreshMsg**. • Some portion of data is known to be invalid.
RequestMsgFlags.HAS_MSG_KEY	Indicates that the **IRefreshMsg** contains a populated **MsgKey**. This can aid in associating a request with its corresponding refresh or identify an item sent from an non-interactive provider application.
RequestMsgFlags.HAS_QOS	Indicates the presence of the **Qos** member. For specific handling information, refer to Section 13.3.
RequestMsgFlags.HAS_SEQ_NUM	Indicates the presence of the **SeqNum** member.
RequestMsgFlags.HAS_PART_NUM	Indicates the presence of the **PartNum** member.
RequestMsgFlags.HAS_PERM_DATA	Indicates the presence of the **PermData** member.
RequestMsgFlags.HAS_POST_USER_INFO	Indicates that this message includes **PostUserInfo**, implying that this **IRefreshMsg** was posted by the user described in **PostUserInfo**.
RequestMsgFlags.HAS_EXTENDED_HEADER	Indicates the presence of the **ExtendedHeader** member.
RequestMsgFlags.PRIVATE_STREAM	Acknowledges the initial establishment of a private stream or, when combined with a **StreamState** value of **StreamStates.REDIRECTED**, indicates that a stream can only be opened as private. For details, refer to Section 13.12.

Table 122: ResreshMsgFlags Values

12.2.3 Update Message Interface

The **IUpdateMsg** interface extends the **IMsg** interface. Providers (both interactive and non-interactive) use **IUpdateMsg** to convey changes to data associated with an item stream. When streaming, update messages typically flow after the delivery of an initial refresh. Update messages can be delivered between parts of a multi-part refresh message, even in response to a non-streaming request. For more information on multi-part message handling, refer to Section 13.1.

Some providers can aggregate the information from multiple update messages into a single update message using a technique called conflation. Conflation typically occurs if a conflated QoS is requested (refer to Section 11.2.6), a stream is paused (refer to Section 13.6), or if a consuming application is unable to keep up with a stream's data rates. If conflation is used, specific information can be provided with **IUpdateMsg** via optional conflation information.

12.2.3.1 UpdateMsg Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicate special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.3.2. • You can use the following convenient methods to set specific **UpdateMsgFlags**: **ApplyDiscardable**, **ApplyDoNotCache**, **ApplyDoNotConflate**, **ApplyDoNotRipple**, **ApplyHasConfInfo**, **ApplyHasExtendedHdr**, **ApplyHasMsgKey**, **ApplyHasPermData**, **ApplyHasPostUserInfo**, **ApplyHasSeqNum**. • You can use the following convenient methods to check whether specific **UpdateMsgFlags** are set: **CheckDiscardable**, **CheckDoNotCache**, **CheckDoNotConflate**, **CheckDoNotRipple**, **CheckHasConfInfo**, **CheckHasExtendedHdr**, **CheckHasMsgKey**, **CheckHasPermData**, **CheckHasPostUserInfo**, **CheckHasSeqNum**.
UpdateType	Sets or gets the type of data (**UpdateType**) in the **IUpdateMsg**, where values are typically defined in an enumeration (valid values range from **0** to **255**). Examples of possible update types include: **Trade**, **Quote**, or **Closing Run**. • Domain message model specifications define available update types. • For domain models provided by LSEG, **LSEG.Eta.Rdm.UpdateEventTypes** defines available update types.
SeqNum	Sets or gets a user-defined sequence number (**SeqNum**), which can range in value from 0 to 4,294,967,295. To help with temporal ordering, **SeqNum** should increase across messages, but can have gaps depending on the sequencing algorithm in use. Details about sequence number use should be defined within the domain model specification or any documentation for products which require the use of **SeqNum**.
ConflationCount	Sets or gets the **ConflationCount**. When conflating data, this value indicates the number of updates conflated or aggregated into this **IUpdateMsg**. **ConflationCount** allows for values ranging from 1 to 32,767.
ConflationTime	Sets or gets the **ConflationTime**. When conflating data, this value indicates the period of time over which individual updates were conflated or aggregated into this **IUpdateMsg** (typically in milliseconds; for further details, refer to specific component documentation). **ConflationTime** allows for values ranging from 1 to 65,535.

Table 123: UpdateMsg Methods

METHOD	DESCRIPTION
PermData	**Optional**. Sets or gets **PermData**, which is a **Buffer** (with position and length) that specifies authorization information for this stream. When specified, **PermData** indicates authorization information for only the content within this message, though this can be overridden for specific content within the message (e.g., **MapEntry.PermData**). **PermData** has a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.4.
PostUserInfo	**Optional**. Returns a **PostUserInfo** object that you can use to set or get information that identifies • For more information about posting, refer to Section 13.9. • For more information about the Visible Publisher Identifier, refer to Section 13.11.

Table 123: UpdateMsg Methods (Continued)

12.2.3.2 Update Message Flag Enumeration Values

UPDATE MSG FLAG	MEANING
UpdateMsgFlags.DISCARDABLE	Indicates that this update can be discarded. Common for options with no open interest.
UpdateMsgFlags.DO_NOT_CACHE	Indicates that payload information associated with this message should not be cached. **UpdateMsgFlags.DO_NOT_CACHE** applies only to the message on which it is present.
UpdateMsgFlags.DO_NOT_CONFLATE	Indicates that this message should not be conflated. This flag value only applies to the message on which it is present.
UpdateMsgFlags.DO_NOT_RIPPLE	Indicates that the contents of this message should not be rippled. Rippling is typically associated with an **FieldList**. For additional information, refer to Section 11.3.1.4.
UpdateMsgFlags.HAS_MSG_KEY	Indicates that the **IUpdateMsg** contains a populated **MsgKey**. The additional key information can help associate a request with updates or identify an item being sent from a non-interactive provider application. This information is typically not necessary in an **IUpdateMsg** as the **StreamId** can be used to determine the same information with less bandwidth cost.
UpdateMsgFlags.HAS_SEQ_NUM	Indicates the presence of the **SeqNum** member.
UpdateMsgFlags.HAS_CONF_INFO	Indicates the presence of **ConflationTime** and **ConflationCount** information.
UpdateMsgFlags.HAS_PERM_DATA	Indicates the presence of the **PermData** member.
UpdateMsgFlags.HAS_POST_USER_INFO	Indicates that this message includes **PostUserInfo**, implying that this **IUpdateMsg** was posted by the user described in the **PostUserInfo**.
UpdateMsgFlags.HAS_EXTENDED_HEADER	Indicates the presence of the **ExtendedHeader** member.

Table 124: UpdateMsgFlags Values

12.2.4 Status Message Interface

The **IStatusMsg** interface extends the **Msg** interface. An **IStatusMsg** can convey changes in **StreamState** or **DataState** (refer to Section 11.2.7), changes in a stream's permissioning information (refer to Section 9.4), or changes to the item group of which the stream is a part (refer to Section 13.4). A Provider application uses **IStatusMsg** to close streams to a consumer, in conjunction with an initial request or later after the stream has been established. An **IStatusMsg** can also indicate the successful establishment of a stream, though the message might not contain data (useful in establishing a stream solely to exchange bi-directional **IGenericMsgs**).

12.2.4.1 StatusMsg Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) indicating special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.4.2. • You can use the following convenient methods to set specific **StatusMsgFlags**: **ApplyClearCache**, **ApplyHasExtendedHdr**, **ApplyHasGroupId**, **ApplyHasMsgKey**, **ApplyHasPermData**, **ApplyHasPostUserInfo**, **ApplyHasState**, **ApplyyPrivateStream**. • You can use the following convenient methods to check whether specific **StatusMsgFlags** are set: **CheckClearCache**, **CheckHasExtendedHdr**, **CheckHasGroupId**, **CheckHasMsgKey**, **CheckHasPermData**, **CheckHasPostUserInfo**, **CheckHasState**, **CheckHasPrivateStream**.
State	Returns a State object that you can use to set or get stream and data state information, which can change over time via subsequent refresh or status messages or group status notifications. • For details about state information, refer to Section 11.2.7. • For a decision table that provides example behavior for various state combinations, refer to Appendix A.
PermData	**Optional**. Sets or gets **PermData**, which is a **Buffer** (with position and length) that specifies authorization information for this stream, unless additional permission information is provided with specific content (e.g., **MapEntry.PermData**). **PermData** allows a maximum length of 32,767 bytes. For more information, refer to Section 11.4.
GroupId	Sets or gets the **GroupId**, which is a **Buffer** (with position and length)with a maximum allowed length of 255 bytes that contains information about the item group to which this stream belongs. A subsequent **IStatusMsg** or **IRefreshMsg** can change the item group's associated **GroupId**, while group status notifications can change the state of an entire group of items. For more information about item groups, refer to Section 13.4.
PostUserInfo	**Optional**. Returns a **PostUserInfo** object that you can use to set or get information that identifies the user who posted this information. • For more information about posting, refer to Section 13.9. • For more information about Visible Publisher Identifier, refer to Section 13.11.

Table 125: StatusMsg Methods

12.2.4.2 StatusMsgFlags Values

FLAG	MEANING
StatusMsgFlags.CLEAR_CACHE	Indicates that the application should clear stored header or payload information associated with the stream. This can happen if some portion of data is invalid.
StatusMsgFlags.HAS_MSG_KEY	Indicates that the **IStatusMsg** contains a populated **MsgKey**. The **MsgKey** can be used to aid in associating a request to a status message or identify an item sent from an non-interactive provider application.
StatusMsgFlags.HAS_STATE	Indicates the presence of **State** information. If **State** information is not present, the message might be changing the stream's permission information or **GroupId**.
StatusMsgFlags.HAS_PERM_DATA	Indicates the presence of **PermData**. When present, the message might be changing the stream's permission information.
StatusMsgFlags.HAS_GROUP_ID	Indicates the presence of **GroupId**. When present, the message might be changing the stream's **GroupId**.
StatusMsgFlags.HAS_POST_USER_INFO	Indicates the presence of **PostUserInfo**, which identifies the user who posted the **IStatusMsg**.
StatusMsgFlags.HAS_EXTENDED_HEADER	Indicates the presence of **ExtendedHeader**.
StatusMsgFlags.PRIVATE_STREAM	Acknowledges the establishment of a private stream, or when combined with a **StreamState** value of **StreamStates.REDIRECTED**, indicates that a stream can be opened only as private. For details, refer to Section 13.12.

Table 126: StatusMsgFlags Values

12.2.5 Close Message Interface

The **ICloseMsg** interface extends the **IMsg** interface. A consumer uses **ICloseMsg** to indicate no further interest in an item stream and to close the stream. The **StreamId** indicates the item stream to which **ICloseMsg** applies.

12.2.5.1 CloseMsg Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicate special behaviors and the presence of optional content. For available flag values, refer to **ICloseMsgFlags** in Section 12.2.5.2. • You can use the following convenient methods to set specific **StatusMsgFlags**: **ApplyAck**, **ApplyHasExtendedHdr**. • You can use the following convenient methods to check whether specific **StatusMsgFlags** are set: **CheckAck**, **CheckHasExtendedHdr**.

Table 127: CloseMsg Methods

12.2.5.2 CloseMsgFlags Values

CLOSE MSG FLAG	MEANING
CloseMsgFlags.ACK	If present, the consumer wants the provider to send an **IAckMsg** to indicate that the **ICloseMsg** has been processed properly and the stream is properly closed. This functionality might not be available with some components; for details, refer to the component's documentation.
CloseMsgFlags.HAS_EXTENDED_HEADER	Indicates the presence of **ExtendedHeader**.

Table 128: CloseMsgFlags Values

12.2.6 Generic Message Interface

The **IGenericMsg** interface extends the **IMsg** interface. **IGenericMsg** is a bi-directional message without any implicit interaction semantics associated with it, hence the name generic. After a stream is established via a request-refresh/status interaction, both consumers and providers can send **IGenericMsgs** to one another, and non-interactive provider applications can leverage them. Generic messages are transient and typically not cached by LSEG Real-Time Distribution System components.

The **MsgKey** of an **IGenericMsg** does not need to match the **MsgKey** information of the stream over which the generic message flows. Thus, key information can be used independently within the stream. A domain message model specification typically defines any specific message usage, **MsgKey** usage, expected interactions, and handling instructions.

12.2.6.1 GenericMsg Methods

METHOD	DESCRIPTION
Flags	Sets or getsa combination of bit values (**Flags**) that indicate special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.6.2. • You can use the following convenient methods to set specific **GenericMsgFlags**: **ApplyHasExtendedHdr**, **ApplyHasMsgKey**, **ApplyHasPartNum**, **ApplyHasPermData**, **ApplyHasSecondarySeqNum**, **ApplyHasSeqNum**, **ApplyMessageComplete**. • You can use the following convenient methods to check whether specific **GenericMsgFlags** are set: **CheckHasExtendedHdr**, **CheckHasMsgKey**, **CheckHasPartNum**, **CheckHasPermData**, **CheckHasSecondarySeqNum**, **CheckHasSeqNum**, **CheckMessageComplete**.
PartNum	Sets or gets the part number (**PartNum**) of this generic message, typically used with multi-part generic messages. **PartNum** can contain values ranging from 0 to 32,767, where a value of **0** indicates the initial part of a refresh. • If sent on a single-part post message, use a **PartNum** of **0**. • On multi-part post messages, use a **PartNum** of **0** on the initial part and increment **PartNum** in each subsequent part by **1**.
SeqNum	Sets or gets a user-defined sequence number (**seqNum**) ranging in value from 0 to 4,294,967,295. A **SeqNum** typically corresponds to the sequencing of this message. To help with temporal ordering, **SeqNum** should increase across messages, but can have gaps depending on the sequencing algorithm in use. Details about using **SeqNum** should be defined in the domain model specification or the documentation for products that must use **SeqNum**.
SecondarySeqNum	Sets or gets an additional user-defined sequence number (**SecondarySeqNum**) ranging in value from 0 to 4,294,967,295. When using **IGenericMsg** on a stream in a bi-directional manner, **SecondarySeqNum** is often used as an acknowledgment sequence number. For example, a consumer sends a generic message with **SeqNum** populated to indicate the sequence of this message in the stream and **SecondarySeqNum** set to the **SeqNum** last received from the provider. This effectively acknowledges all messages received up to that point while still sending additional information. Sequence number use should be defined within the domain model specification or any documentation for products that use **SecondarySeqNum**.
PermData	**Optional**. Sets or gets **PermData**, which is a **Buffer** (with position and length) that indicates authorization information for content within this message only, though this can be overridden for specific content within the message (e.g. **MapEntry.PermData**). **PermData** allows a maximum length of 32,767 bytes. For more information, refer to Section 11.4.

