

Transport API Java Edition

V3.0

DEVELOPERS GUIDE

JAVA EDITION



© Thomson Reuters 2015, 2016. All rights reserved.

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

This document contains information proprietary to Thomson Reuters and may not be reproduced, disclosed, or used in whole or part without the express written permission of Thomson Reuters.

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

Nothing in this document is intended, nor does it, alter the legal obligations, responsibilities or relationship between yourself and Thomson Reuters as set out in the contract existing between us.

Contents

Chapter 1	Introduction	1
1.1	About this Manual	1
1.2	Audience	1
1.3	Programming Language.....	1
1.4	Acronyms and Abbreviations	1
1.5	References	2
1.6	Documentation Feedback	3
1.7	Document Conventions	3
1.7.1	<i>Typographic</i>	3
1.7.2	<i>Diagrams</i>	3
Chapter 2	Product Description	5
2.1	What is the Transport API?	5
2.2	Transport API Features	6
2.2.1	<i>General Capabilities</i>	6
2.2.2	<i>Consumer Applications</i>	6
2.2.3	<i>Provider Applications: Interactive</i>	7
2.2.4	<i>Provider Applications: Non-Interactive</i>	7
2.3	Performance and Feature Comparison.....	7
2.3.1	<i>Java Garbage</i>	8
2.3.2	<i>Use of Assertions</i>	8
2.4	Functionality: Which API to Choose?	9
Chapter 3	Consumers and Providers	13
3.1	Overview	13
3.2	Consumers	14
3.2.1	<i>Subscriptions: Request/Response</i>	15
3.2.2	<i>Batches</i>	15
3.2.3	<i>Views</i>	16
3.2.4	<i>Pause and Resume</i>	17
3.2.5	<i>Symbol Lists</i>	18
3.2.6	<i>Posting</i>	21
3.2.7	<i>Generic Message</i>	22
3.2.8	<i>Private Streams</i>	22
3.3	Providers	24
3.3.1	<i>Interactive Providers</i>	25
3.3.2	<i>Non-Interactive Providers</i>	26
Chapter 4	System View	28
4.1	System Architecture Overview	28
4.2	Advanced Distribution Server (ADS)	29
4.3	Advanced Data Hub (ADH)	30
4.4	Elektron	31
4.5	Data Feed Direct	32
4.6	Internet Connectivity via HTTP and HTTPS.....	33
4.7	Direct Connect	34
Chapter 5	Model and Package Overviews.....	35
5.1	Transport API Models	35
5.1.1	<i>Open Message Model (OMM)</i>	35

5.1.2	<i>Reuters Wire Format (RWF)</i>	35
5.1.3	<i>Domain Message Model</i>	35
5.2	Packages	36
5.2.1	<i>Transport Package</i>	36
5.2.2	<i>Codec Package</i>	36
Chapter 6	Building an OMM Consumer	37
6.1	Overview	37
6.2	Establish Network Communication	37
6.3	Perform Login Process.....	38
6.4	Obtain Source Directory Information	38
6.5	Load or Download Necessary Dictionary Information	39
6.6	Issue Requests and/or Post Information	39
6.7	Log Out and Shut Down	39
6.8	Additional Consumer Details	40
Chapter 7	Building an OMM Interactive Provider	41
7.1	Overview	41
7.2	Establish Network Communication	41
7.3	Perform Login Process.....	42
7.4	Provide Source Directory Information	42
7.5	Provide or Download Necessary Dictionaries	42
7.6	Handle Requests and Post Messages	43
7.7	Disconnect Consumers and Shut Down	43
7.8	Additional Interactive Provider Details	44
Chapter 8	Building an OMM NIP	45
8.1	Overview	45
8.2	Establish Network Communication	45
8.3	Perform Login Process.....	46
8.4	Perform Dictionary Download	46
8.5	Provide Source Directory Information	46
8.6	Provide Content	47
8.7	Log Out and Shut Down	47
8.8	Additional NIP Details	47
Chapter 9	Transport Package Detailed View.	48
9.1	Concepts	48
9.1.1	<i>Transport Types</i>	49
9.1.2	<i>Channel Object</i>	50
9.1.3	<i>Server Object</i>	54
9.1.4	<i>Transport Error Handling</i>	54
9.1.5	<i>General Transport Return Codes</i>	55
9.1.6	<i>Application Lifecycle</i>	56
9.2	Initializing and Uninitializing the Transport.....	57
9.2.1	<i>Initialization and Uninitialization Method</i>	57
9.2.2	<i>Initialization Reference Counting with Example</i>	57
9.2.3	<i>Transport Locking Models</i>	58
9.3	Creating the Connection	59
9.3.1	<i>Network Topologies</i>	59
9.3.2	<i>Creating the Outbound Connection: Transport.connect Method</i>	62
9.3.3	<i>Transport.connect Outbound Connection Creation Example</i>	67
9.3.4	<i>Tunneling Connection Keep Alive</i>	68
9.4	Server Creation and Accepting Connections	69

9.4.1	<i>Creating a Listening Socket</i>	69
9.4.2	<i>Accepting Connection Requests</i>	75
9.4.3	<i>Compression Support</i>	77
9.5	Channel Initialization	78
9.5.1	<i>Channel.init Method</i>	78
9.5.2	<i>InProgInfo Object</i>	79
9.5.3	<i>Calling Channel.init</i>	79
9.5.4	<i>Channel.init Return Codes</i>	80
9.5.5	<i>Channel.init Example</i>	80
9.6	Reading Data	82
9.6.1	<i>Channel.read Method</i>	82
9.6.2	<i>ReadFlags Values</i>	83
9.6.3	<i>Channel.read Return Codes</i>	83
9.6.4	<i>Channel.read Example</i>	85
9.7	Writing Data: Overview	87
9.8	Writing Data: Obtaining a Buffer	88
9.8.1	<i>Transport Buffer Management Channel Methods</i>	88
9.8.2	<i>Transport Buffer Management Server Method</i>	89
9.8.3	<i>Channel.getBuffer Return Values</i>	89
9.9	Writing Data to a Buffer	90
9.9.1	<i>Channel.write Method</i>	90
9.9.2	<i>WriteFlags Values</i>	91
9.9.3	<i>Compression</i>	91
9.9.4	<i>Fragmentation</i>	91
9.9.5	<i>Channel.write Return Codes</i>	92
9.9.6	<i>Channel.getBuffer and Channel.write Example</i>	93
9.10	Managing Outbound Queues	95
9.10.1	<i>Ordering Queued Data: WritePriorities</i>	95
9.10.2	<i>Channel.flush Method</i>	96
9.10.3	<i>Channel.flush Return Codes</i>	96
9.10.4	<i>Channel.flush Example</i>	97
9.11	Packing Additional Data into a Buffer.....	98
9.11.1	<i>Channel.packBuffer Return Values</i>	98
9.11.2	<i>Example: Channel.getBuffer, Channel.packBuffer, and Channel.write</i>	98
9.12	Ping Management	100
9.12.1	<i>Ping Timeout</i>	100
9.12.2	<i>Channel.ping Function</i>	101
9.12.3	<i>Channel.ping Return Values</i>	101
9.12.4	<i>Channel.ping Example</i>	102
9.13	Closing Connections	103
9.13.1	<i>Functions for Closing Connections</i>	103
9.13.2	<i>Close Connections Example</i>	103
9.14	Utility Methods.....	104
9.14.1	<i>General Transport Utility Methods</i>	104
9.14.2	<i>ChannelInfo Methods</i>	105
9.14.3	<i>multicastStats Methods</i>	107
9.14.4	<i>componentInfo Method</i>	107
9.14.5	<i>ServerInfo Methods</i>	107
9.14.6	<i>Channel.ioctl ioctlCodes</i>	108
9.14.7	<i>Server.ioctl ioctlCodes</i>	108
9.15	Tunneling	109
9.15.1	<i>Configuration</i>	109
9.15.2	<i>Proxy Authentication</i>	110

Chapter 10 Encoding and Decoding Conventions 116

10.1	Concepts	116
10.1.1	<i>Data Types</i>	116
10.1.2	<i>Composite Pattern and Nesting</i>	117
10.2	Encoding Semantics	118
10.2.1	<i>Init and Complete Suffixes</i>	118
10.2.2	<i>The Encode Iterator: Encodelterator</i>	118
10.2.3	<i>Content Roll Back with Example</i>	121
10.3	Decoding Semantics	122
10.3.1	<i>The Decode Iterator: Decodelterator</i>	122
10.3.2	<i>Functions for use with Decodelterator</i>	122
10.3.3	<i>Decodelterator: Basic Use Example</i>	123
10.4	Return Code Values.....	124
10.4.1	<i>Success Codes</i>	124
10.4.2	<i>Failure Codes</i>	126
10.4.3	<i>CodecReturnCodes Methods</i>	127
10.5	Versioning	127
10.5.1	<i>Protocol Versioning</i>	127
10.5.2	<i>Library Versioning</i>	128

Chapter 11 Data Package Detailed View..... 130

11.1	Concepts	130
11.2	Primitive Types.....	130
11.2.1	<i>Real</i>	134
11.2.2	<i>Date</i>	138
11.2.3	<i>Time</i>	139
11.2.4	<i>DateTime</i>	140
11.2.5	<i>Qos</i>	142
11.2.6	<i>State</i>	145
11.2.7	<i>Array</i>	150
11.2.8	<i>Buffer</i>	156
11.2.9	<i>RMTES Decoding</i>	158
11.3	Container Types.....	161
11.3.1	<i>FieldList</i>	165
11.3.2	<i>ElementList</i>	173
11.3.3	<i>Map</i>	180
11.3.4	<i>Series</i>	190
11.3.5	<i>Vector</i>	197
11.3.6	<i>FilterList</i>	206
11.3.7	<i>Non-RWF Container Types</i>	215
11.4	Permission Data.....	217
11.5	Summary Data	217
11.6	Set Definitions and Set-Defined Data	218
11.6.1	<i>Set-Defined Primitive Types</i>	219
11.6.2	<i>Set Definition Use</i>	222
11.6.3	<i>Set Definition Database</i>	225

Chapter 12 Message Package Detailed View 235

12.1	Concepts	235
12.1.1	<i>Common Message Interface</i>	235
12.1.2	<i>Message Key</i>	239
12.1.3	<i>Stream Identification</i>	242
12.2	Messages.....	244
12.2.1	<i>Request Message Interface</i>	244
12.2.2	<i>Refresh Message Interface</i>	247
12.2.3	<i>Update Message Interface</i>	250

12.2.4	<i>Status Message Interface</i>	252
12.2.5	<i>Close Message Interface</i>	254
12.2.6	<i>Generic Message Class</i>	255
12.2.7	<i>Post Message Interface</i>	257
12.2.8	<i>Acknowledgment Message Interface</i>	260
12.2.9	<i>Msg Encoding and Decoding</i>	261
Chapter 13 Advanced Messaging Concepts		271
13.1	Multi-Part Message Handling	271
13.2	Stream Priority	271
13.3	Stream Quality of Service	272
13.4	Item Group Use	273
13.4.1	<i>Item Group Buffer Contents</i>	273
13.4.2	<i>Item Group Utility Functions</i>	274
13.4.3	<i>Group Status Message Information</i>	274
13.4.4	<i>Group Status Responsibilities by Application Type</i>	274
13.5	Single Open and Allow Suspect Data Behavior	275
13.6	Pause and Resume.....	276
13.7	Batch Requesting	277
13.7.1	<i>Batch Request Usage</i>	277
13.7.2	<i>Batch RequestMsg Encoding Example</i>	278
13.8	Dynamic View Use	279
13.8.1	<i>RDM ViewTypes Names</i>	280
13.8.2	<i>Dynamic View RequestMsg Encoding Example</i>	280
13.9	Posting	283
13.9.1	<i>Post Message Encoding Example</i>	284
13.9.2	<i>Post Acknowledgement Encoding Example</i>	285
13.10	Private Streams.....	286
Appendix A Item and Group State Decision Table.....		288

List of Figures

Figure 1.	Network Diagram Notation	4
Figure 2.	UML Diagram Notation	4
Figure 3.	OMM-Based Product Offerings	5
Figure 4.	Transport API: Core Diagram.....	5
Figure 5.	TREP Infrastructure	13
Figure 6.	Transport API as a Consumer.....	14
Figure 7.	Batch Request.....	15
Figure 8.	View Request Diagram	16
Figure 9.	Symbol List: Basic Scenario.....	18
Figure 10.	Symbol List: Accessing the Entire ADS Cache	18
Figure 11.	Symbol List: Requesting Symbol List Streams via the Transport API Reactor	19
Figure 12.	Server Symbol List	20
Figure 13.	Posting into a Cache	21
Figure 14.	OMM Post with Legacy Inserts	22
Figure 15.	Private Stream Scenarios	23
Figure 16.	Provider Access Point	24
Figure 17.	Interactive Providers	25
Figure 18.	NIP: Point-To-Point	26
Figure 19.	NIP: Multicast	27
Figure 20.	Typical TREP Components	28
Figure 21.	Transport API and Advanced Distribution Server	29
Figure 22.	Transport API and the Advanced Data Hub.....	30
Figure 23.	Transport API and Elektron.....	31
Figure 24.	Transport API and Data Feed Direct.....	32
Figure 25.	Transport API and Internet Connectivity	33
Figure 26.	Transport API and Direct Connect	34
Figure 27.	Transport Application Lifecycle	56
Figure 28.	Unified TCP Network.....	59
Figure 29.	TCP Connection Creation	60
Figure 30.	Unified Multicast Network.....	60
Figure 31.	Segmented Multicast Network	61
Figure 32.	Multicast Connection Creation	61
Figure 33.	Transport API Server Creation.....	69
Figure 34.	Transport API Writing Flow Chart	87
Figure 35.	Channel.write Priority Scenario.....	95
Figure 36.	Transport API Consumer Application authenticating with a Proxy Server using NTLM.....	114
Figure 37.	Transport API Consumer Application Authenticating with a Proxy Server using Negotiate/Kerberos	115
Figure 38.	Transport API and the Composite Pattern	117
Figure 39.	Item Group Example	273