Table 129: IGenericMsg Methods

12.2.6.2 GenericMsgFlags Values

GENERIC MSG FLAG	MEANING
GenericMsgFlags.MESSAGE_COMPLETE	When set, this flag indicates that the message is the final part of an **IGenericMsg**. This flag should be set on: • Single-part generic messages (i.e., an atomic generic message). • The last message (final part) in a multi-part generic message. For more information on handling multi-part messages, refer to Section 13.1.
GenericMsgFlags.HAS_MSG_KEY	Indicates the presence of a populated **MsgKey**. Use of a **MsgKey** differentiates a generic message from the **MsgKey** information specified for other messages within the stream. Contents and semantics associated with an **IGenericMsg.MsgKey** should be defined by the domain model specification that employs them.
GenericMsgFlags.HAS_SEQ_NUM	Indicates the presence of the **SeqNum** member.
GenericMsgFlags.HAS_SECONDARY_SEQ_NUM	Indicates the presence of the **SecondarySeqNum** member.
GenericMsgFlags.HAS_PART_NUM	Indicates the presence of the **PartNum** member.
GenericMsgFlags.HAS_PERM_DATA	Indicates the presence of the **PermData** member.
GenericMsgFlags.HAS_EXTENDED_HEADER	Indicates presence of the **ExtendedHeader** member.

Table 130: IGenericMsgFlags Values

12.2.7 Post Message Interface

The **IPostMsg** interface extends the **IMsg** interface. A consumer application uses **IPostMsg** to push content to upstream components. Such content can be applied to an LSEG Real-Time Distribution System cache or routed further upstream to the source of data. After upstream components receive the content, the components can republish the data to their downstream consumers.

Post messages can be routed along a specific item stream, referred to as ***on-stream*** posting, or along a user's Login stream, referred to as ***off-stream*** posting. **IPostMsg** can contain any container type, including other messages. User identification information can be associated with a post message and be provided along with posted content. For more details, refer to Section 13.9.

12.2.7.1 IPostMsg Methods

METHOD	DESCRIPTION
Flags	Sets or gets a combination of bit values (**Flags**) that indicate special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.7.2. • You can use the following convenient methods to set specific **PostMsgFlags**: **applyAck**, **ApplyHasExtendedHdr**, **ApplyHasMsgKey**, **ApplyHasPartNum**, **ApplyyHasPermData**, **ApplyHasPostId**, **ApplyHasPostUserRights**, **ApplyHasSeqNum**, **ApplyPostComplete**. • You can use the following convenient methods to check whether specific **PostMsgFlags** are set: **CheckAck**, **CheckHasExtendedHdr**, **CheckHasMsgKey**, **CheckHasPartNum**, **CheckHasPermData**, **CheckHasPostId**, **CheckHasPostUserRights**, **CheckHasSeqNum**, **CheckPostComplete**.
PartNum	Sets or gets the part number for this post message, typically used with multi-part post messages. **PartNum** can contain values ranging from 0 to 32,767, where a value of 0 indicates the initial part of a refresh. • If sent on a single-part post message, use a **PartNum** of **0**. • On multi-part post messages, use a **PartNum** of **0** on the initial part and in each subsequent part, increment **PartNum** part by **1**.
PostId	Sets or gets the consumer-assigned identifier (**PostId**), which can range in value from 0 to 4,294,967,295. **PostId** distinguishes different post messages. In multi-part post messages, each part must use the same **PostId** value.
SeqNum	Sets or gets a user-defined sequence number (**SeqNum**), typically corresponding to the sequencing of the message. **SeqNum** allows for values ranging from 0 to 4,294,967,295. To help with temporal ordering, **SeqNum** should increase, though gaps might exist depending on the sequencing algorithm in use. Details about **SeqNum** use should be defined in the domain model specification or any documentation for products that use **SeqNum**. When acknowledgments are requested, the **SeqNum** will be provided back in the **IAckMsg** to help identify the **IPostMsg** being acknowledged.
PermData	**Optional**. Sets or gets **PermData**, which is a **Buffer** (with position and length) that specifies authorization information for content in this message only. **PermData** can be overridden for specific content within the message (e.g. **MapEntry.PermData**). **PermData** allows a maximum length of 32,767 bytes. For more information, refer to Section 11.4.

Table 131: IPostMsg Methods

METHOD	DESCRIPTION
PostUserInfo	Returns a **PostUserInfo** object which can set or get information that identifies the posting user. **PostUserInfo** can optionally be provided along with posted content via a **IRefreshMsg**, **IUpdateMsg**, and **IStatusMsg**. • For more information about posting, refer to Section 13.9. • For more information about Visible Publisher Identifier, refer to Section 13.11.
PostUserRights	Conveys the rights or abilities of the user posting this content, which can indicate whether the user is permissioned to: • Create items in the cache of record, • Delete items from the cache of record, or • Modify the **PermData** on items already present in the cache of record. For details about different rights, refer to Section 12.2.7.3.

Table 131: IPostMsg Methods (Continued)

12.2.7.2 PostMsgFlags Values

POST MSG FLAG	MEANING
PostMsgFlags.POST_COMPLETE	Indicates that this is the final part of the **IPostMsg**. This flag should be set on: • Single-part post messages (i.e., an atomic post message). • The final part of a multi-part post message. For more information about multi-part message handling, refer to Section 13.1.
PostMsgFlags.ACK	Specifies that the consumer wants the provider to send an **IAckMsg** to indicate that the **IPostMsg** was processed properly. When acknowledging an **IPostMsg**, the provider must include the **PostId** in the **AckId** and communicate any associated **SeqNum**.
PostMsgFlags.HAS_MSG_KEY	Indicates that the **IPostMsg** contains a populated **MsgKey** that identifies the stream on which the information is posted. An **MsgKey** is typically required for off-stream posting and is not necessary when on-stream posting. For more detailed information about posting, refer to Section 13.9.
PostMsgFlags.HAS_SEQ_NUM	Indicates the presence of the **SeqNum** member.
PostMsgFlags.HAS_POST_ID	Indicates the presence of the **PostId** member.
PostMsgFlags.HAS_POST_USER_RIGHTS	Indicates the presence of the **PostUserRights** member.
PostMsgFlags.HAS_PART_NUM	Indicates the presence of the **PartNum** member.
PostMsgFlags.HAS_PERM_DATA	Indicates the presence of the **PermData** member.
PostMsgFlags.HAS_EXTENDED_HEADER	Indicates the presence of the **ExtendedHeader** member.

Table 132: PostMsgFlags Values

12.2.7.3 PostUserRights Values

POST USER RIGHT	MEANING
PostUserRights.NONE	The user has no additional posting abilities.
PostUserRights.CREATE	The user is allowed to create items in the cache of record.
PostUserRights.DELETE	The user is allowed to remove items from the cache of record.
PostUserRights.MODIFY_PERM	The user is allowed to modify the **`PermData`** associated with items already in the cache of record.

Table 133: `PostUserRights` Values

12.2.8 PostUserInfo Methods

METHOD	DESCRIPTION
UserId	Sets or gets the **`UserId`**, which identifies the specific user that posted this data.
UserAddr	Sets or gets the IP Address (**`UserAddr`**) of the user that posted this data. Though the address can be specified as either a **`Long`** or **`String`** (e.g., "127.0.0.1"), if it is specified as a **`String`**, it will be converted to its **`Int`** equivalent.
UserAddrToString	Converts an IP address in integer format to its string equivalent.
Clear	Clears the object, so that it can be reused.

Table 134: PostUserInfo Methods

12.2.9 Acknowledgment Message Interface

A provider can send an **IAckMsg** to a consumer to indicate receipt of a specific message. The acknowledgment carries success or failure (i.e., a negative acknowledgment or 'NAK') information to the consumer. Currently, a consumer can request acknowledgment for a **IPostMsg** or **ICloseMsg**.

12.2.9.1 IAckMsg Methods

METHOD	DESCRIPTION
Flags	Sets or gets flags, which is a combination of bit values indicating special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.8.2. • You can use the following convenient methods to set specific **AckMsgFlags**: **ApplyHasExtendedHdr**, **ApplyHasMsgKey**, **ApplyHasNakCode**, **ApplyHasSeqNum**, **ApplyHasText**, **ApplyPrivateStream**. • You can use the following convenient methods to check whether specific **AckMsgFlags** are set: **CheckHasExtendedHdr**, **CheckHasMsgKey**, **CheckHasNakCode**, **CheckHasSeqNum**, **CheckHasText**, **CheckPrivateStream**.
AckId	Sets or gets **AckId**, which associatesthe **IAckMsg** with the message it acknowledges. **AckId** allows for values ranging from 0 to 4,294,967,295. When acknowledging a **IPostMsg**, **AckId** typically matches the post message's **PostId**.
SeqNum	Sets or gets **SeqNum**, which specifies a user-defined sequence number, ranging in value from 0 to 4,294,967,295. To help with temporal ordering, **SeqNum** should increase, though gaps might exist depending on the sequencing algorithm in use. The acknowledgment message may populate this with the **SeqNum** of the **IPostMsg** being acknowledged. This helps correlate the message being acknowledged when the **PostId** alone is not sufficient (e.g., multi-part post messages).
NakCode	Sets or gets **NakCode**. If present, this message indicates a NAK. The **NakCode** is an enumerated code value (ranging in value from 1 to 255) that provides additional information about the reason for the NAK. **NakCode** values are defined in Section 12.2.8.3
Text	**Optional**. Sets or gets **Text**, which is a **Buffer** (with position and length) that provides additional information about the acceptance or rejection of the message being acknowledged. **Text** has a maximum allowed length of 65,535 bytes.

Table 135: IAckMsg Methods

12.2.9.2 AckMsgFlags Values

ACK MSG FLAG	MEANING
AckMsgFlags.HAS_MSG_KEY	Indicates the presence of a populated **MsgKey**. When present, this is typically populated to match the information being acknowledged.
AckMsgFlags.HAS_SEQ_NUM	Indicates the presence of the **Sequm** member.
AckMsgFlags.HAS_NAK_CODE	Indicates the presence of the **NakCode** member.
AckMsgFlags.HAS_TEXT	Indicates the presence of the **Text** member.

Table 136: AckMsgFlags Values

ACK MSG FLAG	MEANING
AckMsgFlags.HAS_EXTENDED_HEADER	Indicates presence of the **`ExtendedHeader`** member.
AckMsgFlags.PRIVATE_STREAM	Acknowledges the initial establishment of a private stream. For details, refer to Section 13.13.

Table 136: AckMsgFlags Values(Continued)

12.2.9.3 NakCodes Values

NAK CODE VALUE	DESCRIPTION
NakCodes.ACCESS_DENIED	The user is not permissioned to post on the item or service.
NakCodes.DENIED_BY_SRC	The source being posted to has denied accepting this post message.
NakCodes.SOURCE_DOWN	The source being posted to is down or unavailable.
NakCodes.SOURCE_UNKNOWN	The source being posted to is unknown and unreachable.
NakCodes.NO_RESOURCES	Some component along the path of the post message does not have appropriate resources available to continue processing the post.
NakCodes.NO_RESPONSE	There is no response from the source being posted to. This may mean that the source is unavailable or that there is a delay in processing the posted information.
NakCodes.GATEWAY_DOWN	A gateway device for handling posted or contributed information is down or unavailable.
NakCodes.SYMBOL_UNKNOWN	The system does not recognize the item information provided with the post message. This may be an invalid item.
NakCodes.NOT_OPEN	The item being posted to does not have an available stream.
NakCodes.INVALID_CONTENT	The content of the post message is invalid (it does not match the expected formatting) and cannot be posted.

Table 137: NakCodes Values

12.2.10 Msg Encoding and Decoding

12.2.10.1 Msg Encoding Interfaces

When encoding, any message interfaces can call **Msg** encoding methods without the need to explicitly cast to the **Msg** interface. For simplicity, this encoding section will refer to the **Msg** interface.

An **Msg** can be encoded from pre-encoded data or by encoding individual pieces of data as they are provided.

ENCODE INTERFACE	DESCRIPTION
Encode	Encodes a message where all message content is pre-encoded. • **MsgKey** attribute information should be encoded and populated on **MsgKey.EncodedAttrib** prior to this call. • **ExtendedHeader** information should be encoded and populated on the message's **ExtendedHeader** member prior to this call. • Message payload information should be encoded and populated on the **EncodedDataBody** member prior to this call.
EncodeInit	Begins encoding of an **Msg**. All message header elements should be properly populated. The **ContainerType** member should be populated with the specific type of message payload. • If encoding **MsgKey** attribute information: pre-encoded **MsgKey** attribute information should be populated in **MsgKey.EncodedAttrib**. Unencoded **MsgKey** attribute information should be encoded after **Msg.EncodeInit** returns, followed by **EncodeKeyAttribComplete**. • If encoding **ExtendedHeader** information: pre-encoded **ExtendedHeader** information should be populated in the **ExtendedHeader** member of the message. Unencoded **ExtendedHeader** information should be encoded after the call to **Msg.EncodeInit** and after **MsgKey** attribute information is encoded. When **ExtendedHeader** encoding is completed, call **EncodeExtendedHeaderComplete**.
EncodeComplete	Completes encoding of an **Msg**. All message content should be encoded prior to this call. This function expects the same **EncodeIterator** that was used with **Msg.EncodeInit**. • If the content (i.e., payload, **MsgKey** attrib, and **ExtendedHeader**) encodes successfully, the **Bool success** parameter should be set to **true** to finish encoding. • If any of the content fails to encode, the **bool success** parameter should be set to **false** to roll back the encoding of the message.

Table 138: Msg Encode Methods

ENCODE INTERFACE	DESCRIPTION
EncodeKeyAttribComplete	Completes encoding of any non-pre-encoded **MsgKey** attribute information. Can be used only when message encoding leverages **Msg.EncodeInit**. If the **MsgKeyFlags.HAS_ATTRIB** flag is set and **MsgKey.EncodedAttrib** is not populated, **MsgKey** attribute information is expected after **Msg.EncodeInit** returns, with the specific **attribContainerType** methods being used to encode it. This method expects the same **EncodeIterator** used with **Msg.EncodeInit**. • If encoding of the **MsgKey** attribute information succeeds, the **Bool success** parameter should be set to **true** to finish attribute encoding. • If encoding of attributes fails, the **Bool success** parameter should be set to **false** to roll back encoding prior to **MsgKey** attributes. If both **MsgKey** attributes and **ExtendedHeader** information are being encoded, **MsgKey** attributes are expected first with **ExtendedHeader** being encoded after the call to **EncodeKeyAttribComplete**.
EncodeExtendedHeaderComplete	Completes encoding of any non-pre-encoded **ExtendedHeader** information. Can be used only when the message encoding leverages **Msg.EncodeInit**. If the specific message's **HAS_EXTENDED_HEADER** flag is set and **ExtendedHeader** is not populated, this information is expected after **Msg.EncodeInit** (and **EncodeKeyAttribComplete** if encoding **MsgKey** attributes) returns. This function expects the same **EncodeIterator** used with previous message encoding functions. • If encoding of **ExtendedHeader** succeeds, the **Bool success** parameter should be set to **true** to finish encoding. • If encoding of **ExtendedHeader** fails, the **Bool success** parameter should be set to **false** to roll back to encoding prior to **ExtendedHeader**. If both **MsgKey** attributes and **ExtendedHeader** information are being encoded, **MsgKey** attributes are expected first, while **ExtendedHeader** should be encoded after the call to **EncodeKeyAttribComplete**.

Table 138: Msg Encode Methods (Continued)

12.2.10.2 Msg Encoding Example 1

The following code sample demonstrates **Msg** encoding, showing the use of **EncodeInit** with **EncodeComplete** and includes unencoded **MsgKey** attribute information, unencoded payload, and unencoded **ExtendedHeader** information. While this example demonstrates error handling for the initial encode function, it omits additional error handling to simplify the example (though it should still be performed).

```
/* EXAMPLE 1 - Msg.EncodeInit/Complete with unencoded msgKey attribute, payload, and extendedHeader */

/* Populate and encode a requestMsg */
IRequestMsg reqMsg = new Msg();
reqMsg.MsgClass = MsgClasses.REQUEST; /* message is a request */
reqMsg.DomainType = (int)DomainType.MARKET_PRICE;
reqMsg.ContainerType = DataTypes.ELEMENT_LIST;
/* Choose a stream Id that is not in use if this is a new request, otherwise reuse associated id */
reqMsg.StreamId = 6;
/* Populate flags for request message members and behavior - our message is for a streaming request,
will specify a quality of service range, priority, contains an extended header and payload is a
dynamic view request */
reqMsg.ApplyStreaming();
reqMsg.ApplyHasPriority();
reqMsg.ApplyHasQos();
reqMsg.ApplyHasWorstQos();
reqMsg.ApplyHasExtendedHdr();
reqMsg.ApplyHasView();
```

```
/* Populate qos range and priority */
reqMsg.Priority.PriorityClass = 2;
reqMsg.Priority.Count = 1;
/* Populate best qos allowed */
reqMsg.Qos.Rate(QosRates.TICK_BY_TICK);
reqMsg.Qos.Timeliness(QosTimeliness.REALTIME);
/* Populate worst qos allowed, Rate and Timeliness values allow for RateInfo and TimeInfo to be sent */
reqMsg.WorstQos.Rate(QosRates.TIME_CONFLATED);
reqMsg.WorstQos.RateInfo(1500);
reqMsg.WorstQos.Timeliness(QosTimeliness.DELAYED);
reqMsg.WorstQos.TimeInfo(20);

/* Populate msgKey to specify a ServiceId, a Name with type of RIC (which is default NameType) and Attrib
*/
reqMsg.MsgKey.ApplyHasServiceId();
reqMsg.MsgKey.ApplyHasName();
reqMsg.MsgKey.ApplyHasAttrib();
reqMsg.MsgKey.ServiceId = 1;
/* Specify a Name. Because this is a RIC, no NameType is required. */
reqMsg.MsgKey.Name.Data("TRI");
/* Msg Key attribute info will be encoded after Msg.EncodeInit returns */
reqMsg.MsgKey.AttribContainerType = DataTypes.ELEMENT_LIST;

/* begin encoding of message - assumes that encIter is already populated with
buffer and version information, store return value to determine success or failure */
/* data max encoded size is unknown so 0 is used */
if ((retCode = reqMsg.EncodeInit(encIter, 0)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error encountered with Msg.EncodeInit. Error code: {0}",
      retCode.GetAsString());
}
else
{
   Buffer nonRWFBuffer = new Buffer();
   /* retCode should be CodecReturnCode.ENCODE_MSG_KEY_OPAQUE */
   /* encode msgKey attrib as element list to match setting of AttribContainerType */
   {
      elementList.ApplyHasStandardData();
      /* now encode nested container using its own specific encode methods */
      if ((retCode = elementList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

      /*----- Continue encoding element entries. See example in Section 11.3.2 ---- */

      /* Complete nested container encoding */
      retCode = elementList.EncodeComplete(encIter, success);
   }

   /* now that it is done, complete msgKey attrib encoding. */
   retCode = reqMsg.EncodeKeyAttribComplete(encIter, success);

   /* retCode should be CodecReturnCode.ENCODE_EXTENDED_HEADER */
   /* encode extended header as non-RWF type using non-RWF encode methods */
```

```
    {
        retCode = encIter.EncodeNonRWFInit(nonRWFBuffer);
        /* now encode extended header using its own specific encode methods -
        Ensure that we do not exceed nonRWFBuffer.Length */
        /* we could copy into the nonRWFBuffer or use it with other encode methods */
        nonRWFBuffer.Data().Put(encExtendedHeader.Data());
        retCode = encIter.EncodeNonRWFComplete(nonRWFBuffer, success);
    }

    retCode = reqMsg.EncodeExtendedHeaderComplete(encIter, success);
    /* retCode should be CodecReturnCode.ENCODE_CONTAINER */
    /* encode message payload to match ContainerType */
    {
        elementList.ApplyHasStandardData();
        /* now encode nested container using its own specific encode methods */
        if ((retCode = elementList.EncodeInit(encIter, null, 0)) < CodecReturnCode.SUCCESS)

        /*----- Continue encoding element entries. See example in Section 11.3.2 ---- */

        /* Complete nested container encoding */
        retCode = elementList.EncodeComplete(encIter, success);
    }
    /* now that specified msgKey attrib, extendedHeader and payload are done, complete message encoding.
    */
    retCode = reqMsg.EncodeComplete(encIter, success);
}
```

Code Example 41: `Msg` Encoding Example #1, `EncodeInit` / `EncodeComplete` Use

12.2.10.3 Msg Encoding Example 2

The following code sample demonstrates **Msg** encoding and shows the use of **Encode** with pre-encoded **MsgKey** attribute information and payload. While this example demonstrates error handling for the initial encode function, it omits additional error handling to simplify the example (though it should still be performed).

```
/* EXAMPLE 2 - EncodeMsg with pre-encoded msgKey.Attrib and pre-encoded payload, no extendedHeader */

/* Populate and encode a refreshMsg */
IRefreshMsg refreshMsg = new Msg();
refreshMsg.MsgClass = MsgClasses.REFRESH; /* message is a refresh */
refreshMsg.DomainType = (int)DomainType.MARKET_PRICE;
refreshMsg.ContainerType = DataTypes.FIELD_LIST;
/* Use the stream Id corresponding to the request, because it is in reply to a request, it's solicited */
refreshMsg.StreamId = 6;
/* Populate stream and data state information. This is required on an IRefreshMsg */
refreshMsg.State.StreamState(StreamStates.OPEN);
refreshMsg.State.DataState(DataStates.OK);
/* Populate flags for refresh message members and behavior - because this in response to a request
This should be solicited, msgKey should be present, single part refresh so it is complete,
and also want the concrete qos of the stream */
refreshMsg.ApplySolicited();
refreshMsg.ApplyHasMsgKey();
refreshMsg.ApplyRefreshComplete();
refreshMsg.ApplyHasQos();
refreshMsg.ApplyClearCache();
/* Populate msgKey to specify a ServiceId, a name with type of RIC (which is default NameType) and Attrib
*/
refreshMsg.MsgKey.ApplyHasServiceId();
refreshMsg.MsgKey.ApplyHasName();
refreshMsg.MsgKey.ApplyHasAttrib();
refreshMsg.MsgKey.ServiceId = 1;
/* Specify name and length of name. Because this is a RIC, no NameType is required. */
refreshMsg.MsgKey.Name.Data("TRI");
/* Msg Key attribute info is pre-encoded, should be set in encodedAttrib */
refreshMsg.MsgKey.AttribContainerType = DataTypes.ELEMENT_LIST;
/* assuming encodedAttrib Buffer contains the pre-encoded msgKey attribute info with data and length
populated */
refreshMsg.MsgKey.EncodedAttrib = encodedAttrib;
/* assuming encodedPayload Buffer contains the pre-encoded payload information with data and length
populated */
refreshMsg.EncodedDataBody = encodedPayload;
/* encode message - assumes that encIter is already populated with buffer and version information,
store return value to determine success or failure */
/* Because this method expects all portions to be populated and pre-encoded, all Message encoding is
complete after this returns. */
if ((retCode = refreshMsg.Encode(encIter)) < CodecReturnCode.SUCCESS)
{
   /* error condition - switch our success value to false so we can roll back */
   success = false;
   /* print out message with return value string, value, and text */
   Console.WriteLine("Error encountered with Msg.Encode. Error code: {0}",
      retCode.GetAsString());
}
```

Code Example 42: Msg Encoding Example #2, Encode Use

12.2.10.4 Msg Decoding Interfaces

Msg contains common members that can identify the specific message class or domain type. When decoding, you must use the **Msg** interface (because the **MsgClass** is not known until after the message is decoded). Once decoded, the **Msg** can be cast to the appropriate message interface. Because **MsgKey** is optional and specified on a per-message class basis, do not use **Msg.MsgKey** until the specific message class flags are consulted to determine whether the **MsgKey** is present.

A decoded **Msg** structure provides access to the encoded content of the message. You can further decode the message's content by invoking the specific contained type's decode function.

DECODE INTERFACE	DESCRIPTION
Decode	Decodes **Msg** header members. Any **MsgKey** attribute information remains encoded unless the user chooses to decode it. This can be accomplished by setting the **EncodedAttrib** buffer on a separate **DecodeIterator** or by calling **Msg.DecodeKeyAttrib**followed by decode functions for the specified **AttribContainerType**. Any message payload content will be described by the message's **ContainerType** member and will be present in the **EncodedDataBody**. This can be decoded by calling the **ContainerType**'s specific decode methods using the same **DecodeIterator** or by setting the **EncodedDataBody** on a new decode iterator. Any **ExtendedHeader** information is expected to be decoded by using a separate **DecodeIterator**. This method will decode from the **Buffer** to which the passed in **DecodeIterator** refers.
DecodeKeyAttrib	Prepares the **DecodeIterator** to decode **Msg.MsgKey.EncodedAttrib** information. This method expects the same **DecodeIterator** as was used with **Msg.Decode** and the **Msg.MsgKey** member that was populated by calling . This populates **EncodedData** with an encoded entry. After this method returns, you can call the **MsgKey.AttribContainerType** decode methods to decode attribute information. If you do not want to decode **MsgKey** attribute information, you can decode the payload by using the **ContainerType**'s decode methods after **Msg.Decode** returns.

Table 139: Msg Decode Methods

12.2.10.5 Msg Decoding Example

The following code sample demonstrates how to decode an **Msg**. This sample code uses a switch statement to decode the message's content. Typically an application would invoke the specific container type decoder for the housed type or use a switch statement to allow for a more generic message decoding. The example uses the same **DecodeIterator** when decoding the **MsgKey.EncodedAttrib** and the message payload. An application could optionally use a new **DecodeIterator** by setting the **EncodedAttrib** or **EncodedDataBody** on a new iterator. To simplify the sample code, some error handling is omitted.

```
/* decode contents into the Msg structure */
if ((retCode = msg.Decode(decIter)) >= CodecReturnCode.SUCCESS)
{
    /* We can cast to the appropriate message class for convenience or use the accessor methods */
    /* Use the ease of use accessor to get the msgKey if it exists on whatever msgClass this is */
    IMsgKey key = msg.MsgKey;
    /* If we have a key and it has attribute information, decode it */
    if (key is not null && key.CheckHasAttrib())
    {
        /* Need to set up the decodeIterator to expect decoding of attribute information, otherwise
        it assumes we are decoding the payload */
        retCode = msg.DecodeKeyAttrib(decIter, key);
        switch (key.AttribContainerType())
        {
            case DataTypes.FIELD_LIST:
                retCode = fieldList.Decode(decIter, null);
```

```
            /* Continue decoding field entries. See example in Section 11.3.1 */
         break;
         case DataTypes.ELEMENT_LIST:
            retCode = elementList.Decode(decIter, null);
         /* Continue decoding element entries. See example in Section 11.3.2 */
         break;
         /* full switch statement omitted to shorten sample code */
      }
   }
   /* Decode any contained payload information */
   switch (msg.ContainerType)
   {
      case DataTypes.NO_DATA:
         Console.WriteLine("No payload contained in message.");
      break;
      case DataTypes.FIELD_LIST:
         retCode = fieldList.Decode(decIter, null);
         /* Continue decoding field entries. See example in Section 11.3.1 */
      break;
      case DataTypes.ELEMENT_LIST:
         retCode = elementList.Decode(decIter, null);
         /* Continue decoding element entries. See example in Section 11.3.2 */
      break;
      /* full switch statement omitted to shorten sample code */
   }
}
else
{
   /* decoding failure tends to be unrecoverable */
   Console.WriteLine("Error encountered with Msg.Decode. Error code: {0}",
      retCode.GetAsString());
}
```

Code Example 43: Msg Decoding Example

12.2.10.6 EncodeIterator Utility Methods

The Enterprise Transport API provides the following **EncodeIterator** utility methods for use with the **Msg**.