List of Tables

Table 1:	Acronyms and Abbreviations	1
Table 2:	API Performance Comparison	8
Table 3:	Capabilities by API	9
Table 4:	Channel Methods	50
Table 5:	Channel State Values	52
Table 6:	ConnectionType Values	52
Table 7:	Server Methods	54
Table 8:	Error Methods	54
Table 9:	General Transport Return Codes	55
Table 10:	Initialization and Uninitialization Methods	57
Table 11:	Locking Types	58
Table 12:	Transport.connect Method	62
Table 13:	ConnectOptions Methods	62
Table 14:	UnifiedNetworkInfo Method Options	64
Table 15:	SegmentedNetworkInfo Method Options	65
Table 16:	TcpOpts Method Option	65
Table 17:	MCastOpts Method Options	66
Table 18:	ShmemOpts Method Option	66
Table 19:	SeqMCastOpts Method Option	66
Table 20:	Transport.bind Method	69
Table 21:	BindOptions Methods	69
Table 22:	Server.accept Method	75
Table 23:	AcceptOptions Methods	75
Table 24:	CompressionTypes Values	77
Table 25:	Channel.init Method	78
Table 26:	InProgrInfo Methods	79
Table 27:	Channel.init TransportReturnCodes	80
Table 28:	Channel Method	82
Table 29:	ReadFlags Values	83
Table 30:	Channel.read TransportReturnCodes	83
Table 31:	Buffer Management Channel Methods	88
Table 32:	Buffer Management Server Methods	89
Table 33:	Channel.getBuffer TransportReturnCodes	89
Table 34:	Channel.write Function	90
Table 35:	WriteFlags	91
Table 36:	Channel.wri te TransportReturnCodes	92
Table 37:	WritePriorities Values	96
Table 38:	Channel.flush Method	96
Table 39:	Channel.flush TransportReturnCodes	96
Table 40:	Channel.packBuffer Method	98
Table 41:	Channel.packBuffer Return Values	98
Table 42:	Channel.ping method	101
Table 43:	Channel.ping TransportReturnCodes	101
Table 44:	Connection Closing Functionality	103
Table 45:	Transport Utility Methods	104
Table 46:	ChannelInfo Methods	105
Table 47:	multicastStats Methods	107
Table 48:	componentInfo Options	107
Table 49:	ServerInfo Methods	107
Table 50:	Channel.loctl_loctlCodes	108
Table 51:	Server.loctl_loctlCodes	108

Table 52: TunnelingInfo Methods	109
Table 53: CredentialInfo Methods	113
Table 54: EncoderIterator Utility Methods	119
Table 55: DecoderIterator Utility Methods	122
Table 56: Codec Package Success CodecReturnCodes	124
Table 57: Codec Package Failure CodecReturnCodes	126
Table 58: CodecReturnCodes Methods	127
Table 59: Codec Methods	128
Table 60: Library Version Utility Methods	128
Table 61: LibraryVersionInfo Methods	128
Table 62: Transport API Primitive Types	131
Table 63: DataTypes Methods	134
Table 64: Real Methods	134
Table 65: Rss1RealHints Enumeration Values	135
Table 66: Date Methods	138
Table 67: Time Methods	139
Table 68: DateTime Methods	140
Table 69: Qos Methods	142
Table 70: QosTimeliness Values	143
Table 71: QoSRates Values	144
Table 72: State Methods	145
Table 73: StreamStates Values	146
Table 74: StreamStates Methods	146
Table 75: DataStates Values	147
Table 76: DataStates Methods	147
Table 77: StateCodes Values	147
Table 78: StateCodes Methods	149
Table 79: Array Structure Members	150
Table 80: ArrayEntry Methods	152
Table 81: Buffer Methods	156
Table 82: RmtesCacheBuffer Methods	158
Table 83: RmtesBuffer Methods	159
Table 84: RmtesDecoder Decode Functions	159
Table 85: Transport API Container Types	161
Table 86: FieldList Methods	165
Table 87: FieldListFlags Values	167
Table 88: FieldEntry Methods	168
Table 89: ElementList Methods	173
Table 90: ElementListFlags Values	175
Table 91: ElementEntry Methods	175
Table 92: Map Methods	180
Table 93: MapFlags Values	183
Table 94: MapEntry Methods	184
Table 95: MapEntryFlagsValues	186
Table 96: MapEntryActions Values	186
Table 97: Series Methods	190
Table 98: SeriesFlags Values	192
Table 99: SeriesEntry Methods	193
Table 100: Vector Methods	197
Table 101: VectorFlags Values	199
Table 102: VectorEntry Methods	200
Table 103: VectorEntryFlags Values	201
Table 104: VectorEntryActions Values	202
Table 105: FilterList Methods	206
Table 106: FilterListFlags Values	207

Table 107: FlIterEntry Methods.....	208
Table 108: FlIterEntryFlags Values	210
Table 109: FlIterEntryActions Values	210
Table 110: Non-RWF Type Encode Methods.....	215
Table 111: Set-Defined Primitive Types.....	219
Table 112: FlElSetDef Method	222
Table 113: FlElSetDefEntry Methods.....	223
Table 114: ElmentSetDef Methods	223
Table 115: ElmentSetDefEntry Methods	224
Table 116: Local FlElSetDefDb Methods	225
Table 117: Local ElmentSetDefDb Methods	226
Table 118: Local Set Definition Database Encode Methods.....	226
Table 119: Local Set Definition Database Decode Methods.....	227
Table 120: Msg Methods	235
Table 121: MsgClasses Values.....	238
Table 122: MsgClasses Methods	239
Table 123: msgKey Methods	239
Table 124: MsgKeyFlags Values	241
Table 125: RequestMsg Methods	244
Table 126: RequestMsgFlags Values.....	245
Table 127: RefreshMsg Methods.....	247
Table 128: RefreshMsgFlags Values.....	249
Table 129: UpdateMsg Methods	250
Table 130: UpdateMsgFlags Values.....	251
Table 131: StatusMsg Methods	252
Table 132: StatusMsgFlags Values.....	253
Table 133: ClosureMsg Methods	254
Table 134: ClosureMsgFlags Values	254
Table 135: GenericMsg Methods	255
Table 136: GenericMsgFlags Values.....	256
Table 137: PostMsg Methods	257
Table 138: PostMsgFlags Values	258
Table 139: PostUserRights Values	259
Table 140: PostUserRights Methods	259
Table 141: AckMsg Methods	260
Table 142: AckMsgFlags Values	260
Table 143: AckMsgNakCodes Values.....	261
Table 144: Msg Encode Methods.....	262
Table 145: Msg Decode Methods.....	267
Table 146: EncoderIterator Utility Methods	269
Table 147: DecoderIterator Utility Methods	270
Table 148: grouped Buffer Utility Methods	274
Table 149: singleOpen and AllowsSuspectData Effects	276
Table 150: RDM Viewtypes Values	280
Table 151: Item and Group State Decision Table	288

Chapter 1 Introduction

1.1 About this Manual

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

This guide documents the functionality and capabilities of the Transport API Java Edition. In addition to connecting to itself, the Transport API can also connect to and leverage many different Thomson Reuters and customer components. If you want the Transport API to interact with other components, consult that specific component's documentation to determine the best way to configure and interact with these other devices.

1.2 Audience

This manual provides information and examples that aid programmers using the Transport API Java 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 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 Java programming language in a networked environment.

1.3 Programming Language

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

1.4 Acronyms and Abbreviations

ACRONYM	MEANING
ADH	Advanced Data Hub
ADS	Advanced Distribution Server
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATS	Advanced Transformation System
DACS	Data Access Control System
DMM	Domain Message Model
EDF	Elektron Data Feeds
EED	Elektron Edge Device
EMA	Elektron Message API, referred to simply as the Message API

Table 1: Acronyms and Abbreviations

ACRONYM	MEANING
EOA	Elektron Object API, referred to simply as the Object API.
ETA	Elektron Transport API, referred to simply as the Transport API
EWA	Elektron Web API
GC	Garbage Collection
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
IDN	Integrated Data Network
JIT	Just-In-Time
NIP	Non-Interactive Provider
OMM	Open Message Model
QoS	Quality of Service
RDF Direct	Reuters Data Feed Direct
RDM	Reuters Domain Model
RFA	Robust Foundation API
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format
SOA	Service Oriented Architecture
SSL	Source Sink Library
TREP	Thomson Reuters Enterprise Platform
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 References

1. *Transport API Java Edition RDM Usage Guide*
2. *API Concepts Guide*
3. *ANSI Library Reference Manuals*
4. *Transport API DACS LOCK Library Reference Manuals*
5. *Transport API Java Edition Value Added Components Developers Guides*

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 apidocumentation@thomsonreuters.com.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to Thomson Reuters by clicking **Send File** in the **File** menu. Use the apidocumentation@thomsonreuters.com address.

1.7 Document Conventions

This document uses the following types of conventions:

- Typographic
- Diagrams

1.7.1 Typographic

- structures, methods, in-line code snippets, and types are shown in **orange**, **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 an orange background. For example:

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

1.7.2 Diagrams

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

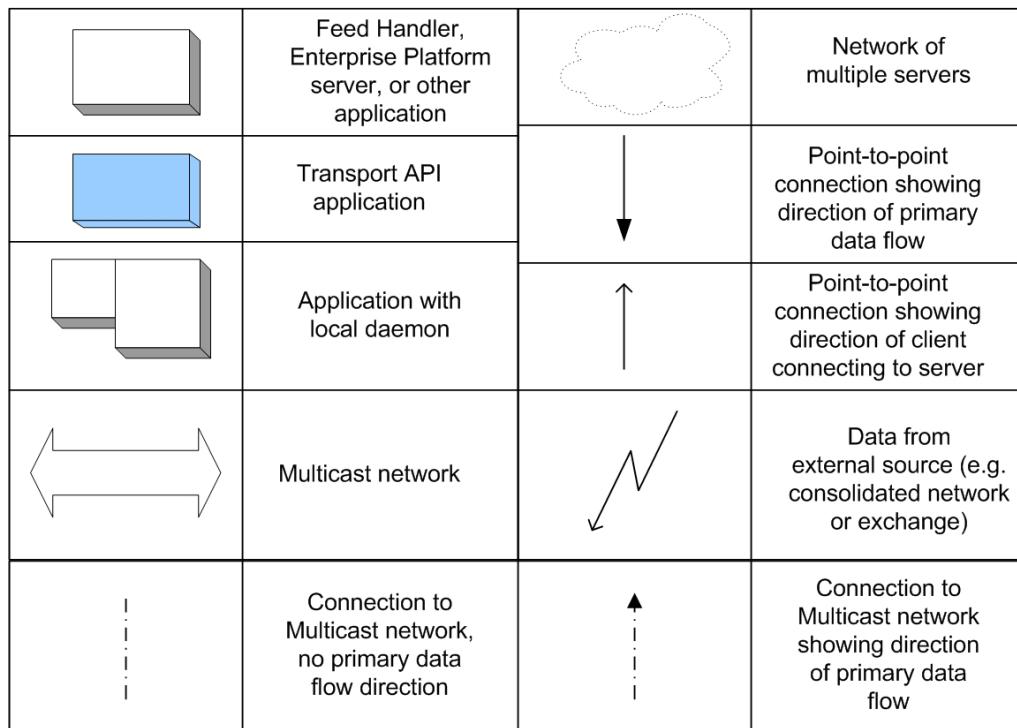


Figure 1. Network Diagram Notation

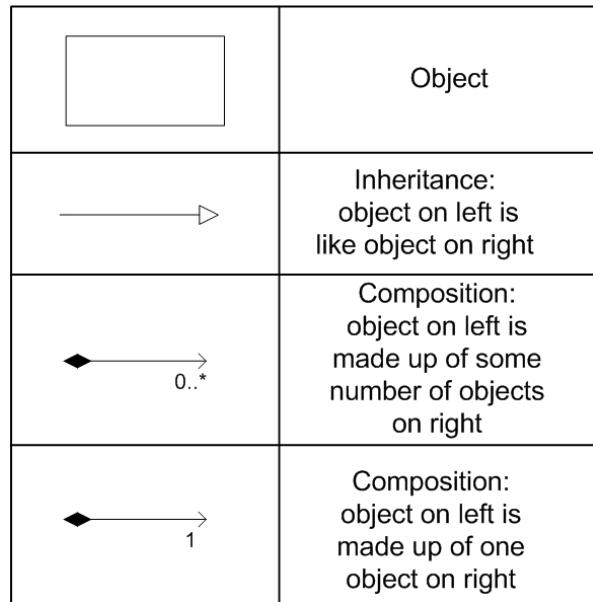


Figure 2. UML Diagram Notation

Chapter 2 Product Description

2.1 What is the Transport API?

The Transport API (also known as the RSSL API) is the customer release of Thomson Reuters's low-level internal API, currently used by the Thomson Reuters Enterprise Platform (TREP) and its dependent APIs for optimal distribution of OMM/RWF data. Due to its well-integrated and common usage across these products, the Transport API allows clients to write applications for use with Thomson Reuters Enterprise Platform (TREP) to achieve the highest performance, highest throughput, and lowest latency.

The Transport API is currently used by products such as the Advanced Distribution Server (ADS), Advanced Data Hub (ADH), Robust Foundation API (RFA), EDF-D, Elektron, and Eikon.