METHOD	DESCRIPTION
ReplaceStreamId	Takes an encoded message and replaces the **StreamId** without re-encoding the message. For more details on the **StreamId**, refer to Section 12.1.3.
ReplaceSeqNum	Takes an encoded message and replaces the **SeqNum** without re-encoding the message.
ReplaceGroupId	Takes an encoded message and replaces the **GroupId** without re-encoding the message. For more information about group use, refer to Section 13.4.
ReplacePostId	Takes an encoded message and replaces the **PostId** without re-encoding the message. For more information, refer to Section 13.9.
ReplaceStreamState	Takes an encoded message and replaces the **StreamState** without re-encoding the message. For more information about state values, refer to Section 11.2.7.
ReplaceDataState	Takes an encoded message and replaces the **DataState** without re-encoding the message. For more information about state values, refer to Section 11.2.7.
ReplaceStateCode	Takes an encoded message and replaces the **State.Code** without re-encoding the message. For more information about state values, refer to Section 11.2.7.
SetConfInfoInUpdatesFlag UnsetConfInfoInUpdatesFlag	Sets or unsets the **RefreshMsgFlags.CONF_INFO_IN_UPDATES** flag on an encoded buffer.
SetGenericCompleteFlag UnsetGenericCompleteFlag	Sets or unsets the **GenericMsg.MESSAGE_COMPLETE** flag on an encoded buffer.
SetMsgKeyInUpdatesFlag UnsetMsgKeyInUpdatesFlag	Sets or unsets the **RefreshMsgFlags.MSG_KEY_IN_UPDATES** flag on an encoded buffer.
SetNoRefreshFlag UnsetNoRefreshFlag	Sets or unsets the **RefreshMsgFlags.NO_REFRESH** flag on an encoded buffer.
SetRefreshCompleteFlag UnsetRefreshCompleteFlag	Sets or unsets the **RefreshMsgFlags.REFRESH_COMPLETE** flag on an encoded buffer.
SetSolicitedFlag UnsetSolicitedFlag	Sets or unsets the **RefreshMsgFlags.SOLICITED** flag on an encoded buffer.
SetStreamingFlag UnsetStreamingFlag	Sets or unsets the **RefreshMsgFlags.STREAMING** flag on an encoded buffer.

Table 140: EncodeIterator Utility Methods

12.2.10.7 DecodeIterator Utility Methods

The Enterprise Transport API provides the following **DecodeIterator** utility methods for use with the **Msg**.

NOTE: Multiple Extract* calls on the same encoded message will likely be less efficient than a single call to **Msg.Decode**.

METHOD	DESCRIPTION
ExtractMsgClass	Takes an encoded message and returns the **MsgClass** information without fully decoding the message header.
ExtractDomainType	Takes an encoded message and returns the **DomainType** information without fully decoding the message header.
ExtractStreamId	Takes an encoded message and returns the **StreamId** information without fully decoding the message header. For more details on the **StreamId**, refer to Section 12.1.3.
ExtractSeqNum	Takes an encoded message and returns the **SeqNum** information without fully decoding the message header.

Table 141: DecodeIterator Utility Methods

13 Advanced Messaging Concepts

13.1 Multi-Part Message Handling

IRefreshMsg, **IPostMsg**, and **IGenericMsg** all support splitting payload content across multiple message parts, commonly referred to as ***message fragmentation***. Each message part includes relevant message header information along with the part's payload, where payload can be combined by following the modification semantics associated with the specific **ContainerType** (for specific container details, refer to Section 11.3). Message fragmentation is typically used to split large payload information into smaller, more manageable pieces. The size of each message part can vary, and is controlled by the application that performs the fragmentation. Often, sizes are chosen based on a specific transport layer frame or packet size.

When sending a multi-part message, several message members can convey additional part information. Each message class that supports fragmentation has an optional **PartNum** member that can order and ensure receipt of every part of the message. For consistency and compatibility with LSEG Real-Time Distribution System components, **PartNum** should begin with **0** and increment by one for each subsequent part. Several container types have an optional **TotalCountHint** value. This can convey information about the expected entry count across all message parts, and often helps size needed storage or display for the message contents.

These message classes have an associated **COMPLETE** flag value (specifically **RequestMsgFlags.REFRESH_COMPLETE**, **PostMsgFlags.POST_COMPLETE**, and **GenericMsgFlags.MESSAGE_COMPLETE**). A flag value of **COMPLETE** indicates the final part of a multi-part message (or that the message is a single-part and no subsequent parts will be delivered).

For both streaming and non-streaming information, other messages might arrive between parts of a fragmented message. For example, it is expected that update messages be received between individual parts of a multi-part refresh message. Such updates indicate changes to data being received on the stream and should be applied according to the modification semantics associated with the **containerType** of the payload. If non-streaming, no additional messages should be delivered after the final part.

If a transport layer is used, messages can fan out in the order in which they are received. On a transport where reliability is not guaranteed and the order can be determined by a sequence number, special rules should be used by consumers when processing a multi-part message. The following description explains how a multi-part refresh message can be handled. After the request is issued, any messages received on the stream should be stored and properly ordered based on sequence number. When an application encounters the first part of the **IRefreshMsg**, the application should process the part and note its sequence number. The application can drop (i.e., not process) stored messages with earlier sequence numbers. When the application encounters the next part of the **IRefreshMsg**, the application should first process any stored message with a sequence number intermediate between this refresh part and the previous part then the application should process the refresh part. This process should continue until the final part of the **IRefreshMsg** is encountered, at which time any remaining stored messages with a later sequence number should be processed and the stream's data flow can continue as normal.

13.2 Stream Priority

Consumers use **IRequestMsg** to indicate the stream's level of importance, conveyed by the priority information. When a consumer is aggregating streams on behalf of multiple users, the priority typically corresponds to the number of users interested in the particular stream. A consumer can increase or decrease a stream's associated priority information by issuing a subsequent request message on an already open stream.

A Provider application tracks the priority of each of its open streams. If the consumer reaches some kind of item count limitation (i.e., the maximum allowable number of streams), the provider can employ a preemption algorithm. Specific details must be defined by the provider application. The LSEG Real-Time Advanced Distribution Hub uses the combination of **PriorityCount** and **PriorityClass** to preempt items when the user's allowable cache list size is exceeded. LSEG Real-Time Advanced Distribution Hub always preempts the item with the lowest **PriorityCount** within the **PriorityClass** and then provides an **IStatusMsg** with a **StreamState** of **StreamStates.CLOSED_RECOVER** for the item.

Priority is represented by a **PriorityClass** value and a **PriorityCount** value.

- The ***priority class*** indicates the general importance of the stream to the consumer.
- The ***priority count*** indicates the stream's specific importance within the priority class.

The **PriorityClass** value takes precedence over any **PriorityCount** value. For example, a stream with a **PriorityClass** of 5 and **PriorityCount** of 1 has a higher overall priority than a stream with a **PriorityClass** of 3 and a **PriorityCount** of 10,000.

Because priority information is optional on a **IRequestMsg**:

- If priority information is not present on an initial request to open a stream, it is assumed that the stream has a **PriorityClass** and a **PriorityCount** of 1.
- If priority information is not present on a subsequent request message on an open stream, this means that the priority has not changed and previously stored priority information continues to apply.

If a consumer aggregates identical streams, the consumer should use the highest **PriorityClass** value. Individual **PriorityCount** values are always combined on a per-**PriorityClass** basis.

For example, if a consumer application combines three identical streams:

- One with **PriorityClass** 3 and **PriorityCount** 5
- One with **PriorityClass** 2 and **PriorityCount** 10
- One with **PriorityClass** 3 and **PriorityCount** of 1

In this case, the aggregate priority information would be **PriorityClass** 3 (i.e., the highest **PriorityClass**) and **PriorityCount** of 6 (the combined **PriorityCount** values for that class level).

13.3 Stream Quality of Service

A consumer can use **IRequestMsg** to indicate the desired QoS for its streams. This can be a request for a specific QoS or a range of qualities of service, where any value within the range will satisfy the request. The **IRefreshMsg** includes the QoS used to indicate the QoS being provided for a stream. When issuing a request, the QoS specified on the request typically matches the advertised QoS of the service, as conveyed via the Source Directory domain model. For more information, refer to the *Enterprise Transport API C# Edition LSEG Domain Model Usage Guide*.

- An initial request containing only **IRequestMsg.Qos** indicates a request for the specified QoS. If a provider cannot satisfy this QoS, the request should be rejected.
- An initial request containing both **IRequestMsg.Qos** and **IRequestMsg.WorstQos** sets the range of acceptable QoSs. Any QoS within the range, inclusive of the specified **Qos** and **WorstQos**, will satisfy the request. If a provider cannot provide a QoS within the range, the provider should reject the request.

When a provider responds to an initial request, the **IRefreshMsg.Qos** should contain the actual QoS being provided for the stream. Subsequent requests issued on the stream should not specify a range as the QoS has been established for the stream.

Because QoS information is optional on an **IRequestMsg** some special handling is required when it is absent.

- If neither **Qos** nor **WorstQos** are specified on an initial request to open a stream, it is assumed that any QoS will satisfy the request.
- If QoS information is absent on a subsequent reissue request, it is assumed that QoS, timeliness, and rate conform to the stream's currently established settings.
- If QoS information is absent in an initial **IRefreshMsg**, this should be assumed to have a **Timeliness** of **QosTimeliness.REALTIME** and a **Rate** of **QosRates.TICK_BY_TICK**. On any subsequent solicited or unsolicited refresh, this should be assumed to match any QoS already established by the initial **IRefreshMsg**.

To determine whether components require QoS information on initial and reissue requests, refer to the documentation for the specific component.

13.4 Item Group Use

You can use item groups to efficiently update the state for multiple item streams via a single group status message (instead of using multiple, individual item status messages). Each open data stream is assigned an item group. This information is associated with the stream through the **IRefreshMsg.GroupId** (refer to Section 12.2.2) or **IStatusMsg.GroupId** (refer to Section 12.2.4) members. Once established, item group information can be modified via a subsequent **IStatusMsg** or **IRefreshMsg** containing a different **GroupId** affiliation.

Item groups are defined on a per-service basis. While two item groups can have the same **GroupId**, each group's **ServiceId** will be unique. A consumer application should track **ServiceId-GroupId** pairings to ensure the correct sets of items are modified whenever group status messages are received. A provider can establish item group assignments according to the application's needs, but must maintain the uniqueness of each item group within a service. For example, a provider that aggregates multiple upstream services into a single downstream service might establish a different item group for each aggregated service. Thus, should an upstream service become unavailable, the provider can mark all items as being suspect while items from other upstream services remain in their prior state.

13.4.1 Item Group Buffer Contents

The consuming application should treat data (which may be of varying length) contained in the **GroupId** buffer as opaque. A simple memory comparison operation can determine whether two groups are equivalent. The actual data contained in the **GroupId** buffer is a collection of one or more unsigned two-byte, unsigned integer values, where each two-byte value is appended to the end of the current **GroupId Buffer**. Providers that combine multiple data sources must ensure that the item groups in the resulting service are unique, which can be accomplished by appending an additional two-byte value to each on-passed **GroupId**.

For example, the following figure depicts two non-interactive provider applications, each publishing item streams belonging to specific services and item groups.

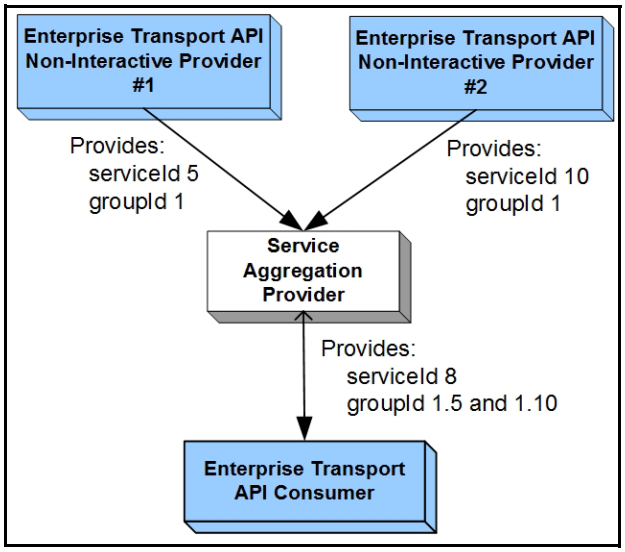


Figure 34. Item Group Example

Though the providers in this diagram use the same **GroupId** for an item, using different **ServiceId**s makes items unique. Both providers communicate with an application that consumes data from both services, aggregates the data into a single service, and then distributes the data to consumer applications. To ensure uniqueness to downstream components, the service aggregation provider appends additional identifiers to the group information it receives from the provider applications. In this example, the aggregation device modifies **ServiceId 5**, **GroupId 1** into a **GroupId** of **1.5** and **ServiceId 10**, **GroupId 1** into a **GroupId** of **1.10**. If for any reason non-interactive provider #1's service becomes unavailable, the aggregation device can send a single group status message to inform the consumer that all items belonging to **GroupId 1.5** are suspect. This would have no impact to any items belonging to **GroupId 1.10**.