The Transport API supports all constructs available as part of the Open Message Model. It complements RFA and the Message API by allowing users to choose the type of functionality and layer (Session or Transport) at which they want to access the TREP. With the addition of the Transport API, customers have a choice between a feature-loaded session-level API (i.e., the Message API) and high-performance transport-level API (i.e., the Transport API).

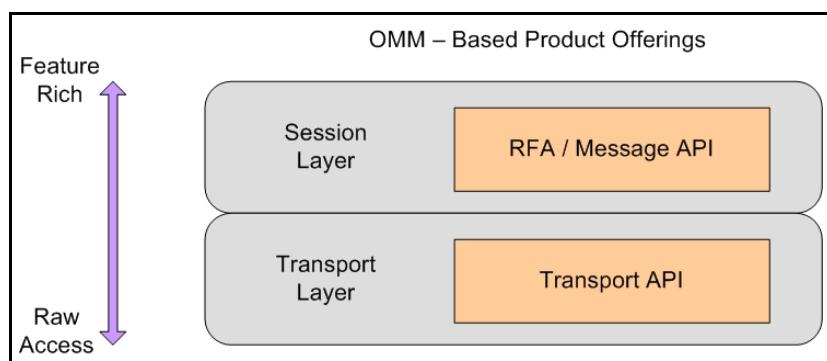


Figure 3. OMM-Based Product Offerings

The Transport API is a low-level API that provides application developers with the most flexible development environment and is the foundation on which all Thomson Reuters OMM-based components are built. By utilizing an API at the transport level, a client can write to the same API as the ADS / ADH and achieve the same levels of performance.

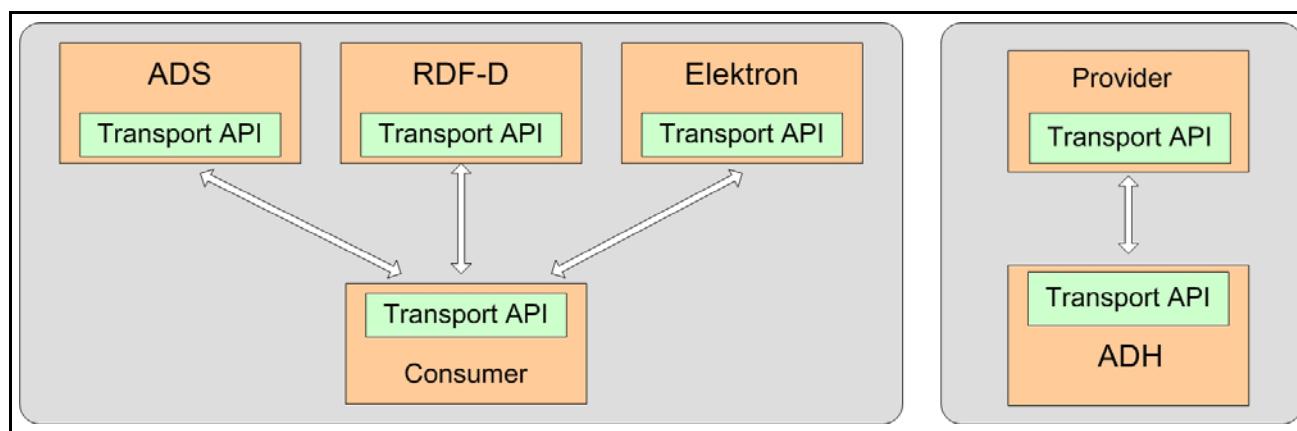


Figure 4. Transport API: Core Diagram

2.2 Transport API Features

The Transport API is:

- Depending on the particular API, available in C and Java.
- 64-bit.
- Thread-safe and thread-aware.
- Capable of handling:
 - Any and all OMM primitives and containers.
 - All Domain Models, including those defined by Thomson Reuters as well as other user-defined models.
- A reliable, transport-level API which includes OMM encoders/decoders.

Additionally, certain the Transport APIs provides an ANSI Page parser to encode/decode ANSI sequences and a DACS Library to allow generation of DACS Locks.

2.2.1 General Capabilities

The Transport API provides general capabilities independent of the type of application. The 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 Transport API to create consumer-based applications that can:

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

2.2.3 Provider Applications: Interactive

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

- Receive requests and respond to streaming and snapshot-based Requests from ADH (previously known as Managed or Sink-Driven Server applications).
- Receive and respond to batch, views, and symbol list requests from ADH.
- Receive and respond to requests for a Private Stream from the ADH.
- Receive requests for pause and resume on active data streams.
- Receive and acknowledge post messages (used receiving consumer- based Publishing and Contributions) from ADH.
- Send and receive Generic Messages with ADH.

Additionally, you can use the Transport API to create server-based applications that can accept multiple connections from ADH, or allows multiple ADHs to connect to a provider.

2.2.4 Provider Applications: Non-Interactive

Using the Transport API, you can write non-interactive applications that start up and begin publishing data to ADH (previously known as Source-Driven (Src-Driven) or broadcast-style server applications). This includes both TCP and UDP multicast-based Non-Interactive Provider (NIP) applications.

2.3 Performance and Feature Comparison

Though TREP'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., RFA) when developing high-performance applications. By writing to the Transport API, a client can leverage the full throughput and low latency of the core infrastructure while bypassing the full set of RFA'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 Elektron Transport API. A Transport API-based application's maximum achievable performance (latency, throughput, etc) is determined by the infrastructure component to which it connects. Thus, to know performance metrics, you should look at the performance numbers for the associated infrastructure component. For example:

- If a Transport API consumer application talks to the ADS and you want to know the maximum throughput and latency of the consumer, look at the performance numbers for the ADS configuration you use.
- If a Transport API provider application talks to an ADH and you want to know the maximum throughput and latency of the Transport API provider, look at the performance numbers for the ADH Configuration you use.

 **Tip:** The 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 TREP infrastructure documentation, look for Transport API or RSSL numbers (TREP documentation often refers to the Transport API as RSSL), which will give the performance and latency of the 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 (watch list ^a)
Reactor ^b	Safe and Aware	Very High	Low	Medium (watch list optional)
RFA	Safe and Aware	High	Low	Medium (watch list, allows optional queues)
SFC C++	None	Medium	High	Medium – High (watch list, cache)
SSL 4.5	Big Lock	Medium - High	Medium	Low (watch list optional)
SSL 4.0 Classic Edition	Big Lock	Low - Medium	Medium - High	Medium (watch list)

Table 2: API Performance Comparison

a. The Elektron Message API leverages the reactor watchlist.

b. The Reactor is an ease-of-use layer provided with the Elektron Transport API.

2.3.1 Java Garbage

Within its own implementation, the Transport API C Edition minimizes garbage collection. Additionally, the interface allows user applications to limit their own garbage collection, if desired.

If the performance overhead of garbage collection is a concern, you have several options in reducing its impact on your application, such as:

- Objects obtained through Transport API C Edition's factories are owned by the application, with some exceptions (e.g.; TransportBuffers, Channels, Servers). An application can limit garbage collection by reusing these objects (e.g., pooling).
- Maintain long term references to objects in the application until such a time as garbage collection is tolerable (i.e., after trading hours).
- Avoid the use of collections that have internal garbage collection. When using collections, ensure they are sized to allow for growth without implicit resizing, and attempt reuse where possible. Alternately, write your own variant or leverage a third-party package.
- Use a profiler to detect hot spots in your application, which you can then optimize to meet your requirements.
- Java Strings are immutable, resulting in contents being garbage collected whenever an attempt is made to modify or redefine the string's contents. Consider the use of an alternative type or use StringBuilder to avoid some garbage collection. `Object.toString` conversion methods will generally create a new string whenever it is invoked, which can also add to garbage collection overhead.

This manual and the Transport API C Edition Reference Manual both denote any method that internally incurs garbage collection. However, non-Transport API libraries (i.e., Apache, DACS, ANSIPage, XPP, etc) might also collect garbage.

2.3.2 Use of Assertions

In some cases the Transport API C Edition library uses assertions, rather than IF statements, to verify the integrity of expected values. To aid in troubleshooting during development, we recommend that you enable Java assertions (JVM arg: `-enableassertions`). When running in production, do not enable Java assertions as this adversely affects performance.

2.4 Functionality: Which API to Choose?

To make an informed decision on which API to use, you should balance the tradeoffs between performance and functionality (for performance comparisons, refer to Section 2.3).

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

The following table lists API capabilities using the following legend:

- X: Supported in current version, natively implemented
- X**: Supported in current version, leverages lower-level capability
- Future: Planned for a future release
- Legacy: A legacy functionality

CAPABILITY TYPE	CAPABILITY	TRANSPORT API 3.0	REACTOR ^A	MESSAGE API 3.0 ^B	OBJECT API 3.0	ELEKTRON WEB API 1.7	RFA 8.0
Transport	Compression via OMM	X	X**	X**	Future		X
	HTTP Tunneling (RWF)	X	X**	X**	Future		X
	TCP/IP: RWF	X	X**	X**	Future		X
	Reliable Multicast: RWF	X	X**	X**	Future		X
	Sequenced Multicast	X					
	WebSocket					X	
	Unidirectional Shared Memory	X					
Application Type	Consumer	X	X	X	Future	X	X
	Provider: Interactive	X	X	Future	Future		X
	Provider: Non-Interactive	X	X	Future	Future		X
General	Batch Request	X	X	X**	Future		X
	Batch Re-issue and Close	X	X				X
	Generic Messages	X	X	X	Future		X
	Pause/Resume	X	X	X	Future		X

Table 3: Capabilities by API

Capability Type	Capability	Transport API 3.0	Reactor ^A	Message API 3.0 ^B	Object API 3.0	Elektron Web API 1.7	RFA 8.0
General <i>(Continued)</i>	Posting	X	X	X	Future		X
	Snapshot Requests	X	X	X	Future	X	X
	Streaming Requests	X	X	X	Future	X	X
	Private Streams	X	X	X	Future	Future	X
	Qualified Streams	X	X	X	Future	Future	
	Views	X	X	X	Future		X
Domain Models	Custom Data Model Support	X	X	X	Future		X
	RDM: Dictionary	X	X	X	Future		X
	RDM: Enhanced Symbol List	X	X				X
	RDM: Login	X	X	X	Future		X
	RDM: Market Price	X	X	X	Future	X	X
	RDM: MarketByOrder	X	X	X	Future	Future	X
	RDM: MarketByPrice	X	X	X	Future	Future	X
	RDM: Market Maker	X	X	X	Future	Future	X
	RDM: Queue Messaging		X	Future	Future	Future	
	RDM: Service Directory	X	X	X	Future	Future	X
	RDM: Symbol List	X	X	X	Future	Future	X
	RDM: Yield Curve	X	X	X	Future	Future	X
Encoders/Decoders	AnsiPage	X	X**	X**			Legacy
	DACS Lock	X	X**	X**	Future		X
	OMM	X	X	X**	Future	Future	X
	RMTEs	X	X	X**	Future	Future	X

Table 3: Capabilities by API

Chapter 2 Product Description

Capability Type	Capability	Transport API 3.0	Reactor ^A	Message API 3.0 ^B	Object API 3.0	ELEKTRON Web API 1.7	RFA 8.0
Layer Specific	Config: file-based			X	Future		X
	Config: programmatic	X	X	X	Future		X
	Group fanout to items		X	X**	Future		X
	Load balancing: API-based		X	X**	Future		X
	Logging: file-based			X	Future		X
	Logging: programmatic	X	X	Future	Future		X
	QoS Matching		X	X**	Future		X
	Network Pings: automatic		X	X**	Future		X
	Recovery: connection		X	X**	Future		X
	Recovery: items		X	X**	Future		X
	Request routing		X	X**	Future		X
	Session management		X	X	Future		X
	Service Groups						X
	Single Open: API-based		X	X**	Future		X
	Warm Standby: API-based						X
	Watchlist		X	X**	Future		X
	Controlled fragmentation and assembly of large messages	X	X**				
	Controlled locking / threading model	X					
	Controlled dynamic message buffers with ability to programmatically modify during runtime	X	X**				
	Controlled message packing	X	X**				
	Messages can be written at different priority levels	X	X**				

Table 3: Capabilities by API

CAPABILITY TYPE	CAPABILITY	TRANSPORT API 3.0	REACTOR ^A	MESSAGE API 3.0 ^B	OBJECT API 3.0	ELEKTRON WEB API 1.7	RFA 8.0
Other	API Class Generation				Future	Future	
	API based Content Caching				Future		
	Content Aware Objects				Future	Future	

Table 3: Capabilities by API

a. The Reactor is an open source component that functions within the ETA.

b. Not yet released. This version number is tentative and subject to change.

Chapter 3 Consumers and Providers

3.1 Overview

For those familiar with previous API products or concepts from TREP, Rendezvous, or Triarch, we map how the Transport API implements the same functionality.

At a very high level, the TREP system facilitates controlled and managed interactions between many different service **providers** and **consumers**. Thus, TREP 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.).

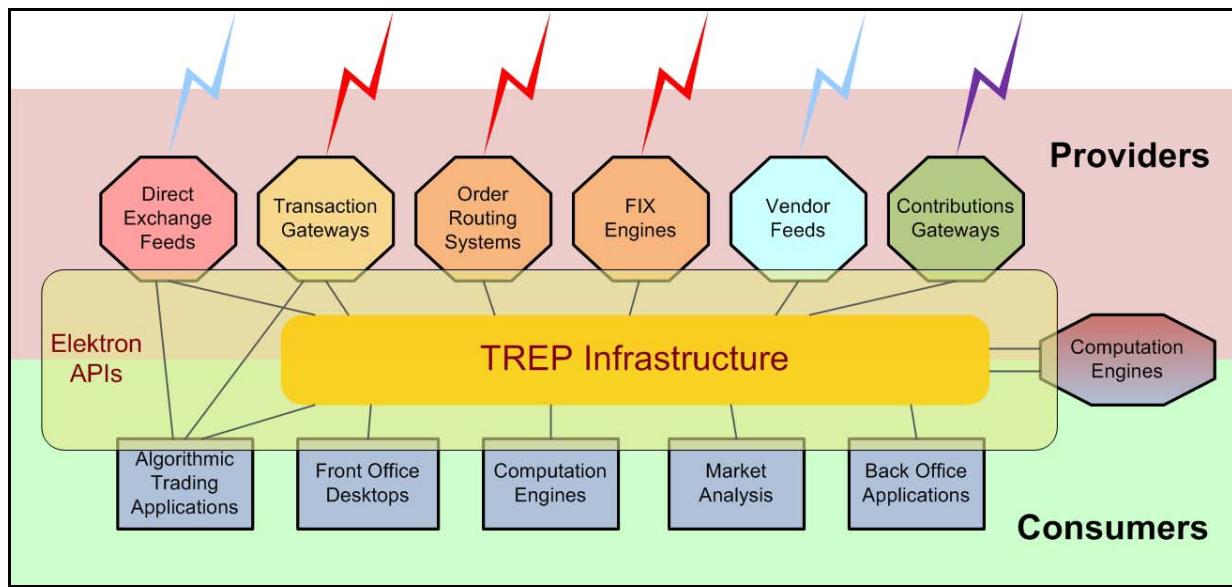


Figure 5. TREP Infrastructure

To access needed capabilities, consumers always interact with a provider, either directly and/or via TREP. Consumer applications that want the lowest possible latency can communicate directly via TREP 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 TREP infrastructure between provider and consumer applications.

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 Figure 6 illustrates a Transport API consumer connecting to Elektron via a direct access point.
- A **proxy access point**. A proxy access point is point-to-point based or multicast (according to your needs) and implemented by a TREP Infrastructure component (i.e., an ADS). Figure 6 also illustrates a Transport API consumer connecting to the provider by first passing through a proxy access point.

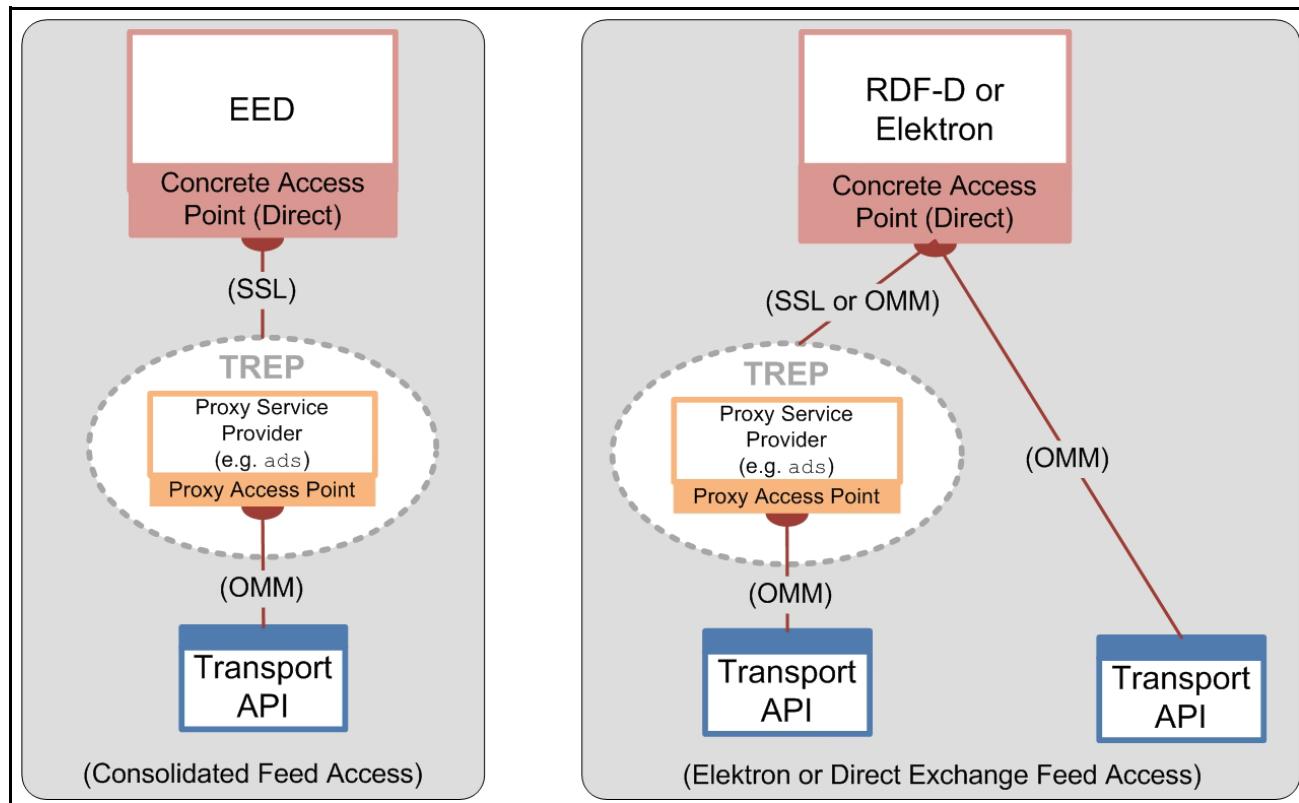


Figure 6. Transport API as a Consumer

Examples of consumers include:

- An application that subscribes to data via TREP, EDF, or Elektron.
- An application that posts data to TREP or Elektron (e.g., contributions/inserts or local publication into a cache).
- An application that communicates via generic messages with TREP or Elektron.
- 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., ADS or Elektron) 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 ADS, the ADS 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 ADS responses.

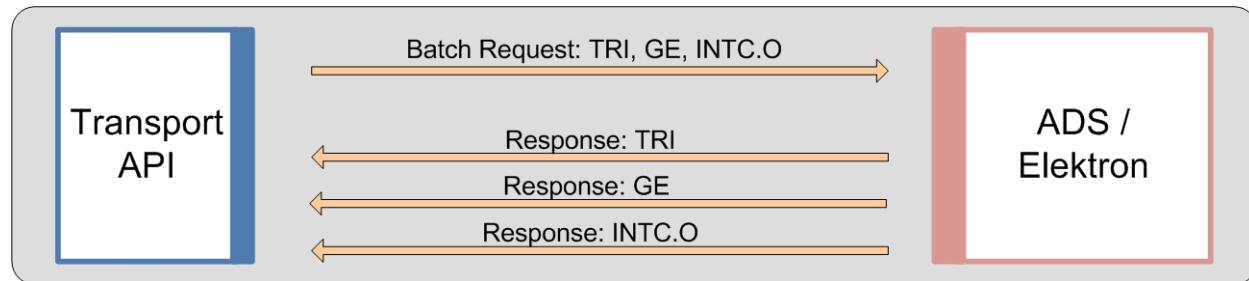


Figure 7. Batch Request

3.2.3 Views

The system reduces the amount of data flowing across the network by means of **Filtering**. To improve performance and maximize bandwidth, you can configure the TREP 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 ADS/Elektron) 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.

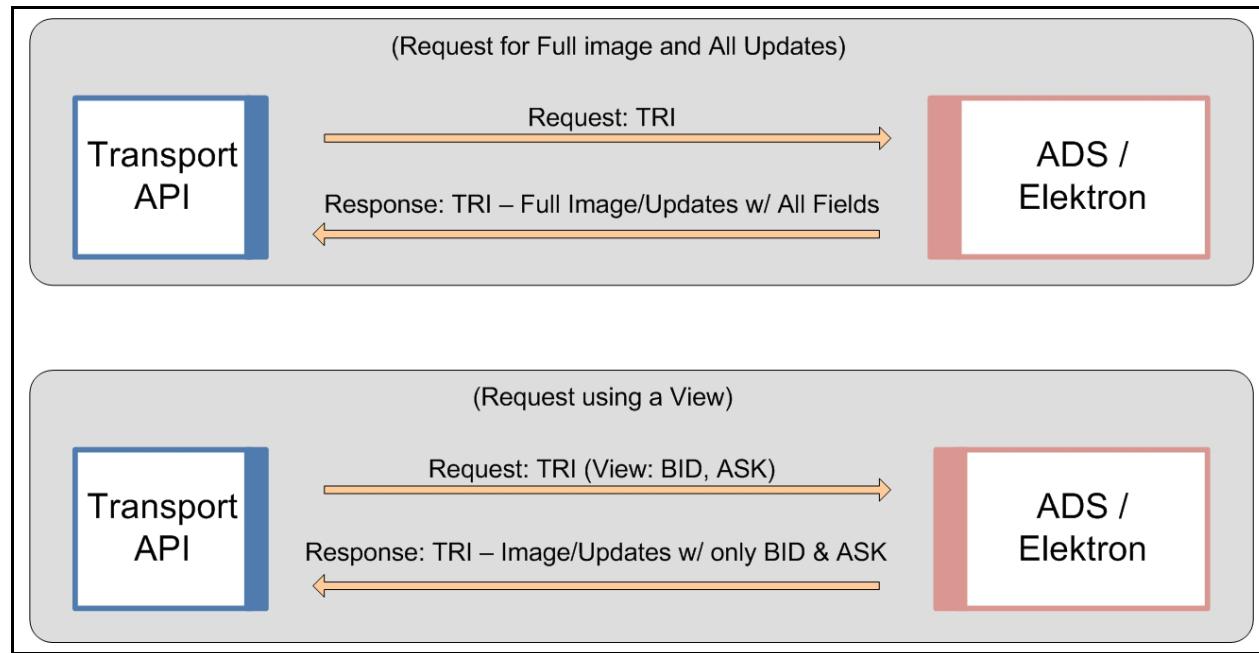


Figure 8. View Request Diagram

Views were designed to provide the same filtering functionality as the Legacy STIC device and SFC (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 ADS. The ADS receives the pause request and stops sending new data to the client for that item, though the item remains open and in the ADS Cache. The ADS 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 ADS, which then responds by sending an aggregated update or a refresh (a current image) to the client. After the ADS resumes sending data, the ADS sends all subsequent messages.

By using the Pause/Resume feature a client can avoid issuing multiple open/close requests which can disrupt the ADS 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 ADS and risk having ADS abruptly cut its connection, nor does the client want to close its own connection (or close all the items on the ADS) 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 ADS 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 ADS 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 ADS 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 SSL 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).

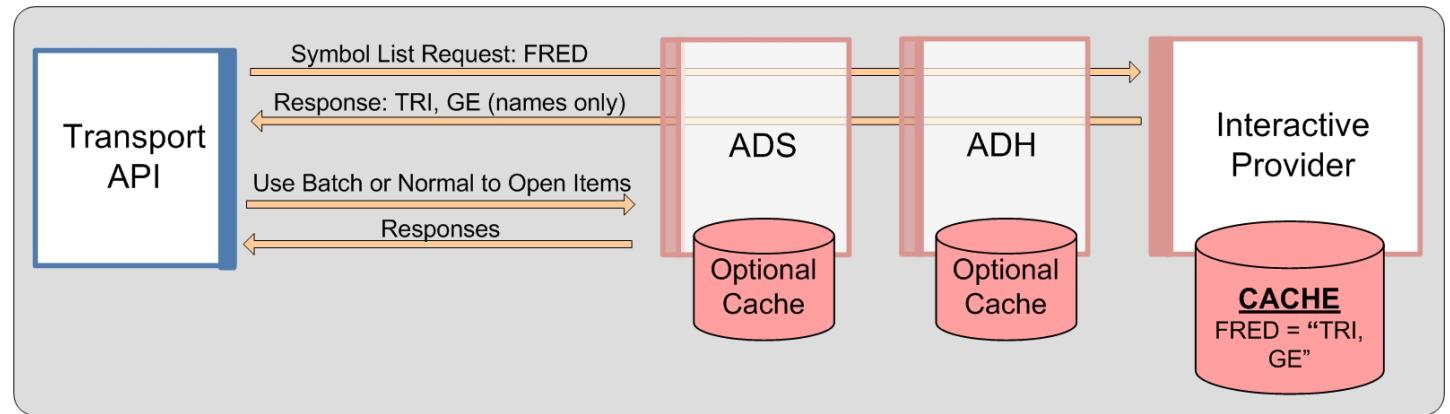


Figure 9. Symbol List: Basic Scenario

The following diagram illustrates how a consumer can access all items in the ADS Cache, effectively dumping the cache to the OMM client. In this scenario, the client requests the symbol list **_ADS_CACHE_LIST**. The ADS 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 ADS provides an additional symbol list (**_SERVER_LIST**) for obtaining lists of items stored in specific ADH instances. For details on this symbol list, refer to the *ADS and ADH Software Installation Manuals*.

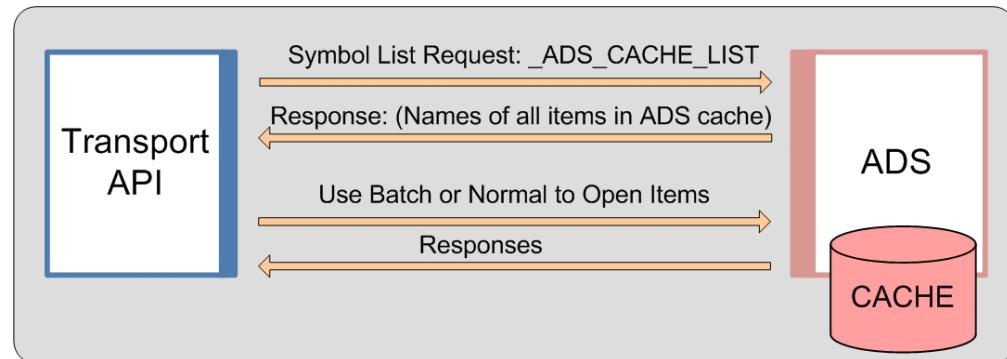


Figure 10. Symbol List: Accessing the Entire ADS Cache

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 ADS or in direct connections with interactive providers. For details on the model for requesting symbol list data streams, see the *Transport API RDM 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 (QoS) advertised by the service in the provider's source directory response.