13.4.2 Item Group Utility Functions

The Transport API provides the following utility methods for use with and modification of the **GroupId Buffer**.

METHOD	DESCRIPTION
GroupId (from **IRefreshMsg** and **IStatusMsg**)	Takes a populated **Msg** structure, determines if **GroupId** information is present and if available, returns it; NULL otherwise.
DecodeIterator.ExtractGroupId	Takes an encoded message and returns the **GroupId** without fully decoding the message header.
	NOTE: Multiple **DecodeIterator.Extract*** calls on the same encoded message will likely be less efficient than a single call to **Msg.** Decode.
EncodeIterator.ReplaceGroupId	Takes an encoded message and replaces the **GroupId** without re-encoding the message.

Table 142:

13.4.3 Group Status Message Information

Information regarding state changes and the merging of item groups occurs via group status messages. A group status message is communicated via the Source Directory domain message model. Specific group information is contained in the Directory's Group **FilterEntry** which corresponds to the specific service associated with the group.

- For more specific information, refer to the Source Directory Domain section in the *Transport API C# Edition LSEG Domain Model Usage Guide*.
- For a decision table providing example behavior for various state combinations, refer to .

NOTE: If an application does not subscribe to the Source Directory's group filter, the application will not receive group status messages. This can result in potentially incorrect item state information, as relevant status information might be missed.

13.4.4 Group Status Responsibilities by Application Type

Dissemination and handling of group status information is distributed across providers and consumers. This section discusses responsibilities by application type.

A provider application (interactive or non-interactive) is responsible for:

- Assigning and providing item group id values. This is accomplished by specifying the **IRefreshMsg.GroupId** or **IStatusMsg.GroupId** for all provided content[1].
- If a group of items becomes unavailable (i.e., an upstream service or provider goes down), group status messages should be sent out for all affected item groups. These are sent via the Source Directory domain.

  For more information about group status messages (including specific message content and formatting), refer to the *Transport API C# Edition LSEG Domain Model Usage Guide*.
- If items become available again, recovery should occur and items' states should be updated via a subsequent **IRefreshMsg** or **IStatusMsg** provided to any downstream components interested in the item.

A consumer application is responsible for:

- Subscribing to the item group filter when requesting Source Directory information.

  For more information about the item group filter and group status messages (including specific message content and formatting), refer to the *Transport API C# Edition LSEG Domain Model Usage Guide*.

1. This does not include administrative domains such as Login, Source Directory, and Dictionary.

- If group status changes are received, the state change should be propagated to all items associated with the indicated group, as noted by the **IRefreshMsg.GroupId** or **IStatusMsg.GroupId** provided with the item stream.
- Any recovery should follow **SingleOpen** and **AllowSuspectData** rules, as described in the *Transport API C# Edition LSEG Domain Model Usage Guide*.

13.5 Single Open and Allow Suspect Data Behavior

A consumer application can specify desired item recovery and state transition information on its Login domain **IRequestMsg** using the **SingleOpen** and **AllowSuspectData** **MsgKey** attributes. A providing application can acknowledge support for the behavior in the Login domain **IRefreshMsg**, in which case the provider performs certain state transitions. This section offers a high-level description of item recovery and state transition behavior modifications.

- ***Single open*** behavior allows a consumer application to open an item stream once and have an upstream component handle stream recovery (if needed). With single open enabled, a consumer should not receive a **StreamState** of **CLOSED_RECOVER**, as the providing application should convert to **SUSPECT** and attempt to recover on the consumer's behalf. If a stream is **CLOSED**, this will be propagated to the consumer application.
- ***Allow suspect data*** behavior indicates whether an application can tolerate an open stream with a **DataState** of **SUSPECT**, or if it is preferable to have the stream closed. If an application indicates that it does not wish to allow **SUSPECT** streams to remain open, the providing application should transition the **StreamState** to **CLOSED_RECOVER**.

If the providing application does not support either behavior, the application should indicate such a restriction in the Login domain's **IRefreshMsg**. For additional information, including on the **DomainTypes.LOGIN** domain definition, refer to the *Enterprise Transport API C# Edition LSEG Domain Model Usage Guide*.

The following table shows how a provider can convert messages to correspond with the consumer's **SingleOpen** and **AllowSuspectData** settings. The first column in the table shows the actual **StreamState** and **DataState**. Each subsequent column shows how this state information can be modified to follow the column's specific **SingleOpen** and **AllowSuspectData** settings. If a **SingleOpen** and **AllowSuspectData** configuration causes a behavioral contradiction (e.g., **SingleOpen** indicates that the provider should handle recovery, but **AllowSuspectData** indicates that the consumer does not want to receive suspect status), the **SingleOpen** configuration takes precedence.

NOTE: The Transport API does not perform special processing based on the **SingleOpen** and **AllowSuspectData** settings. The provider application must perform any necessary conversion.

ACTUAL STATE INFORMATION	CONVERSION WHEN: SINGLEOPEN = 1 ALLOWSUSPECTDATA = 1	CONVERSION WHEN: SINGLEOPEN =1 ALLOWSUSPECTDATA = 0	CONVERSION WHEN: SINGLEOPEN = 0 ALLOWSUSPECTDATA = 1	CONVERSION WHEN: SINGLEOPEN = 0 ALLOWSUSPECTDATA = 0
StreamState = OPEN **DataState** = SUSPECT	No conversion required	No conversion required	No conversion required	**StreamState** = CLOSED_RECOVER **DataState** = SUSPECT
StreamState = CLOSED_RECOVER **DataState** = SUSPECT	**StreamState** = OPEN **DataState** = SUSPECT	**StreamState** = OPEN **DataState** = SUSPECT	No conversion required	No conversion required

Table 143: SingleOpen and AllowSuspectData Effects

13.6 Pause and Resume

The Transport API allows applications to send or receive requests to pause or resume content flow on a stream.

- Issuing a ***pause*** on a stream can result in the temporary stop of **IUpdateMsg** flow.
- Issuing a ***resume*** on a paused stream restarts the **IUpdateMsg** flow.

Pause and resume can help optimize bandwidth by pausing streams that are only temporarily not of interest, instead of closing and re-requesting a stream. Though a pause request may be issued on a stream, it does not guarantee that the contents of the stream will actually be paused. Additionally, if the contents of the stream are paused, state-conveying messages can still be delivered (i.e., status messages and unsolicited refresh messages). Pause and resume is only valid for data streams instantiated as streaming (**RequestMsgFlags.STREAMING**). The consumer application is responsible for continuing to handle all delivered messages, even after the issuance of a pause request.

A consumer application can request to pause an individual item stream by issuing **IRequestMsg** with the **RequestMsgFlags.PAUSE** flag set. This can occur on the initial **IRequestMsg** or via a subsequent **IRequestMsg** on an established stream (i.e., a reissue). If a pause is issued on the initial request, it should always result in the delivery of the initial **IRefreshMsg** (this conveys initial state, permissioning, Quality of Service, and group association information necessary for the stream). A paused stream remains paused until a resume request is issued. To resume data flow on a stream a consumer application can issue a subsequent **IRequestMsg** with the **RequestMsgFlags.STREAMING** flag set.

If a provider application receives a pause request from a consumer, it can choose to pause the content flow or continue delivering information. When pausing a stream, where possible, the provider should aggregate information updates until the consumer application resumes the stream. When resuming, an aggregate update message should be delivered to synchronize the consumer's information to the current content. However, if data cannot be aggregated, resuming the stream should result in a full, unsolicited **IRefreshMsg** to synchronize the consumer application's information to a current state.

A pause request issued on the **StreamId** associated with a user's login is interpreted as a request to ***pause all*** streams associated with the user. A pause all request is only valid for use on an already established login stream and cannot be issued on the initial login request. A 'pause all' request affects open streams only. Thus, newly-requested streams begin in a resumed state. After a pause all request, the application can choose to either resume individual item streams or resume all streams. A ***resume all*** will result in all paused streams being transitioned to a resumed state. This is performed by issuing a subsequent **IRequestMsg** with the **RequestMsgFlags.STREAMING** flag set using the **StreamId** associated with the applications login.

For more information about the **IRequestMsg** and the **RequestMsgFlags.PAUSE** or **RequestMsgFlags.STREAMING** flag values, refer to Section 12.2.1.

A provider application can indicate support for pause and resume behavior by sending the **MsgKey** attribute **SupportOptimizedPauseResume** in the Login domain **IRefreshMsg**. For more details on the Login **DomainType** (**DomainTypes.LOGIN**), refer to the *Transport API C# Edition LSEG Domain Model Usage Guide*.

13.7 Batch Messages

Applications can use the Enterprise Transport API to send and / or receive batch messages as a more efficient way to handle requests, reissues, or closes of multiple items. When a consumer application wishes to open multiple similar items at once, or close multiple streams, it may perform the operation using a single message instead of sending a message for each individual stream.

NOTE: Batch messages use the **`ElementEntry`** names **`:ItemList`** and **`:StreamIdList`** in message payloads. These names follow a namespacing scheme in which a name's content prior to the character **`:`** indicates a namespace. LSEG reserves the empty namespace (e.g., **`:Element`**), while other namespaces are left for custom element names (e.g., **`Customer:Element`**)

This section defines the following types of operations that can be performed using a batch message:

- ***Batch Requests***, to open streams for items that have that have different names but for which other key content (if any) is identical.
- ***Batch Reissues***, to change attributes of multiple open streams such as priority, or to pause or resume streams.
- ***Batch Closes***, to close multiple open streams.
- A provider application can indicate support for each form of batch messaging by sending a bitmask in the **`MsgKey`** attribute **`SupportBatchRequests`** in the Login domain **`IRefreshMsg`**. For more details on the Login domainType (**DomainTypes.LOGIN**) and the general use of batch messages, refer to the Enterprise Transport API *LSEG Domain Model Usage Guide*. The Transport API **LSEG.Eta.Rdm** namespace defines the batch request-related enumerations and element name string constants.

13.7.1 Batch Request

Batch requesting can be leveraged across all non-administrative[2] domain model types, where specific usage and support should be indicated in the model definition. If an item requested as part of a batch is not available, the provider should send a **IStatusMsg** on the stream (this is handled in the same manner as an individual item request).

A consumer application can issue a batch request by using an **IRequestMsg** with the **RequestMsgFlags.HAS_BATCH** flag set and including a specifically formatted payload. The payload should contain an **ElementList** along with an **ElementEntry** named **:ItemList**. Because payload content can include customer-defined portions and LSEG-defined portions, the Transport API uses a name-spacing scheme. Any content in an element **Name** prior to **:** is used as name space information (e.g., **Customer:Element**). LSEG reserves the empty name space (e.g., **:Element**). The **LSEG.Eta.Rdm.ElementNames** defines batch request-related enumeration and element name buffer constant.

The **:ItemList** contains an **Array**, where the **Array.PrimitiveType** is **DataTypes.ASCII_STRING**. Each contained string (populated in an **Buffer**) corresponds to a requested name. The **MsgKey** contents will be applied to all names in the list, and a **MsgKey.Name** (or **MsgKeyFlags.HAS_NAME_TYPE**) should not be present.

When a provider application receives a batch request, it should respond on the same stream with an **IStatusMsg** that acknowledges receipt of the batch by indicating the **DataState** is **DataStates.OK** and **StreamState** is **StreamStates.CLOSED**. The stream on which the batch request was sent (i.e., the 'batch stream') then closes, because all additional responses are provided on individual streams, and thus no reissuing is possible on a batch stream. The **:ItemList** should be traversed to obtain each requested name and the batch **IRequestMsg.MsgKey** content should be associated with each item. If any request cannot be fulfilled, the provider should send an **IStatusMsg** to close the stream and indicate the reason, using the stream that corresponds to that particular item (for further details, refer to Section 12.2.4).

Assignment of **StreamId** values for all requested items is sequential, beginning with (**1** + **StreamId**) of the batch **IRequestMsg**. Because a consumer requests the batch, positive **StreamId** values should be assigned. For example, if the batch request uses **StreamId** **20** and requests ten items, the **IStatusMsg** response to the batch request would be delivered on **StreamId** **20**, then the first item in the list receives a response with **StreamId** **21**, the second item with **StreamId** **22**, etc.By setting the initial **StreamId**, the consumer application can control the resultant **StreamId** range, ensuring enough available **StreamId** values exist to allocate identifiers for all requested items.

Any view information (described in Section 13.8) included in a batch request should be applied for each item in the request. If a consumer application wants to reissue any item that was requested as part of a batch, the application can issue a subsequent **IRequestMsg** on that item's **StreamId**.

A provider application can indicate support for batch request handling by sending the **MsgKey** attribute **SupportBatchRequests** in the Login domain **IRefreshMsg**.

- For an example of encoding a batch request, refer to Section 13.7.2.
- For more information about **IRequestMsg** and **RequestMsgFlags.HAS_BATCH** flag values, refer to Section 12.2.1.
- For more information about **ElementList**, refer to Section 11.3.2.
- For more details on the Login **DomainType** (**DomainTypes.LOGIN**) and batch request use in general, see the *Enterprise Transport API C# Edition LSEG Domain Model Usage Guide*.

2. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. All other domains are considered non-administrative.

13.7.2 Batch Reissue

Consumers may use a batch reissue message to change attributes of multiple open streams (such as changing priority, or to pause or resume them) using a single **IRequestMsg**. In a batch reissue message, the consumer specifies a list of **StreamId**s in the message payload representing the streams it wishes to reissue. Batch reissues can be leveraged across all non-Login domain model types.

A consumer application can issue a batch reissue by using an **IRequestMsg** with the **RequestMsgFlags.HAS_BATCH** flag set and including a specifically formatted payload. The payload should contain an **ElementList** along with an **ElementEntry** named **:StreamIdList**.