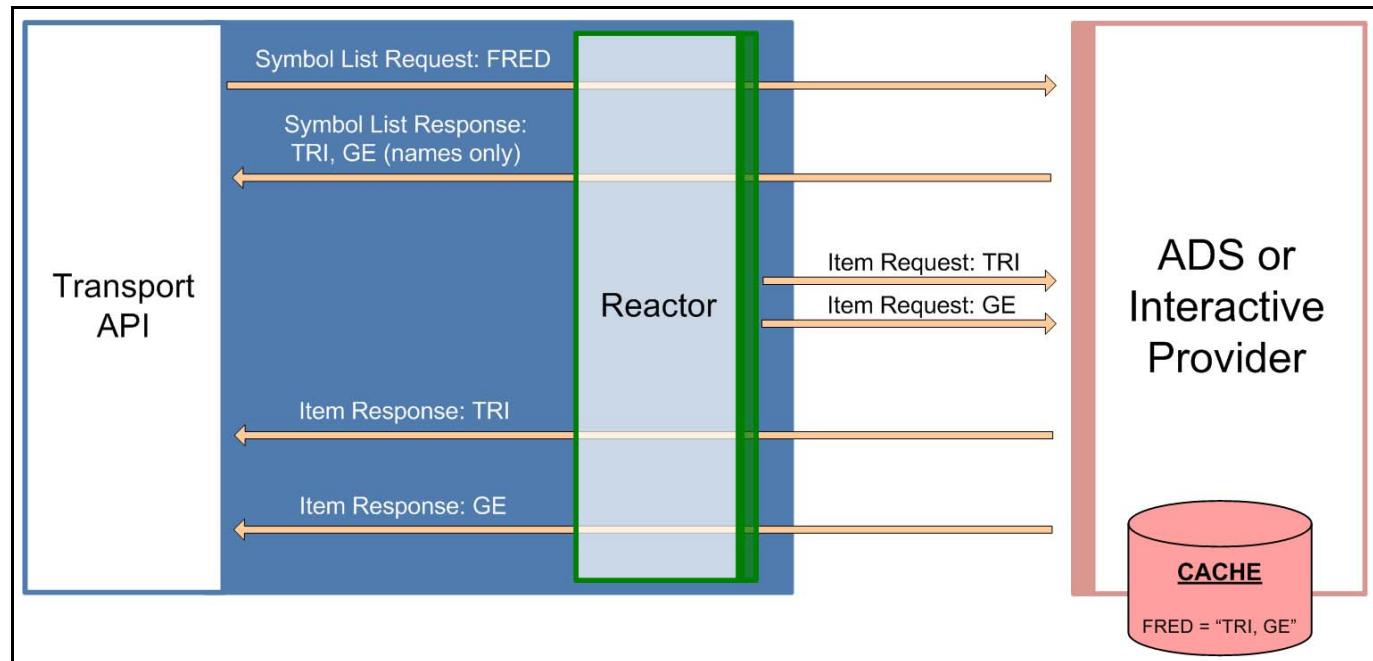


Figure 11. Symbol List: Requesting Symbol List Streams via the Transport API Reactor

3.2.5.2 Server Symbol Lists

Using certain Elektron APIs, client applications can request a list of all symbols maintained in the cache of all ADH 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 ADH 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 ADH cache for both interactive and non-interactive services and is supported for both RSSL (symbol list) and SSL 4.5 (criteria) clients.

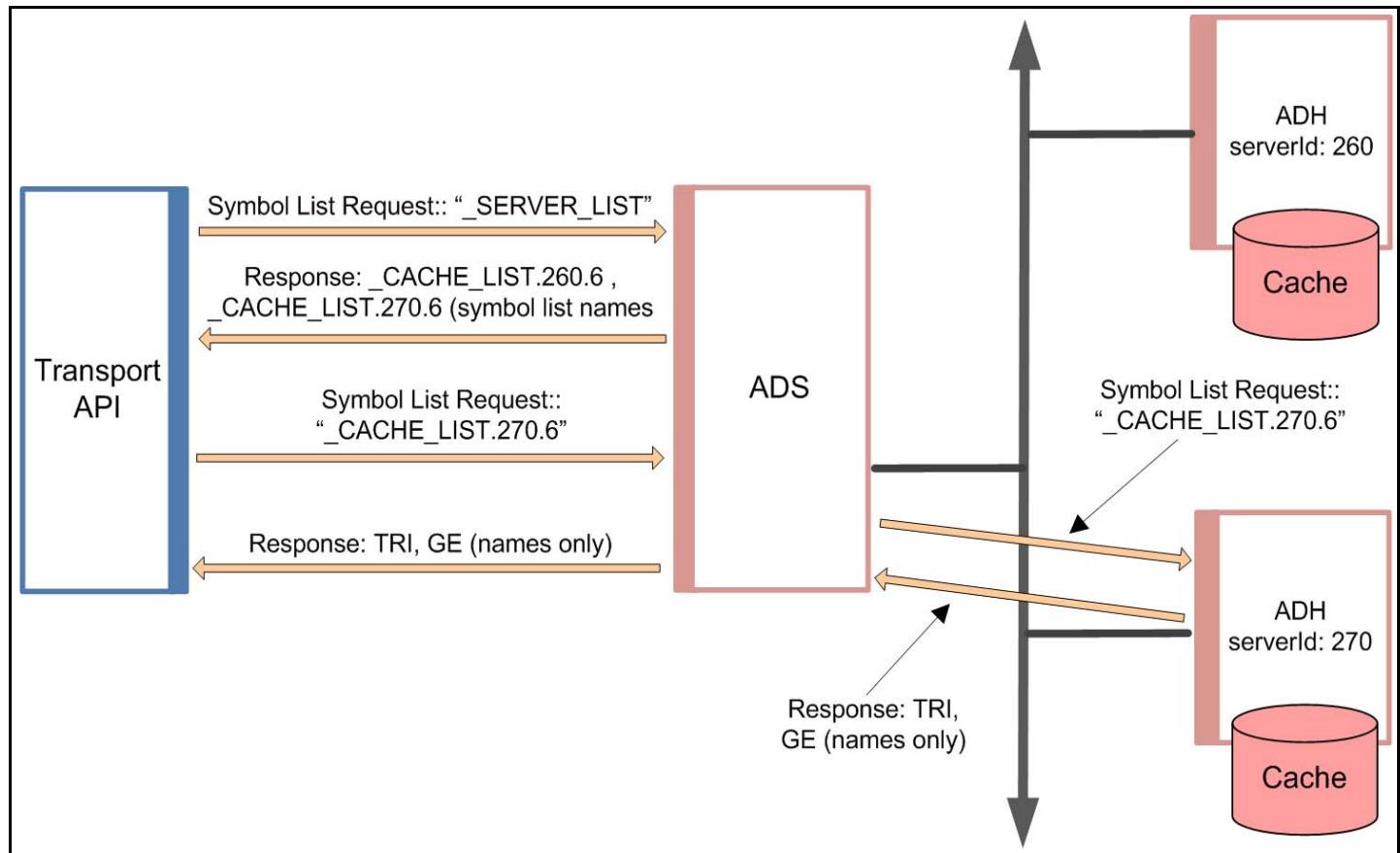


Figure 12. Server Symbol List

3.2.6 Posting

Through posting, API consumers can easily push content into any cache within the TREP (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 TREP (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., ADS, ADH, 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 ADH. In both cases the ADH receives posted messages and then distributes these messages to interested consumers. In the right-side segment, the ADS 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 ADH cache. The Transport API can even post to provider applications (i.e., the Purple service in this diagram) that support posting.

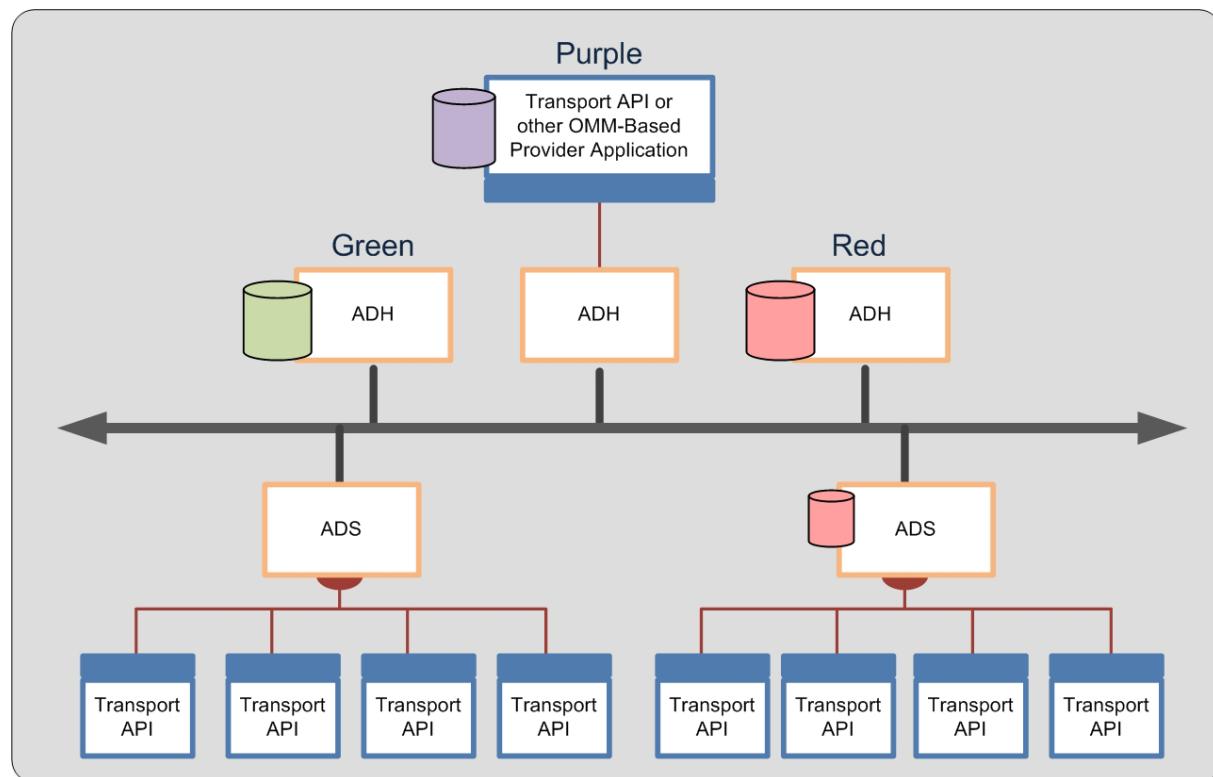


Figure 13. Posting into a Cache

You can use the Transport API to post into an ADH cache. If a cache exists in the ADS (the Red service), the ADS cache is also populated by responses from the ADH cache. If you configure TREP to allow such behavior, posts can be sent beyond

the ADH (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 ADS 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 OMM-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 Transport API sends an OMM post to a provider application that supports such functionality.

While this diagram is similar to the example in Figure 13, the difference is that core components (such as the ADS/ADH) in TREP can convert a post into an SSL Insert for legacy connectivity. This functionality is provided for migration purposes.

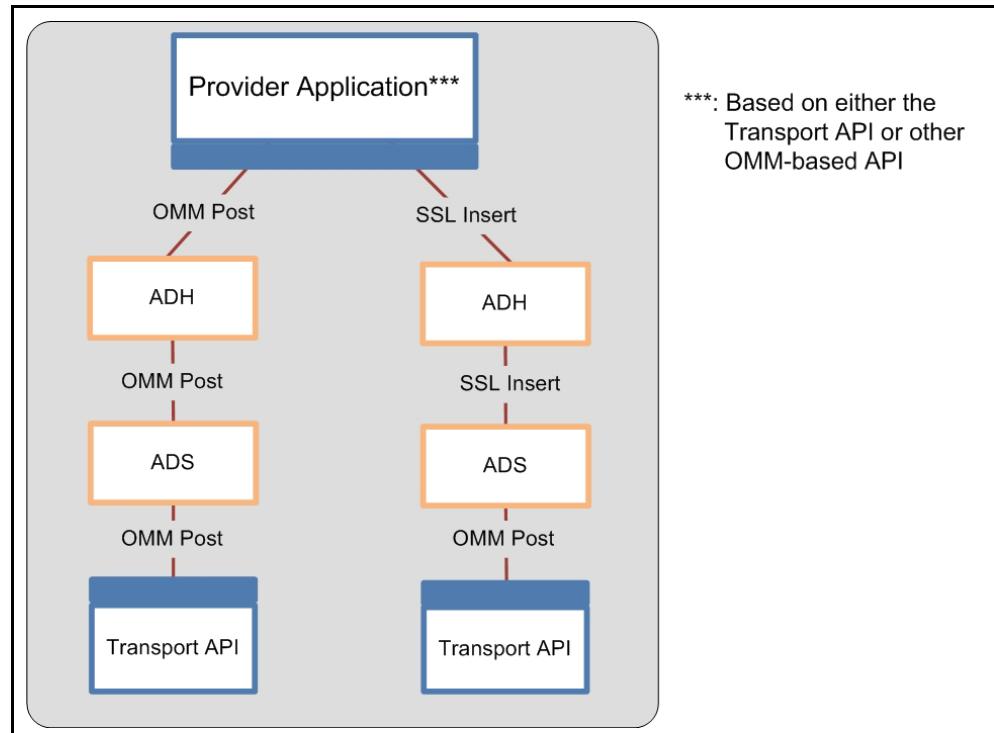


Figure 14. OMM Post with Legacy Inserts

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 OMM primitive type. Whereas the request/response type message flows from TREP 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 ADS, while the ADS 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 TREP, or via a cascaded set of platforms. The following diagram illustrates these different configurations.

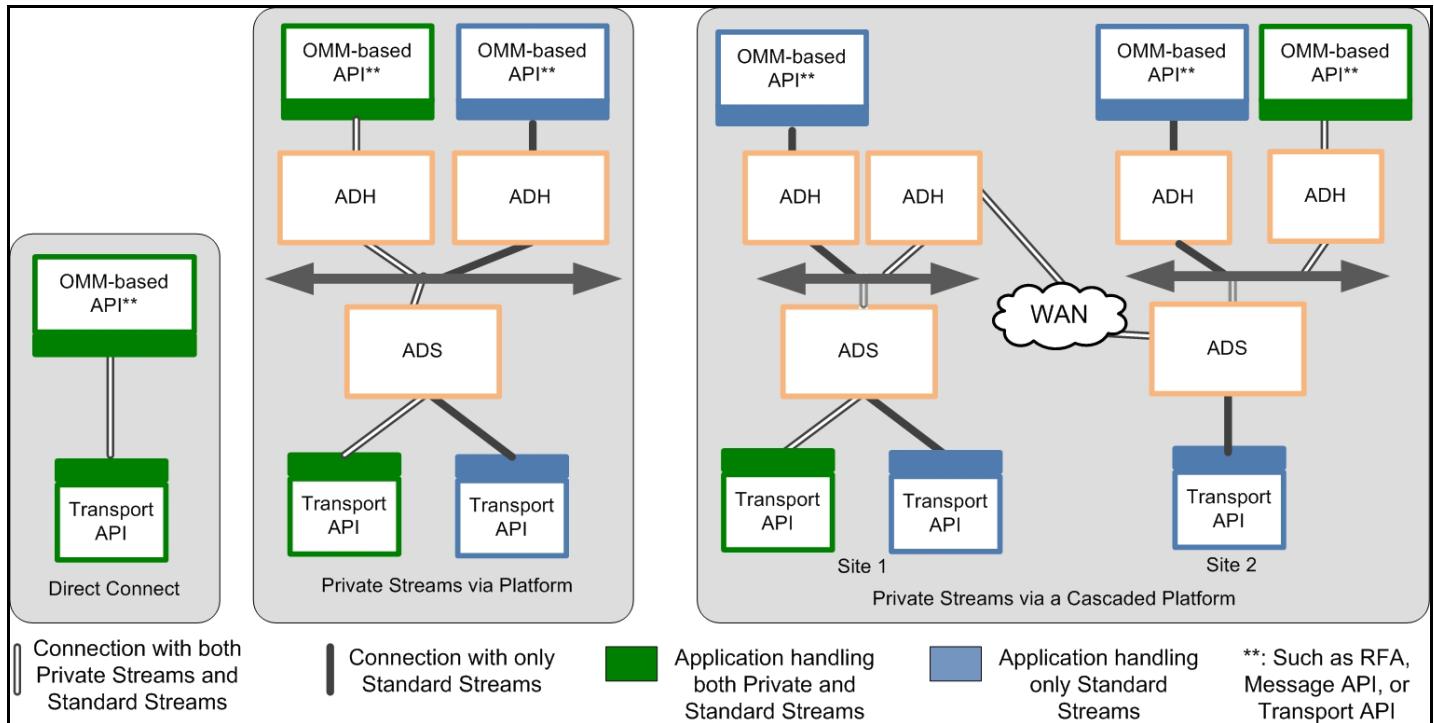


Figure 15. Private Stream Scenarios

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 Transport API or TREP 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 Thomson Reuters Domain Models & Custom Domain Models

3.3 Providers

Providers make their services available to consumers through TREP 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 TREP infrastructure component (like the ADH).

Examples of providers include:

- A user who receives a subscription request from TREP.
- A user who publishes data into TREP, whether in response to a request or using a broadcast-publishing style.
- A user who receives post data from TREP. 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 TREP.

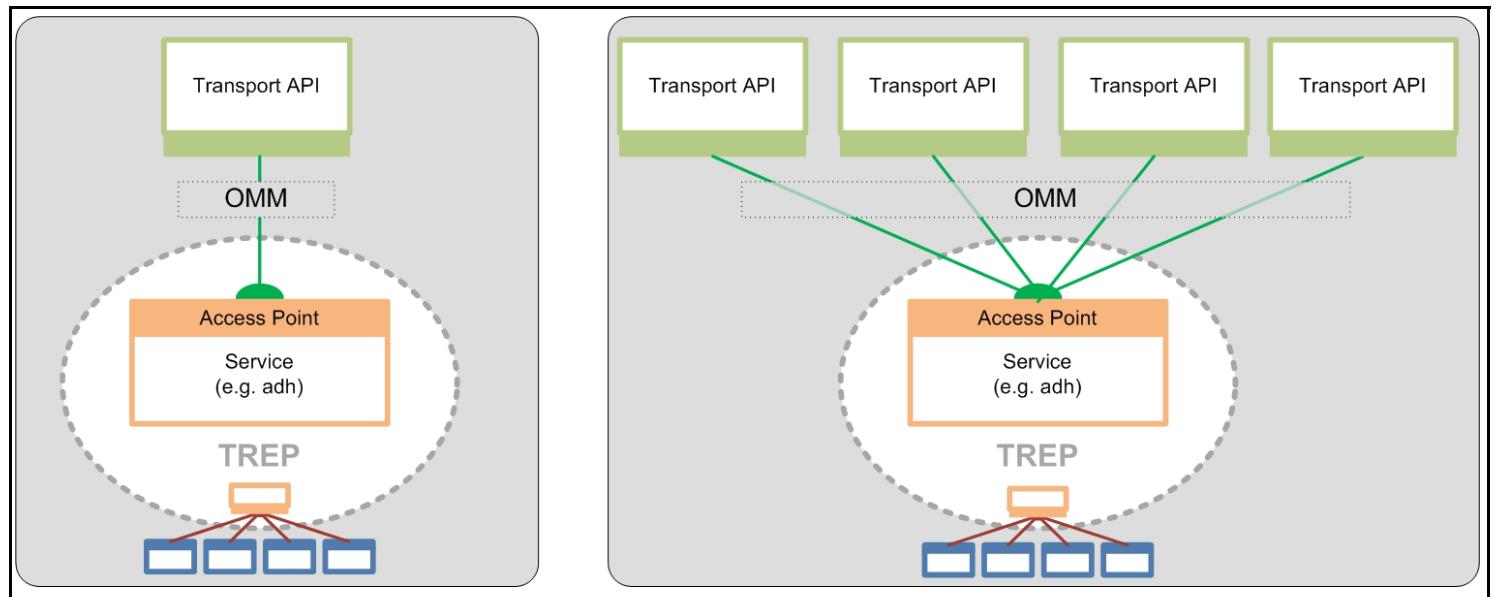


Figure 16. Provider Access Point

3.3.1 Interactive Providers

An **interactive provider** is one that communicates with the TREP, accepting and managing multiple connections with TREP components. The following diagram illustrates this concept.

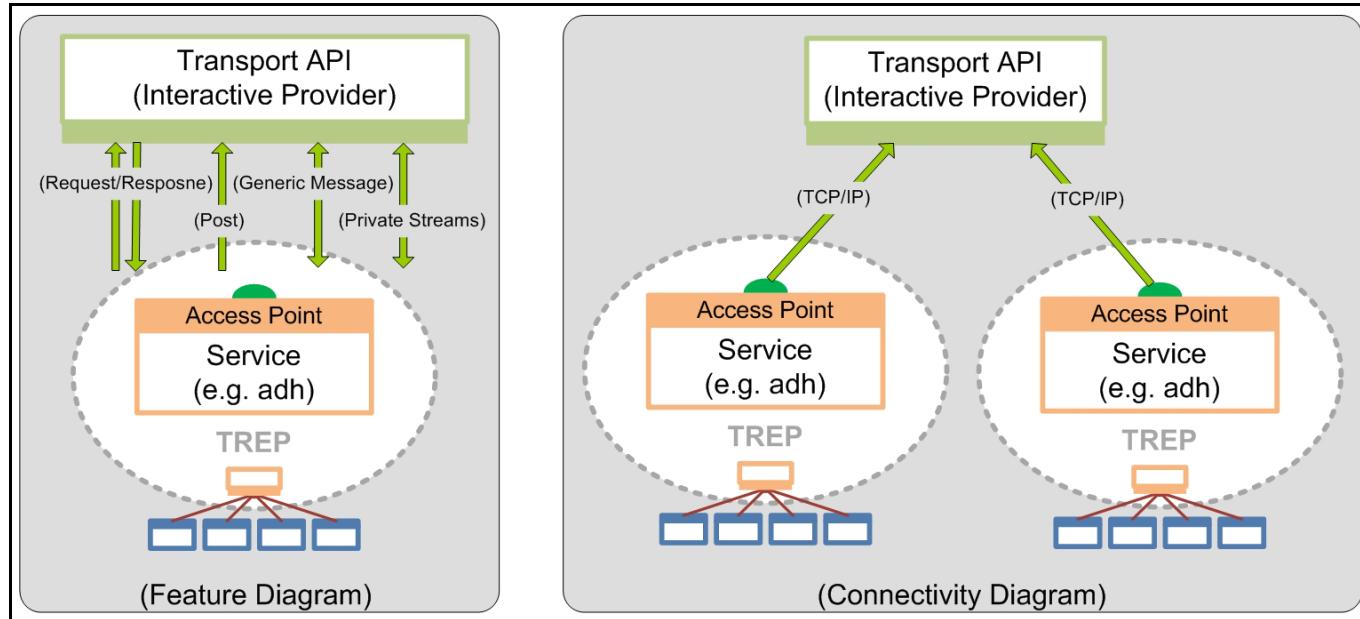


Figure 17. Interactive Providers

An interactive provider receives connection requests from the TREP. 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 legacy Triarch users or early TREP 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 Transport API interactive provider can accept and manage connections from multiple TREP components.

3.3.1.1 Request /Response

In a standard request/response scenario, the interactive provider receives requests from consumers on TREP (e.g., “Provide data for item TRI”). 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 TREP. Post messages will state whether an acknowledgment is required. If required, TREP 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 TREP 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 TREP 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 OMM data type.

Interactive providers can receive a generic message from and publish a generic message to TREP.

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.

3.3.2 Non-Interactive Providers

A ***non-interactive provider*** (NIP) writes a provider application that connects to TREP and sends a specific set of non-interactive data (services, domains, and capabilities).