The **:StreamIdList** contains an **Array**, where the **Array.PrimitiveType** is **DT_INT**. Each contained **StreamId** (populated in an **Int**) corresponds to the **StreamId** of an open stream. The stream attributes specified (e.g., specifying the **RequestMsgFlags.PAUSE** flag, or changes to **PriorityClass** and **PriorityCount**) will be applied to each **StreamId** in the list.

The consumer application may specify **StreamId**s from from any non-Login domain in the **:StreamIdList** of a batch reissue message; only the **StreamId** is needed to identify the stream. The **Qos**, **WorstQos**, **MsgKey**, **DomainType**, and **ExtendedHeader** of the **IRequestMsg** are not used (do not set the **RequestMsgFlags.HAS_QOS**, **RequestMsgFlags.HAS_WORST_QOS**, or **RequestMsgFlags.HAS_EXTENDED_HEADER** flags. Set **MsgKey.Flags** to **MsgKeyFlags.NONE**. LSEG recommends setting the **DomainType** to **DomainTypes.MARKET_PRICE**).As with a batch request, a provider should respond on the same stream with an **IStatusMsg** that acknowledges receipt of the batch by indicating the **DataState** is **OK** and **StreamState** is **CLOSED**, and the provider sends any additional responses on the individual streams. If any stream's reissue cannot be fulfilled, the provider should send an **IStatusMsg** on that stream to indicate the reason (for further details, refer to Section 12.2.4). If the provider is unable to process the batch message itself, it should use a **SUSPECT** **DataState** in the response to the batch message.

Consider the following interaction example:

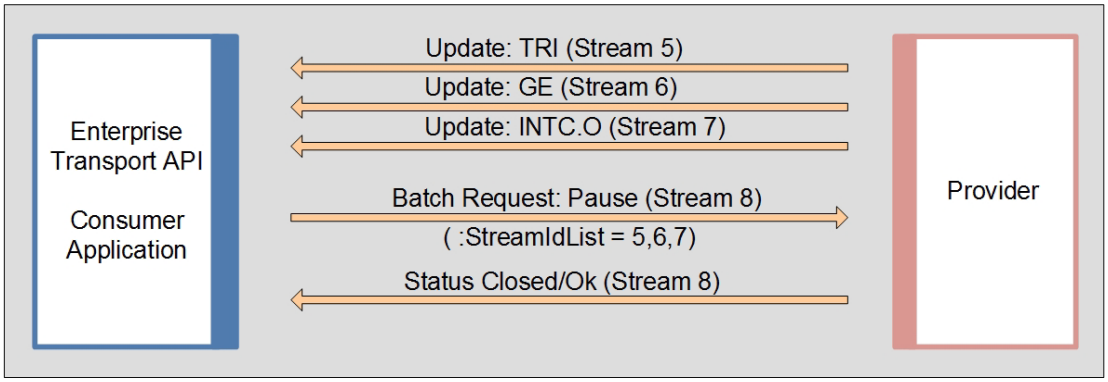


Figure 35. Batch Reissue (Pause) Interaction Example

In the above figure, the consumer currently has three items open on **StreamId**s **5**, **6**, and **7** to a provider that supports pausing those streams. The consumer wishes to pause these three streams, so it sends an **IRequestMsg** using an unused **StreamId**, 8. This message includes the **RequestMsgFlags.PAUSE** flag, and encodes the **StreamId**s **5**, **6**, and **7** in the **:StreamIdList** element. The provider then responds with an **IStatusMsg** on **StreamId** **8** to acknowledge the reissue message, and considers streams **5**, **6**, and **7** to be paused.

- For an example of encoding a batch reissue, refer to Section 13.7.5.
- For more information about **IRequestMsg** and **RequestMsgFlags.HAS_BATCH** flag values, refer to Section 12.2.1.
- For more information about the **ElementList**, refer to Section 11.3.2.

13.7.3 Batch Request Encoding Example

The following example demonstrates how to encode a batch request using an **RequestMsg**. The request is sent using a **StreamId** of **10** and contains an **:ItemList** of three items. Such a message should result in four responses:

- An **StatusMsg** delivered on **StreamId** **10** which indicates that the batch is being processed and closes the stream.
- Three **RefreshMsg**s are delivered, where the first item returns on **StreamId** **11**, the second on **StreamId** **12**, and the third on **StreamId** **13**.

To simplify the example, some error handling has been omitted; though applications should perform all appropriate error handling.

```
/* Example assumes encode iterator is properly initialized */
/* Create and populate request message with information pertaining to all items in batch, set batch flag
*/
reqMsg.MsgClass = MsgClasses.REQUEST; /* message is a request */
reqMsg.DomainType = (int)DomainType.MARKET_PRICE;
/* Set RequestMsgFlags.HAS_BATCH so provider application is alerted to batch payload */
reqMsg.ApplyHasQos();
reqMsg.ApplyStreaming();
reqMsg.ApplyHasBatch();
reqMsg.Qos.Timeliness(QosTimeliness.REALTIME);
/* Populate msgKey - no name should be provided as all names should be in payload */
reqMsg.MsgKey.ApplyHasNameType();
reqMsg.MsgKey.ApplyHasServiceId();
reqMsg.MsgKey.NameType = InstrumentNameTypes.RIC;
reqMsg.MsgKey.ServiceId = 5;
/* Payload type is an element list */
reqMsg.ContainerType = DataTypes.ELEMENT_LIST;
/* Populate streamId with value to start streamId assignment */
reqMsg.StreamId = 10; /* Batch status response should be delivered using streamId 10 */
/* Begin message encoding */
retCode = reqMsg.EncodeInit(encIter, 0);
{
   Array nameList = new Array();
   ArrayEntry nameEntry = new ArrayEntry();
   elementList.ApplyHasStandardData();
   /* now encode nested container using its own specific encode methods */
   retCode = elementList.EncodeInit(encIter, null, 0);
   /* Batch requests require an element with the name of :ItemList */
   elemEntry.Name().Data(":ItemList");
   elemEntry.DataType = DataTypes.ARRAY;
   /* encode array of item names in the element entry */
   retCode = elemEntry.EncodeInit(encIter, 0);
   {
      Buffer nameBuf = new Buffer();
      /* Encode the array and the names */
      nameList.PrimitiveType = DataTypes.ASCII_STRING;
      nameList.ItemLength = 0; /* Array will have variable length entries */
      retCode = nameList.EncodeInit(encIter);
      /* Populate first name in the list. This should use streamId 11 when the response comes */
      nameBuf.Data("TRI");
      nameEntry.Clear();
      nameEntry.Encode(encIter, nameBuf);
      /* Populate the second name in the list. This should use streamId 12 when the response comes */
      nameBuf.Data("GOOG.O");
      nameEntry.Clear();
      nameEntry.Encode(encIter, nameBuf);
```

```
        /* Populate the third name in the list. This should use streamId 13 when the response comes */
        nameBuf.Data("AAPL.O");
        nameEntry.Clear();
        nameEntry.Encode(encIter, nameBuf);
        /* List is complete, finish encoding array */
        retCode = nameList.EncodeComplete(encIter, true);
    } /* Complete the element encoding and then the element list */
    retCode = elemEntry.EncodeComplete(encIter, true);
    retCode = elementList.EncodeComplete(encIter, success);
}
/* now that :ItemList is encoded in the payload, complete the message encoding */
retCode = reqMsg.EncodeComplete(encIter, success);
```

Code Example 44: Batch Request Encoding Example

13.8 Dynamic View Use

Applications can use the Enterprise Transport API to send or receive requests for a dynamic view of a stream's content. A consumer application uses a ***dynamic view*** to specify a subset of data in which the application has interest. A provider can choose to supply only this requested subset of content across all response messages. Filtering content in this manner can reduce the volume of data that flows across the connection. View use can be leveraged across all non-administrative[3] domain model types, where the model definition should specify associated usage and support. Though a consumer might request a specific view, the provider might still send additional content and/or content might be unavailable (and not provided).

A consumer application can request a view through an **IRequestMsg** with the **RequestMsgFlags.HAS_VIEW** flag set and by including a specially-formatted payload. The payload should contain an **ElementList** along with:

- An **ElementEntry** for **:ViewType** which contains an **DataTypes.UINT** value indicating the specific type of view requested. Section 13.8.1 describes the currently defined **:ViewType** values.
- An **ElementEntry** for **:ViewData** which contains an **Array** populated with the content being requested. For instance, when specifying a **FieldId** list, the array would contain two-byte fixed length **DataTypes.INT** entries. The specific contents of the **:ViewData** are indicated in the definition of the **:ViewType**.

Because payload content can include customer-defined portions and LSEG-defined portions, the Enterprise Transport API uses a name-spacing scheme. Any content in the **Name** member prior to the colon (**:**) is used as name space information (e.g., **Customer:Element**). LSEG reserves the empty name space (e.g., **:Element**). View-related enumerations and element name string constants are defined in **LSEG.Eta.Rdm.ElementNames**.

If a consumer application wishes to change a previously-specified view, the same process can be followed by issuing a subsequent **IRequestMsg** using the same **StreamId** (a reissue). In this case, **:ViewData** would contain the newly desired view. If a reissue is required and the consumer wants to continue using the same view, the **IRequestMsg** should continue to include the **RequestMsgFlags.HAS_VIEW** flag,**:ViewType** or **:ViewData** are not required. Sending an **IRequestMsg** without the **RequestMsgFlags.HAS_VIEW** flag removes any view associated with a stream.

A provider application can receive a view request and determine an appropriate way to respond. Response content can be filtered to abide by the view specification, or the provider can send full/additional content. Several **State.code** values are available to convey view-related status. If a view's possible content changes (e.g., a previously requested field becomes available), an **IRefreshMsg** should be provided to convey such a change to the data. This refresh should follow the rules associated with solicited or unsolicited refresh messages.

A provider application can indicate support for dynamic view handling by sending the **MsgKey** attribute **supportViewRequests** in the Login domain **IRefreshMsg**.

- For details on **State.Code** values, refer to Section 11.2.7.6.
- For details on the **IRequestMsg** and **RequestMsgFlags.HAS_VIEW** flag values, refer to Section 12.2.1.
- For details on the **ElementList**, refer to Section 11.3.2.

3. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. Other domains are considered non-administrative.

- For rules associated with refresh messages, refer to Section 12.2.2.
- For details on the Login **DomainType** (**DomainTypes.LOGIN**) and general view use, refer to the *Transport API LSEG Domain Model Usage Guide*.

13.8.1 RDM ViewTypes Names

The following table defines the **LSEG.Eta.Rdm.ViewTypes**.

VIEW TYPE	DESCRIPTION
ViewTypes.FIELD_ID_LIST	Indicates that **:ViewData** contains an array populated with **FieldId** values. The array should specify a **PrimitiveType** of DataTypes.INT and a fixed two-byte **ItemLength**. For specific details about the **Array**, refer to Section 11.2.8.
ViewTypes.ELEMENT_NAME_LIST	Indicates that **:ViewData** contains an array populated with element **Name** values. The array should specify a **PrimitiveType** corresponding to the type used for the domain model's element names (e.g. **DataTypes.ASCII_STRING**). For specific details about the **Array**, refer to Section 11.2.8.

Table 144: RDM ViewTypes Values

13.8.2 Dynamic View RequestMsg Encoding Example

The following example demonstrates how to encode an **IRequestMsg** which specifies a **FieldId**-based view. The request asks for two fields, though it is possible that more will be delivered. For the sake of simplicity, some error handling is omitted from the example; though applications should perform all appropriate error handling.

```
/* Example assumes encode iterator is properly initialized */
/* Create and populate request message, set view flag */
reqMsg.MsgClass = MsgClasses.REQUEST; /* message is a request */
reqMsg.DomainType = (int)DomainType.MARKET_PRICE;
/* Set RequestMsgFlags.HAS_VIEW so provider application is alerted to view payload */
reqMsg.ApplyHasQos();
reqMsg.ApplyStreaming();
reqMsg.ApplyHasView();
reqMsg.StreamId = 15;
reqMsg.Qos.Timeliness(QosTimeliness.REALTIME);
/* Populate msgKey */
reqMsg.MsgKey.ApplyHasName();
reqMsg.MsgKey.ApplyHasNameType();
reqMsg.MsgKey.ApplyHasServiceId();
reqMsg.MsgKey.nameType = InstrumentNameTypes.RIC;
reqMsg.MsgKey.Name.Data("TRI");
reqMsg.MsgKey.ServiceId = 5;
/* Payload type is an element list */
reqMsg.ContainerType = DataTypes.ELEMENT_LIST;
/* Begin message encoding */
retCode = reqMsg.EncodeInit(encIter, 0);
{
   UInt viewTypeUInt = new UInt();
   Array fidList = new Array();
   ArrayEntry fidEntry = new ArrayEntry();
```

```
    elementList.ApplyHasStandardData();
    /* now encode nested container using its own specific encode methods */
    retCode = elementList.EncodeInit(encIter, null, 0);
    /* Initial view requests require two elements, one with the name of :ViewType and the other :ViewData
    */
    elemEntry.Name.Data(":ViewType");
    elemEntry.DataType = DataTypes.UINT;
    viewTypeUInt.Value(ViewTypes.FIELD_ID_LIST);
    retCode = elemEntry.Encode(encIter, viewTypeUInt);
    /* encode array of fieldIds in the element entry */
    elemEntry.Name.Data(":ViewData");
    elemEntry.DataType = DataTypes.ARRAY;
    retCode = elemEntry.EncodeInit(encIter, 0);
    {
        Int fieldIdInt = new Int();
        /* Encode the array and the fieldIds. FieldId list should be fixed two byte integers */
        fidList.PrimitiveType = DataTypes.INT;
        fidList.ItemLength = 2; /* Array will have fixed 2 byte length entries */
        retCode = fidList.EncodeInit(encIter);
        /* Populate first fieldId in the list. */
        /* Passed in as third parameter as data is not pre-encoded */
        fieldIdInt.Value(22); /* fieldId for BID */
        fidEntry.Clear();
        fidEntry.Encode(encIter, fieldIdInt);
        /* Populate the second fieldId in the list */
        fieldIdInt.Value(25); /* fieldId for ASK */
        fidEntry.Clear();
        fidEntry.Encode(encIter, fieldIdInt);
        /* List is complete, finish encoding array */
        retCode = fidList.EncodeComplete(encIter, true);
    } /* Complete the element encoding and then the element list */
    retCode = elemEntry.EncodeComplete(encIter, true);
    retCode = elementList.EncodeComplete(encIter, success);
}
/* now that :ViewType and :ViewData are encoded in the payload, complete the message encoding */
retCode = reqMsg.EncodeComplete(encIter, success);
```

Code Example 45: View Request Encoding Example

13.9 Posting

The Enterprise Transport API provides ***posting*** functionality: an easy way for consumer applications to publish content to upstream components for further distribution. Posting is similar in concept to unmanaged publications or SSL Inserts, where content originates from a consuming application and flows upstream to some destination component. After arriving at the destination component, content can be incorporated into cache and republished to downstream applications with an acknowledgment issued to the posting application. Via posting, the Enterprise Transport API can push content to all non-administrative[4] domain model types, where specific usage and support should be indicated in the model definition. **IPostMsg.PostUserInfo** payloads can include any container type; often this is an **Msg** (**DataTypes.MSG**). When payload is an **Msg**, the contained message should be populated with any contributed header and payload information. For additional information on how to encode and decode container types, refer to Section 11.3.