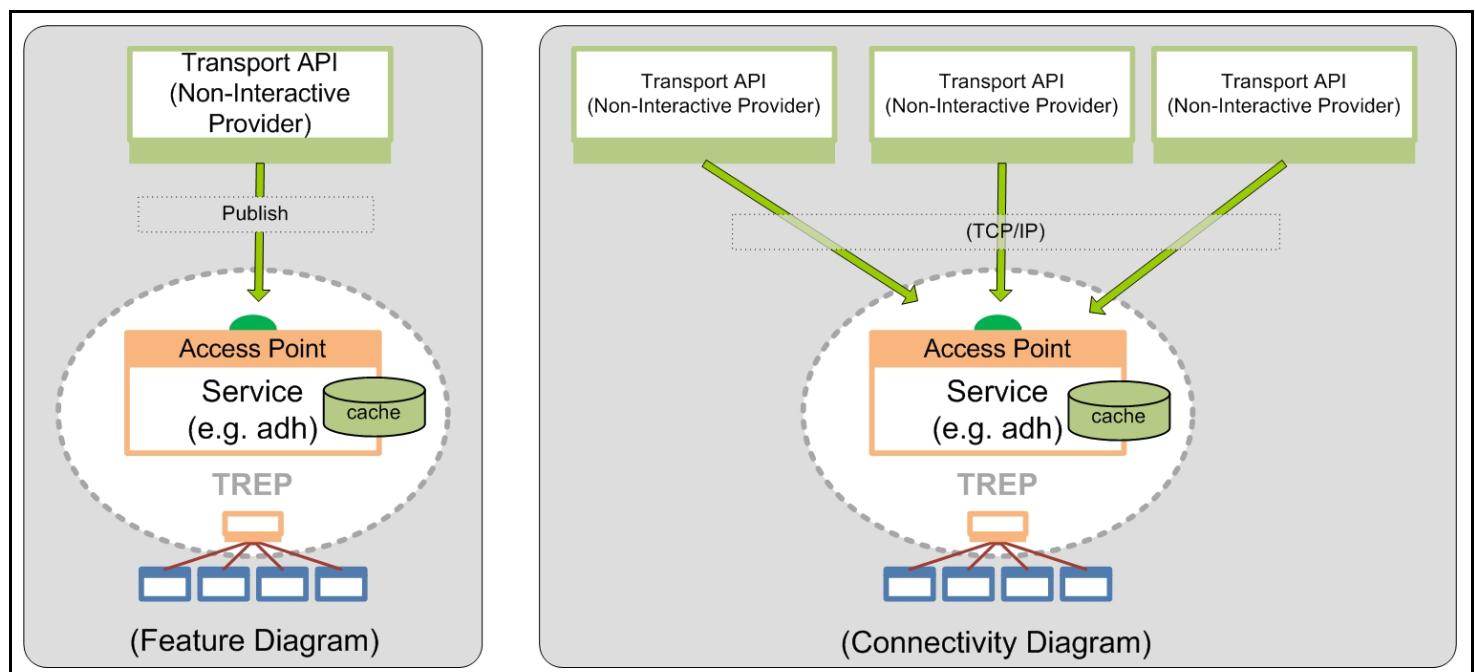


Figure 18. NIP: Point-To-Point

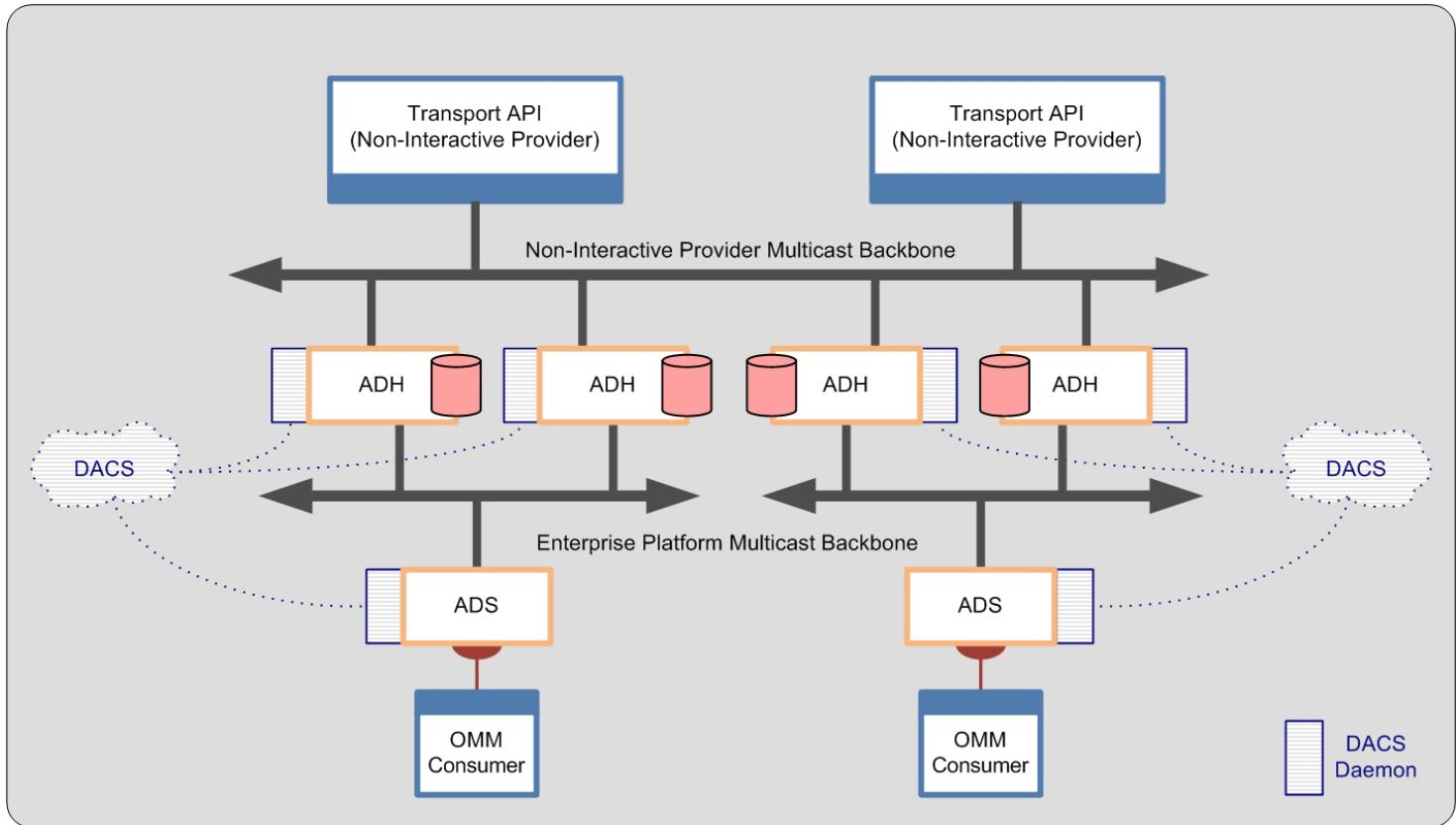


Figure 19. NIP: Multicast

After a NIP connects to TREP, the NIP can start sending information for any supported item and domain. For legacy Triarch users or early TREP adopters, the NIP 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 NIPs can connect to the same TREP and publish the same items and content. For example, two NIPs can publish the same or different fields for the same item "INTC.O" to the same TREP.

NIP 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.

Chapter 4 System View

4.1 System Architecture Overview

A TREP network typically hosts the following:

- Core Infrastructure (i.e., ADS, 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.
 - NIP applications which publish data, regardless of user requests or which applications consume the data.
- Permissioning infrastructure (i.e., DACS)
- Devices which interact with the markets (i.e., Data Feed Direct and the Elektron Edge Device)

The following figure illustrates a typical deployment of a TREP network and some of its possible components. Components that use the 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 Transport API integrates with each.

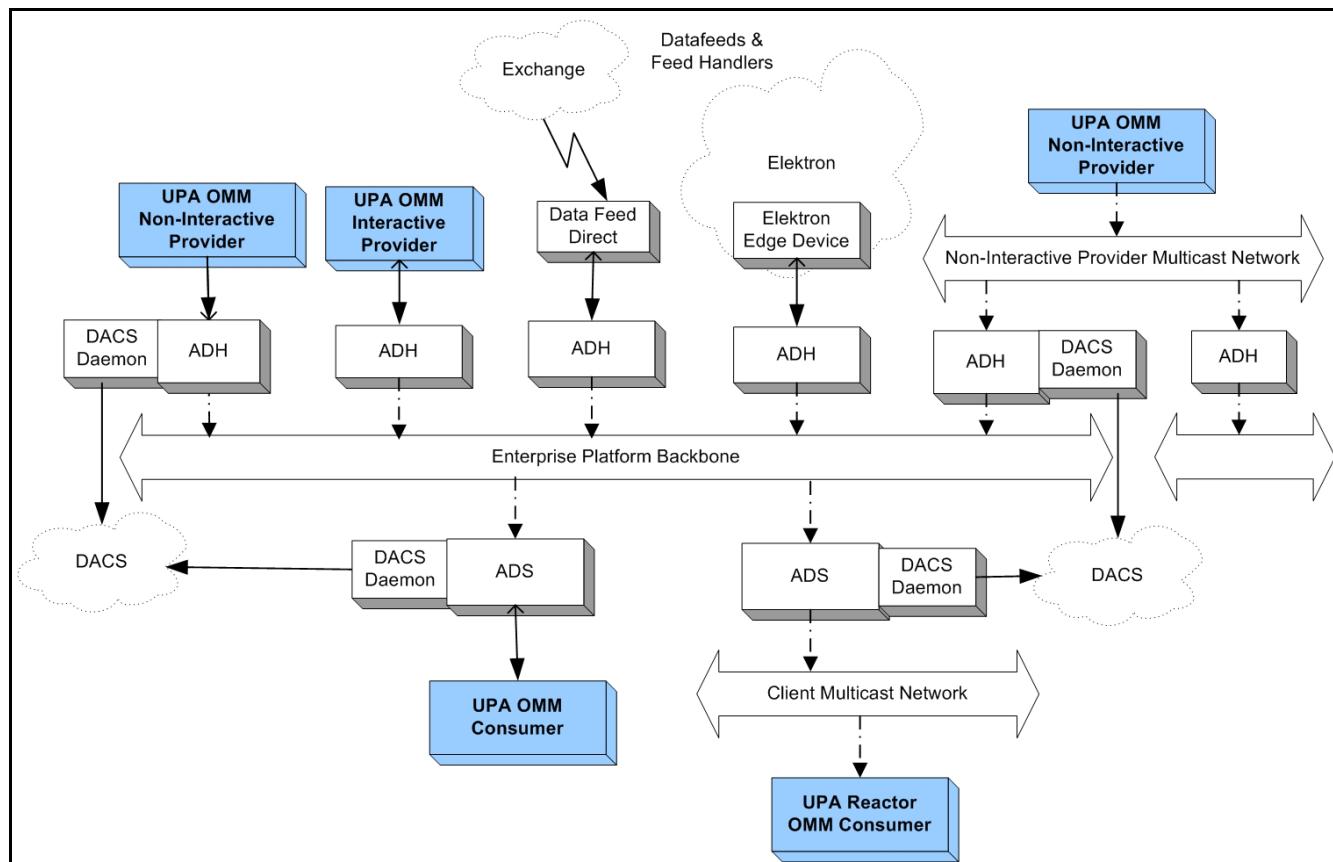


Figure 20. Typical TREP Components

4.2 Advanced Distribution Server (ADS)

The ADS provides a consolidated distribution solution for Thomson Reuters, value-added, and third-party data for trading-room systems. It distributes information using the same OMM and RWF protocols exposed by the Transport API.

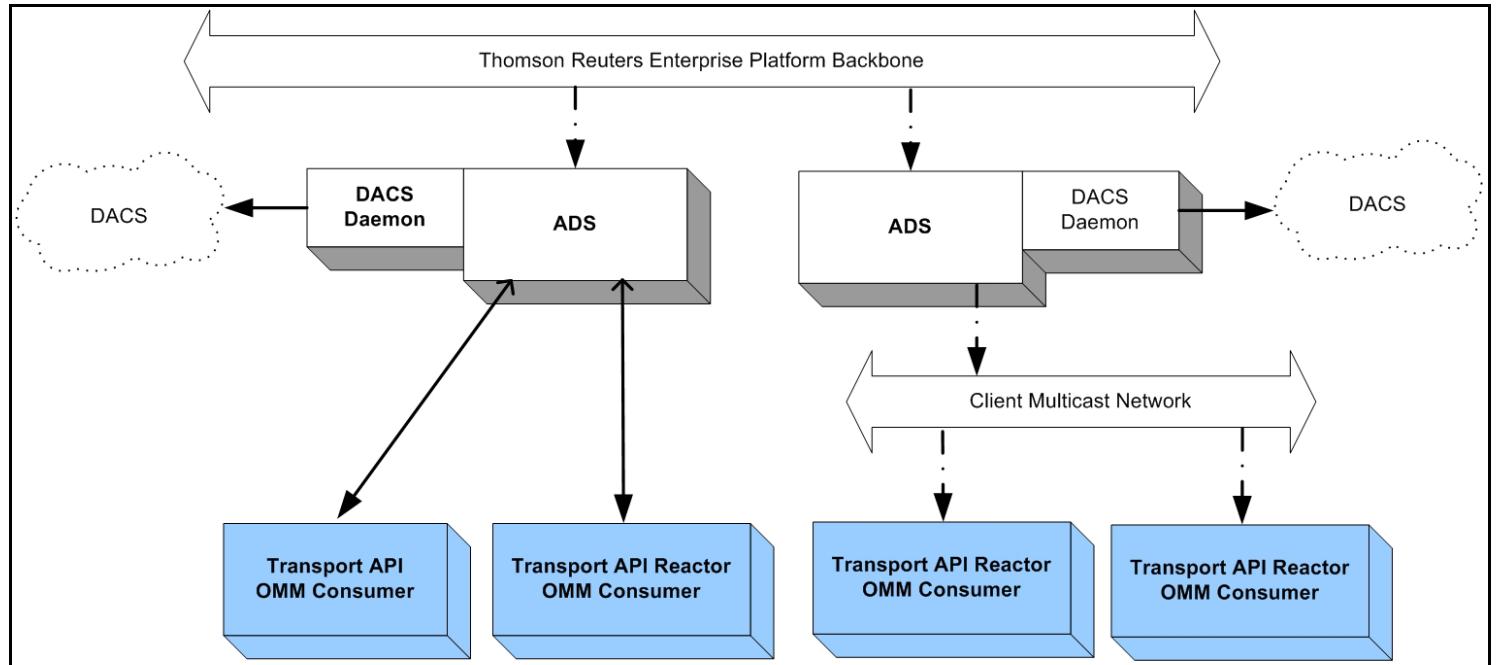


Figure 21. Transport API and Advanced Distribution Server

As a distribution device for market data, the ADS delivers data from the Advanced Data Hub (ADH). Because the ADS leverages multiple threads, it can offload the encoding, fan out, and writing of client data. By distributing its tasks in this fashion, ADS can support far more client applications than could any previous Thomson Reuters distribution solution.

The ADS supports two types of data delivery when communicating with API clients:

- Via point-to-point communication.
- Via multicast communication.

To take advantage of multicast communications, consumers must use a Value-Add component. For further information:

- On the Transport API Reactor component, refer to the *Transport API C Edition Value Added Components Developers Guide*.
- On network topologies as they relate to the Transport API, refer to Section 10.3.1.

4.3 Advanced Data Hub (ADH)

The **ADH** is a networked, data distribution server that runs in the TREP. It consumes data from a variety of content providers and reliably fans this data out to multiple ADSs over a backbone network (using either multicast or broadcast technology). Transport API-based non-interactive or interactive provider applications can publish content directly into an ADH, thus distributing data more widely across the network. NIP applications can publish content to an ADH via TCP or multicast connection types.

The ADH leverages multiple threads, both for inbound traffic processing and outbound data fanout. By leveraging multiple threads, ADH 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 ADH can support higher throughputs than could previous Thomson Reuters data hub solutions.