The Transport API offers two types of posting:

- ***On-stream posting***, where you send an **IPostMsg** on an existing data stream, in which case posted content corresponds to the stream on which it is posted. The upstream route of an on-stream post is determined by the route of the data stream over which it is sent. On-stream posting should be directed towards the provider that sources the item. Because on-stream post messages are flowing on the stream related to the item, a **MsgKey** is not required. If the content is republished by the upstream provider, the consumer should receive it on the same stream over which they posted it.
- ***Off-stream posting***, where you send an **IPostMsg** on the **StreamId** associated with the users Login. Thus a consumer application can post data, regardless of whether they have an open stream associated with the post-related item. Post messages issued on this stream must indicate the specific **DomainType** and **MsgKey** corresponding to the content being posted. Off-stream posting is typically routed by configuration values on the upstream components.

The **IPostMsg** contains ***Visible Publisher Identifier*** information (contained in **IPostMsg.PostUserInfo**), which identifies the user who posted it. **IPostMsg.PostUserInfo** must be populated and consists of:

- **PostUserId**: which should be an ID associated with the user. For example, a Data Access Control System user ID or if unavailable, a process id)
- **PostUserAddr**: which should contain the IP address[5] of the application posting the content.

Optionally, such information can be carried along with republished **IRefreshMsgs**, **IUpdateMsgs**, or **IStatusMsgs** so that receiving consumers can identify the posting user. For more information about the Visible Publisher Identifier, refer to Section 13.11.

IPostMsg.PermData permissions the user who posts data. If the payload of the **IPostMsg** is another nested message type (i.e., **IRefreshMsg**) with permission data, such permission data can change the permission expression of the item being posted. However, if the permission data for the nested message is the same as the permission data on the **IPostMsg**, the nested message does not need to include permission data. The permission data is used in conjunction with the **IPostMsg.PostUserRights**, which indicate:

- Whether the posting user can create or destroy items in the cache of record.
- Whether the user has the ability to change the **PermData** associated with an item in the cache of record.

Each independent post message flowing in a stream should use a unique **PostId** to distinguish between individual post messages and those used for acknowledgment purposes. The consumer can request an acknowledgment upon the successful receipt and processing of content. When the provider responds, the **IAckMsg.ackId** should be populated using the **IPostMsg.PostId** to match the two messages. **seqNum** information can also be used during acknowledgment.

NOTE: Provider applications that support posting must have the ability to properly acknowledge posted content.

You can split content across multiple **IPostMsg** messages. When sending a multi-part **IPostMsg**, the **PostId** should match all parts of the post. If the consumer requests an acknowledgment, the **SeqNum** is also required. Each part should be acknowledged by the receiving component, where each **IAckMsg.AckId** is populated using the **IPostMsg.PostId**, and each **IAckMsg.SeqNum** is populated using the **IPostMsg.SeqNum**. Each part of the **IPostMsg** should specify a **PartNum**, where the first part begins with **0**. The final part of a multi-part **IPostMsg** should have the **PostMsgFlags.POST_COMPLETE** flag set to indicate that it is the final part.

4. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. Other domains are considered non-administrative.
5. The **Transport.HostByName**method can be used to help obtain the IP address of the application. Refer to Section 10.14.

A provider application can indicate support for posting and acknowledgment use by sending the **MsgKey** attribute **SupportOmmPost** in the Login domain **IRefreshMsg**.

- For more information on the **IPostMsg**, refer to Section 12.2.7.
- For more information on the **IAckMsg**, refer to Section 12.2.8.
- For more information on managing multi-part **IPostMsg**s, refer to Section 13.1.
- For more details on the Login **DomainType** (**DomainTypes.LOGIN**), see the *Transport API LSEG Domain Model Usage Guide*.

13.9.1 Post Message Encoding Example

The following example demonstrates how to encode an off-stream **PostMsg** with a nested **Msg**.

```
/* Example assumes encode iterator is properly initialized */
/* Create and populate post message - since it's off stream, msgKey is required */
IPostMsg postMsg = new Msg();
postMsg.MsgClass = MsgClasses.POST;
postMsg.StreamId = 1; /* Use streamId of the Login stream for off-stream posting */
postMsg.DomainType = (int)DomainTypes.MARKET_PRICE; /* domainType of data being posted */
/* off stream requires key. Post asking for ACK and including postId and seqNum for ack purposes.
Since it's a single part post, the POST_COMPLETE flag must be set as well */
postMsg.AapplyHasMsgKey();
postMsg.ApplyAck();
postMsg.ApplyHasPostId();
postMsg.ApplyHasSeqNum();
postMsg.ApplyPostComplete();
/* Populate msgKey with information about the item being posted to */
postMsg.MsgKey.ApplyHasName();
postMsg.MsgKey.ApplyHasNameType();
postMsg.MsgKey.ApplyHasServiceId();
postMsg.MsgKey.NameType = InstrumentNameTypes.RIC;
postMsg.MsgKey.Name.Data("TRI");
postMsg.MsgKey.ServiceId = 5;
/* populate postId with a unique ID for this posting, this and seqNum are used on ack */
postMsg.PostId = 42;
postMsg.SeqNum = 124;
/* postUserInfo must be populated, with processId and IP address */
postMsg.PostUserInfo.UserId = Environment.ProcessId;
postMsg.PostUserInfo.UserAddrFromString(Dns.GetHostAddresses(Dns.GetHostName())
                        .Where(ip => ip.AddressFamily ==
    System.Net.Sockets.AddressFamily.InterNetwork)!.FirstOrDefault()?.ToString());
/* put a message in the postMsg */
postMsg.ContainerType = DataTypes.MSG;
/* Begin message encoding */
retCode = postMsg.EncodeInit(encIter, 0);
{
    /* populate the message that is in the payload of the post message */
    IUpdateMsg updMsg = new Msg();
    updMsg.MsgClass = MsgClasses.UPDATE;
    updMsg.StreamId = 1;
    updMsg.DomainType = (int)DomainType.MARKET_PRICE;
    updMsg.UpdateType = UpdateEventTypes.QUOTE;
    updMsg.ContainerType = DataTypes.FIELD_LIST;
```

```
    /* begin encoding of the payload message */
    retCode = updMsg.EncodeInit(encIter, 0);
    /* Continue encoding field list contents of the message - see example in Section 11.3.1 */
    /* Complete the postMsg payload messages encoding */
    retCode = updMsg.EncodeComplete(encIter, true);
}
/* now complete encoding of postMsg */
retCode = postMsg.EncodeComplete(encIter, success);
```

Code Example 46: Off-Stream Posting Encoding Example

13.9.2 Post Acknowledgement Encoding Example

The following example demonstrates how to encode an **AckMsg**.

```
/* Example assumes encode iterator is properly initialized */
/* Create and populate ack message with information used to acknowledge the post */
IAckMsg ackMsg = new Msg();
ackMsg.MsgClass = MsgClasses.ACK;
ackMsg.DomainType = (int)DomainType.MARKET_PRICE;
ackMsg.StreamId = 1; /* Ack should be sent back on same stream that post came on */
ackMsg.ApplyHasSeqNum();
/* Acknowledge the post from above, use its postId and seqNum */
ackMsg.AckId = postMsg.PostId;
ackMsg.SeqNum = postMsg.SeqNum;
/* No payload associated with this acknowledgment */
ackMsg.ContainerType = DataTypes.NO_DATA;
/* Since there is no payload, no need for Init/Complete as everything is in the msg header */
retCode = ackMsg.Encode(encIter);
```

Code Example 47: Post Acknowledgment Encoding Example

13.10 Visible Publisher Identifier

The Enterprise Transport API offers the ***Visible Publisher Identifier*** feature, which inserts originating publisher data into message payloads. You can use Visible Publisher Identifier data to identify the user ID and user address for users who post, insert, or publish to an interactive service or to a non-interactive service cache on the LSEG Real-Time Advanced Distribution Hub.

Visible Publisher Identifier data is present on Post, Refresh, Update, and Status Messages and is carried in **`IPostMsg.PostUserInfo`**, which consists of:

- Post user ID (i.e., publisher ID)
- Post user address (i.e., publisher address)

They can both contain values assigned by and specific to the application.

An **`IPostMsg`** contains data (in **`IPostMsg.PostUserInfo`**) that identifies the user who posts content. For this reason, **`IPostMsg.PostUserInfo`** must be populated with:

- **`PostUserInfo.UserId`**: An ID associated with the posting user. The application should determine what information to put into this field (e.g., a Data Access Control System user ID).
- **`PostUserInfo.UserAddr`**: The address of the posting user's application that posted the contents. The application should decide what information to put into this field, for example, an IP address.

Optionally, this data can be republished by the provider in **`IRefreshMsg`**s, **`IUpdateMsg`**s, or **`IStatusMsg`**s so that receiving consumers can identify the posting user.

The Enterprise Transport API allows the Visible Publisher Identifier to be populated on Post messages submitted by a consumer application before the post is sent over the network.

Provider applications receive Visible Publisher Identifier data in Post Messages. Additionally, OMM providers can optionally set Visible Publisher Identifier data in their response messages. If the upstream provider is an intermediary device getting data from an upstream source, then the intermediary device will route Visible Publisher Identifier data as set in the **`IPostMsg`** to the upstream source. The final publisher in the upward chain decides whether to set Visible Publisher Identifier data in its published responses.

Visible Publisher Identifier information can also be communicated using Field IDentifiers defined in the publisher component. For further details refer to the publishing component's documentation.

13.10.1 Example: Encoding PostUserInfo into a Refresh Message

The following example shows how **`PostUserInfo`** is set in the **`RefreshMsg`** by the Provider application.

```
m_RefreshMsg.Clear();
m_RefreshMsg.MsgClass = MsgClasses.REFRESH;
m_RefreshMsg.ApplyHasPostUserInfo();
m_RefreshMsg.PostUserInfo.UserAddr = postUserInfo.UserAddr;
m_RefreshMsg.PostUserInfo.UserId = postUserInfo.UserId;
m_RefreshMsg.DomainType = itemInfo.Attributes.DomainType;

try
{
    m_RefreshMsg.PostUserInfo.UserAddr = BitConverter.ToUInt32(Dns.GetHostAddresses(Dns.GetHostName())
            .Where(ip => ip.AddressFamily ==
    System.Net.Sockets.AddressFamily.InterNetwork)!.FirstOrDefault()?.GetAddressBytes());
}
catch (Exception e)
{
```

Table 145: Setting PostUserInfo in Provider Example

```
    Console.WriteLine("Populating postUserInfo failed. Dns.GetHostAddresses(Dns.GetHostName())
   exception: " + e.Message);
}

m_RefreshMsg.PostUserInfo.UserId = Process.GetCurrentProcess().Id;
```

Table 145: Setting PostUserInfo in Provider Example

13.10.2 Example: Decoding PostUserInfo from Refresh Message

The following example shows the Consumer application using **PostUserInfo** to obtain Visible Publisher Identifier data in the **processMarketPriceResponse()** method.

```
/* The Visible Publisher Identifier (VPI) can be found within the PostUserInfo.
/* This will provide both the publisher ID and publisher address. Consumer can obtain the
/* information from the msg - The partially decoded message. */
if (refreshMsg.CheckHasPostUserInfo())
{
    Console.WriteLine("\nReceived RefreshMsg for stream " + refreshMsg.StreamId);
    Console.WriteLine(" from publisher with user ID: " + refreshMsg.PostUserInfo.UserId +
                      " at user address: " +
   refreshMsg.PostUserInfo.UserAddrToString(refreshMsg.PostUserInfo.UserAddr) + "\n");
}
```

Code Example 48: Consumer Using PostUserInfo to obtain Visible Publisher Information

13.10.3 Example: Encoding PostUserInfo into a Post Message

The following example populates Visible Publisher Identifier on Post messages submitted by a Transport API consumer application in the **EncodePostWithMsg()** method internally used by the **SendOnstreamPostMsg()** function. It encodes a **PostMsg** and populates the **PostUserInfo** with the IP address and process ID of the machine running the application.

```
m_PostMsg.Clear();

// set-up message
m_PostMsg.MsgClass = MsgClasses.POST;
m_PostMsg.StreamId = StreamId;
m_PostMsg.DomainType = (int)DomainType.MARKET_PRICE;
m_PostMsg.ContainerType = DataTypes.MSG;

// Note: post message key not required for on-stream post
m_PostMsg.ApplyPostComplete();
m_PostMsg.ApplyAck();
m_PostMsg.ApplyHasPostId();
m_PostMsg.ApplyHasSeqNum();
m_PostMsg.ApplyHasMsgKey();
m_PostMsg.ApplyHasPostUserRights();
m_PostMsg.PostId = m_NextPostId++;
```

Table 146: Populating Visible Publisher Identifier on Post Messages Submitted by Transport API Consumer Application

```
m_PostMsg.SeqNum = m_NextSeqNum++;

try
{
    m_PostMsg.PostUserInfo.UserAddr = BitConverter.ToUInt32(Dns.GetHostAddresses(Dns.GetHostName())
        .Where(ip => ip.AddressFamily ==
   System.Net.Sockets.AddressFamily.InterNetwork)!.FirstOrDefault()?.GetAddressBytes());
}
catch (Exception e)
{
    Console.WriteLine("Populating postUserInfo failed. Dns.GetHostAddresses(Dns.GetHostName())
   exception: " + e.Message);
    return CodecReturnCode.FAILURE;
}
m_PostMsg.PostUserInfo.UserId = Process.GetCurrentProcess().Id;

m_PostMsg.PostUserRights = PostUserRights.CREATE | PostUserRights.DELETE;
```

Table 146: Populating Visible Publisher Identifier on Post Messages Submitted by Transport API Consumer Application

13.10.4 Example: Decoding PostUserInfo from Post Message

The following example shows how **PostUserInfo** is set in the Provider application from the message sent by the Consumer.

```
IPostMsg postMsg = msg;

// if the post message contains another message, then use the
// "contained" message as the update/refresh/status
if (postMsg.ContainerType == DataTypes.MSG)
{
    m_NestedMsg.Clear();
    ret = m_NestedMsg.Decode(dIter);
    if (ret != CodecReturnCode.SUCCESS)
    {
        error = new Error
        {
            ErrorId = TransportReturnCode.FAILURE,
            Text = $"Unable to decode msg"
        };

        return ret;
    }
    switch (m_NestedMsg.MsgClass)
    {
        case MsgClasses.REFRESH:
            m_NestedMsg.MsgClass = MsgClasses.REFRESH;
            int flags = m_NestedMsg.Flags;
            flags |= RefreshMsgFlags.HAS_POST_USER_INFO;
            flags &= ~RefreshMsgFlags.SOLICITED;
```

Table 147: Setting PostUserInfo in Provider Example

```
            m_NestedMsg.Flags = flags;

            m_NestedMsg.PostUserInfo.UserAddr  = postMsg.PostUserInfo.UserAddr;
            m_NestedMsg.PostUserInfo.UserId = postMsg.PostUserInfo.UserId;
            if (UpdateItemInfoFromPost(itemInfo, m_NestedMsg, dIter, out error) !=
CodecReturnCode.SUCCESS)
            {
              ret = SendAck(chnl, postMsg, NakCodes.INVALID_CONTENT, error != null ? error!.Text : "",
out error);
                if (ret != CodecReturnCode.SUCCESS)
                {
                    return ret;
                }
            }

            break;

        case MsgClasses.UPDATE:
            m_NestedMsg.MsgClass = MsgClasses.UPDATE;
            m_NestedMsg.Flags |= UpdateMsgFlags.HAS_POST_USER_INFO;
            m_NestedMsg.PostUserInfo.UserAddr = postMsg.PostUserInfo.UserAddr;
            m_NestedMsg.PostUserInfo.UserId = postMsg.PostUserInfo.UserId;
            if (UpdateItemInfoFromPost(itemInfo, m_NestedMsg, dIter, out error) !=
CodecReturnCode.SUCCESS)
            {
              ret = SendAck(chnl, postMsg, NakCodes.INVALID_CONTENT, error != null ? error!.Text : "",
out error);
                if (ret != CodecReturnCode.SUCCESS)
                {
                    return ret;
                }
            }
            break;

        case MsgClasses.STATUS:
            m_NestedMsg.MsgClass =MsgClasses.STATUS;
            m_NestedMsg.Flags |= StatusMsgFlags.HAS_POST_USER_INFO;
            m_NestedMsg.PostUserInfo.UserAddr = postMsg.PostUserInfo.UserAddr;
            m_NestedMsg.PostUserInfo.UserId = postMsg.PostUserInfo.UserId;
            if (m_NestedMsg.CheckHasState() && m_NestedMsg.State.StreamState() == StreamStates.CLOSED)
            {
                // check if the user has the rights to send a post that
                // closes an item
                if (postMsg.CheckHasPostUserRights() || postMsg.PostUserRights == 0)
                {
                ret = SendAck(chnl, postMsg, NakCodes.INVALID_CONTENT, "client has insufficient rights
to close/delete an item", out error);
                    if (ret != CodecReturnCode.SUCCESS)
                        return ret;
                }
```

Table 147: Setting PostUserInfo in Provider Example

```
                }
                break;
            default:
                break;
        }
}
```

Table 147: Setting PostUserInfo in Provider Example

13.11 UserAuthn Authentication

The Enterprise Transport API can use the UserAuthn Authentication feature, which provides enhanced authentication functionality when used with the LSEG Real-Time Distribution System and Data Access Control System. This feature requires LSEG Real-Time Distribution System 3.1 or later.

A consumer or non-interactive provider application can pass a token generated from a token generator based on the user's credentials to LSEG Real-Time Distribution System. LSEG Real-Time Distribution System passes this token to a local token authenticator for verification.

The token must be encoded in the initial login **`IRequestMsg`** with:

- **`MsgKey.Name`** set to one byte of **0x00**, and
- **`MsgKey.NameType`** set to **UserIdTypes.USER_AUTHN_TOKEN**.

The token will be in the **`MsgKey.attrib`**'s **`ElementList`**, with an **`ElementEntry`** named **`authenticationToken`**.

For additional information, refer to the *Transport API LSEG Domain Model Usage Guide* for encoding and decoding Login messages, and the *UserAuthn Authentication User Manual*[6] for details on setting up the LSEG Real-Time Distribution System and the token generator.

13.12 Private Streams

The Enterprise Transport API provides ***private stream*** functionality, an easy way to ensure delivery of content only between a stream's two endpoints. Private streams behave in a manner similar to standard streams, with the following exceptions:

- All data on a private stream flow between the end provider and the end consumer of the stream.
- Intermediate components do not fan out content (i.e., do not distribute it to other consumers).
- Intermediate components should not cache content.
- In the event of connection or data loss, intermediate components do not recover content. All private stream recovery is the responsibility of the consumer application.

These behaviors ensure that only the two endpoints of the private stream send or receive content associated with the stream. As a result, a private stream can exchange identifying information so the provider can validate the consumer, even through multiple intermediate components (such as might exist in an LSEG Real-Time Distribution System deployment). After a private stream is established, content can flow freely within the stream, following either existing market data semantics (i.e., private Market Price domain) or any other user-defined semantics (i.e., bidirectional exchange of **`IGenericMsg`**s).

In standard streams, if an application attempts to open the same stream using multiple, unique **`StreamId`** values, provider applications reject subsequent requests. With private streams, even if the streams' identifying information (**`MsgKey`**, domain type, etc.) matches, multiple private stream instances can be opened, allowing for the possibility of different user data contained in each private stream.

To establish a private stream, a consumer observes the following general process:

- The consumer application issues a request for the item data it wants on a private stream. This **`IRequestMsg`** should include the **`RequestMsgFlags.PRIVATE_STREAM`** flag. If user-identifying information is required, it should be described in the respective domain message model definition.
- When a capable OMM provider application receives a request for a private stream, if it can honor the request, the provider application should acknowledge that the stream is established and is private by sending:
  - **`IRefreshMsg`** with the **`RequestMsgFlags.PRIVATE_STREAM`** flag; typically sent when there is immediate content to provide in the response.
  - **`IStatusMsg`** with the **`StatusMsgFlags.PRIVATE_STREAM`** flag; typically sent when there is no immediate content to provide in the response but the provider wants to acknowledge the establishment of the private stream.
  - **`IAckMsg`** with the **`AckMsgFlags.PRIVATE_STREAM`** flag; can be used as an alternative to the **`IStatusMsg`**.

6. For further details on UserAuthn Authentication, refer to the *UserAuthn Authentication User Manual*, accessible on MyAccount in the Data Access Control System product documentation set.

- When the consumer application receives the above acknowledgment, the private stream is established and content can be exchanged. The **PRIVATE_STREAM** flag is no longer required on any messages exchanged within the stream.
- If the consumer application receives any other message, or the above messages without their respective **PRIVATE_STREAM** flag, the private stream is not established and the consumer should close the stream if it does not want to consume a standard stream.

Some content might be available as both standard stream and private stream delivery mechanisms. In the standard stream case, all users see the same stream content. Because private streams can support user identification, each private stream instance can contain modified or additional content tailored for the specific user.

Some content might be available only as standard streams, in which case the private stream request is ignored or rejected by sending an **IStatusMsg** with a **StreamState** of **StreamStates.CLOSED** or **StreamStates.CLOSED_RECOVER**, or by responding to the request with a standard stream (e.g., no **PRIVATE_STREAM** flag).

Some content might be available only as a private stream (e.g., some kind of restricted data set where users must be validated). If an OMM provider has private-only content, the provider can indicate to downstream applications that its content is private by redirecting standard stream requests.

If a standard stream **IRequestMsg** is received for private-only content, a provider can:

- Inform downstream applications that its content is private by sending a message (including the **MsgKey**), with a **StreamState** of **StreamStates.REDIRECTED** in an:
  - **IStatusMsg** including the **StatusMsgFlags.PRIVATE_STREAM** flag; typically sent when there is not any content to provide as part of the redirect.
  - **IRefreshMsg** including the **RequestMsgFlags.PRIVATE_STREAM** flag; typically sent when there is some kind of content to provide as part of the redirect.
- If the consumer application sees a **StreamState** of **StreamStates.REDIRECTED** and a **PRIVATE_STREAM** flag, it can issue a new **IRequestMsg** and use the **RequestMsgFlags.PRIVATE_STREAM** flag. This process follows standard stream redirect logic and the private stream establishment protocol described above.

13.13 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 *Transport API C# Edition DACSLock API Developers Guide*.

The following example code illustrates how to create a DACSLock.

```
List<uint> peList = new() { 1001 };
AuthorizationLock authLock = new(serviceID: 261, AuthorizationLock.OperatorEnum.OR, peList);
AuthorizationLockData lockData = new();
AuthorizationLockStatus retStatus = new();
LockResultEnum result = authLock.GetLock(lockData, retStatus);
if (result == LockResultEnum.LOCK_SUCCESS)
{
    Console.WriteLine("Success - GetLock()");
}
else
{
    Console.WriteLine("Failure - GetLock()");
}
```

Code Example 49: Creating a DACSLOCK for Publishing Permission Data

Appendix A Item and Group State Decision Table

The following table describes various item and group status combinations and the common results in terms of application behavior. Though applications are not required to follow this behavior, the information is provided as an example of one possible behavior.

- For general information about **`State`**, refer to Section 11.2.7.
- For general information about Item Groups, refer to Section 13.4.
- For information about group status delivery and formatting, refer to the *Transport API LSEG Domain Model Usage Guide*.
- For information about how item state is conveyed, refer to Section 12.2.2 and Section 12.2.4.

STATUS TYPE	STREAM STATE	DATA STATE	DESCRIPTION	APPLICATION ACTION
Item	StreamStates.OPEN	DataStates.OK	Stream is open and streaming. Data is ok.	No action.
Item	StreamStates.OPEN	DataStates.SUSPECT	Stream is open and streaming. Data is suspect.	No action. Upstream device should recover data and onpass.
Item	StreamStates.NON_STREAMING	DataStates.OK	Stream was opened as non-streaming. Data was provided for item and was OK.	No action.
Item	StreamStates.CLOSED	DataStates.SUSPECT	Stream is closed. Data is suspect.	Application can attempt to recover this or another service or provider.
Item	StreamStates.CLOSED_RECOVER	DataStates.SUSPECT	Stream is closed, but may become available on same service and provider later. Data is suspect.	Application can attempt to recover to this or another service or provider.
Item	StreamStates.CLOSED	DataStates.OK	Stream is closed. Data provided was OK.	Application can attempt to recover to this or another service or provider. This state combination is not common.
Item	StreamStates.CLOSED_RECOVER	DataStates.OK	Stream is closed, but may become available on same service and provider later. Data provided was OK.	Application can attempt to recover to this or another service or provider. This state combination is not common.

Table 148: Item and Group State Decision Table

STATUS TYPE	STREAM STATE	DATA STATE	DESCRIPTION	APPLICATION ACTION
Group	StreamStates.OPEN	DataStates.NO_CHANGE	All streams associated with the group remain open. Previous state communicated via item or group status continues to apply.	No action.
Group	StreamStates.OPEN	DataStates.SUSPECT	All streams associated with the group remain open. Data on all streams associated with the group is suspect.	Application should fan out **`DataState`** change to all items that are part of the group. Upstream device should recover data and onpass.
Group	StreamStates.OPEN	DataStates.OK	All streams associated with the group remain open. Data on all streams associated with the group is ok.	Application should fan out **`DataState`** change to all items that are part of the group. This state combination is not common. Typically individual item statuses are used to change items from suspect to ok.
Group	StreamStates.CLOSED_RECOVER	DataStates.SUSPECT	All streams associated with the group are closed, but may become available on same service and provider later. Data on all streams associated with the group is suspect.	Application should fan out **`StreamState`** and **`DataState`** change to all items that are part of the group. Application can attempt to recover to this or another service or provider.

Table 148: Item and Group State Decision Table (Continued)

LSEG DATA & ANALYTICS

Document ID: ETACSharp340L2UM.250
Date of issue: May 2025