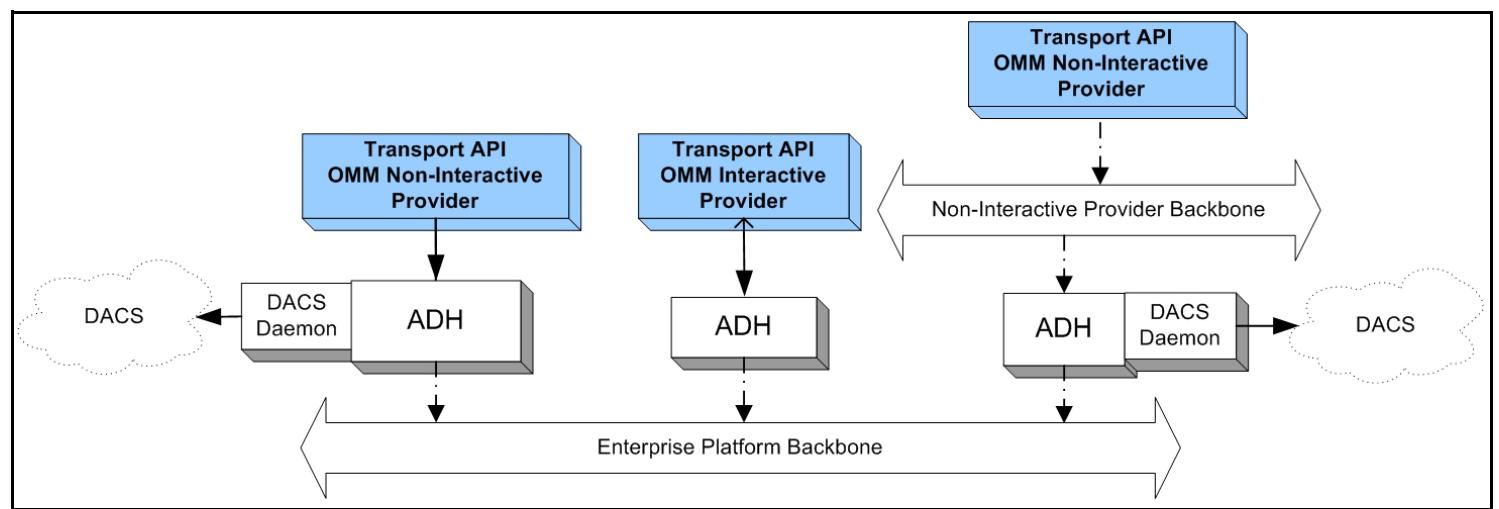


Figure 22. Transport API and the Advanced Data Hub

4.4 Elektron

Elektron is an open, global, ultra-high-speed network and hosting environment, which allows users to access and share various types of content. Elektron allows access to information from a wide network of content providers, including exchanges, where all exchange data is normalized using the OMM.

The Elektron Edge Device, based on ADS technology, is the access point for consuming this data. To access this content, a Transport API consumer application can connect directly to the Edge Device or via a cascaded Enterprise Platform architecture (as illustrated in the following diagram).

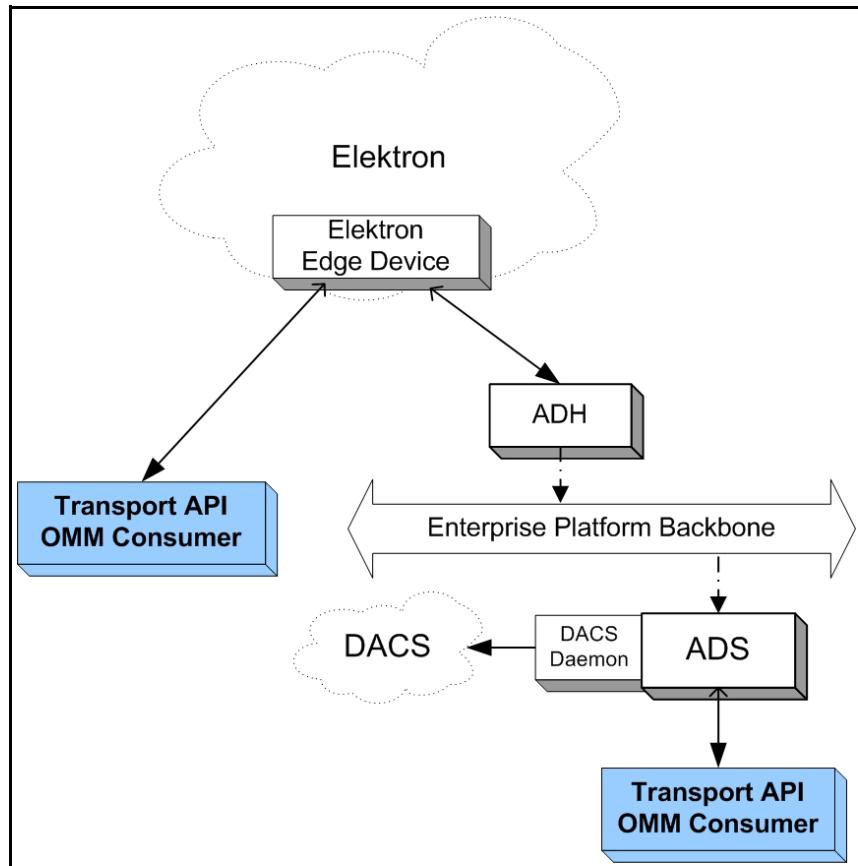


Figure 23. Transport API and Elektron

4.5 Data Feed Direct

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

To access this content, a Transport API consumer application can connect directly to the Data Feed Direct or via a cascaded TREP architecture.

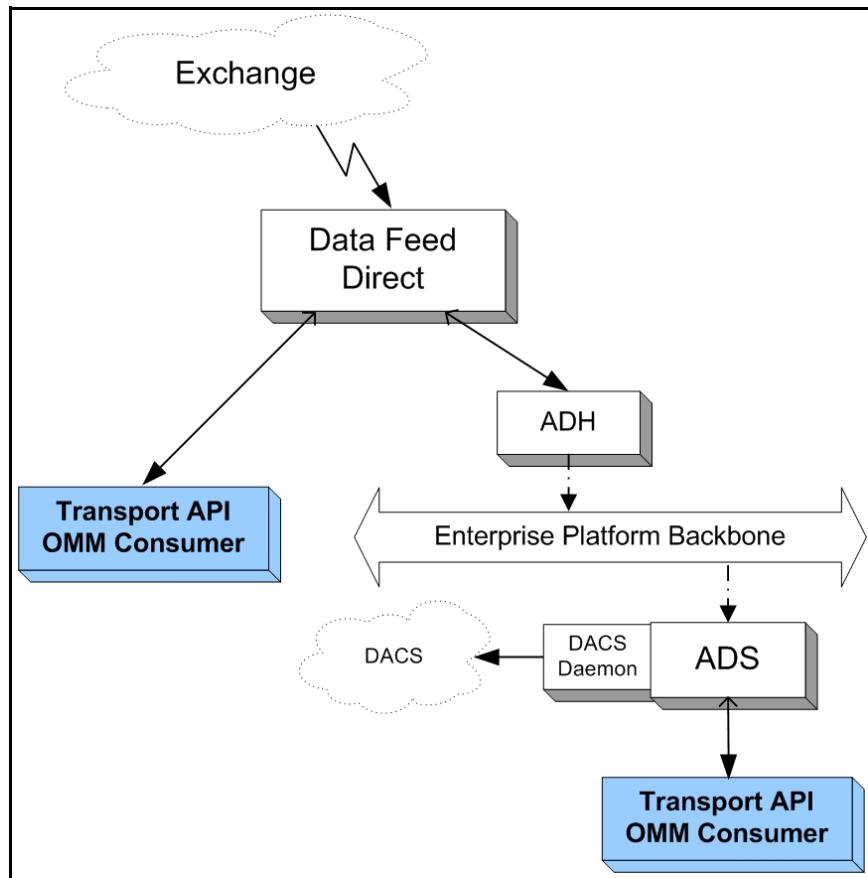


Figure 24. Transport API and Data Feed Direct

4.6 Internet Connectivity via HTTP and HTTPS

OMM consumer and Provider applications can use the Transport API to establish connections by tunneling through the Internet.

- OMM consumer and NIP applications can establish connections via HTTP tunneling.
- ADS and OMM Interactive Provider applications can accept incoming 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 Thomson Reuters Hosted Solutions, performing key and certificate exchange.
- Consumer-side functionality leverages Microsoft WinINET. Users can configure certificate and proxy use via Internet Explorer. Because of its dependency on the Microsoft WinINET library, consumer HTTP and HTTPS tunneling are available only on supported Windows platforms.

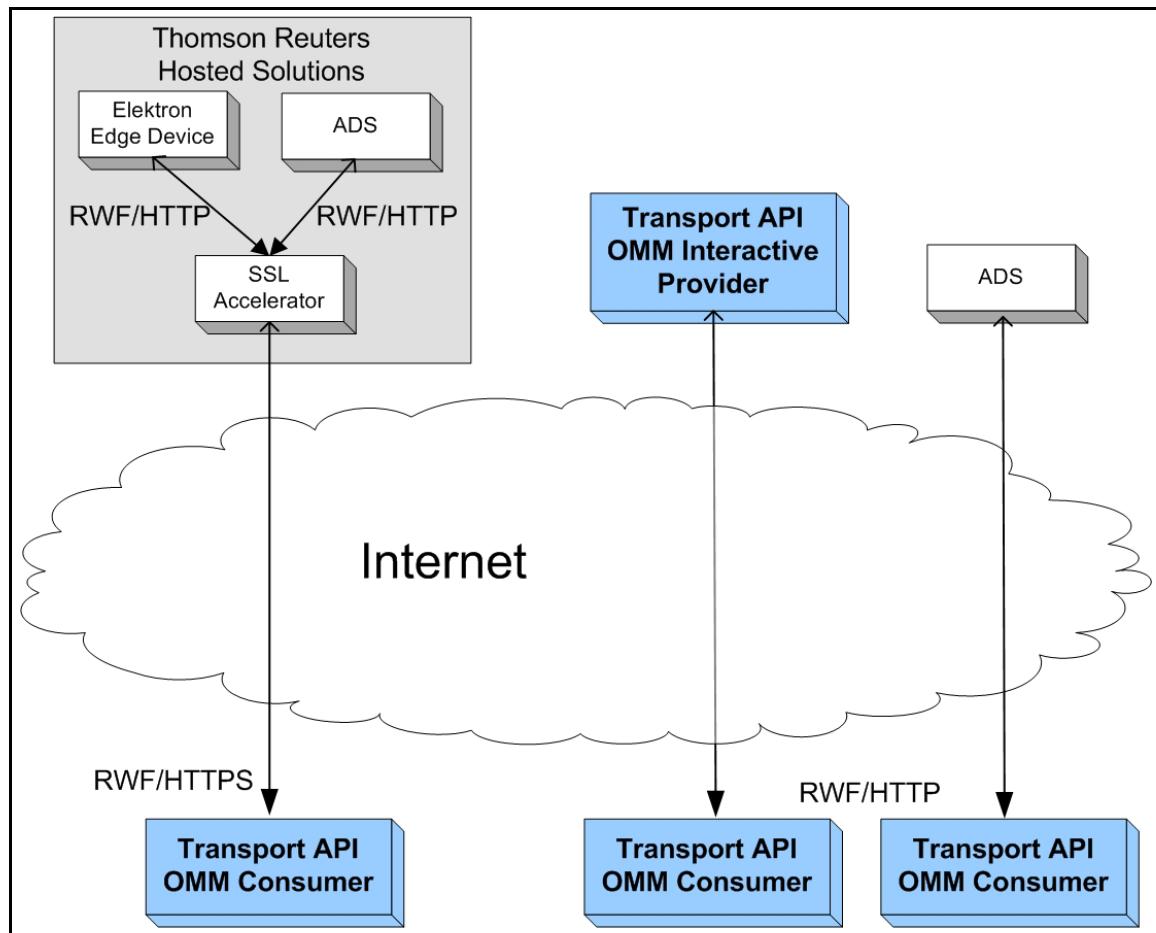


Figure 25. Transport API and Internet Connectivity

4.7 Direct Connect

The Transport API allows OMM Interactive Provider applications and OMM consumer applications to directly connect to one another. This includes OMM applications written to RFA. The following diagram illustrates various direct connect combinations.

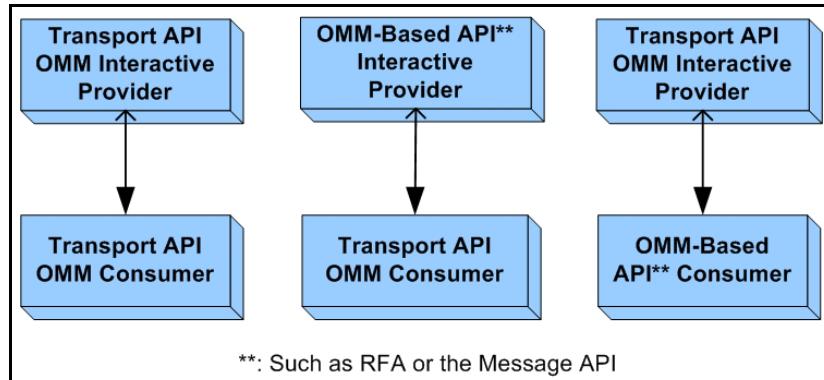


Figure 26. Transport API and Direct Connect

Chapter 5 Model and Package Overviews

5.1 Transport API Models

5.1.1 Open Message Model (OMM)

The **Open Message Model (OMM)** is a collection of message header and data constructs. Some OMM 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 OMM data constructs in various ways to model data ranging from simple (i.e., flat) primitive types to complex multi-level hierachal data.

The layout and interpretation of any specific OMM model (also referred to as a domain model) is described within that model's definition and is not coupled with the API. The OMM 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 Transport API provides structural representations of OMM constructs and manages the RWF binary-encoded representation of the OMM. Users can leverage Thomson Reuters-provided OMM constructs to consume or provide OMM data throughout the Enterprise Platform.

5.1.2 Reuters Wire Format (RWF)

RWF is the encoded representation of the OMM; 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. RWF allows for serializing OMM message and data constructs in an efficient manner while still allowing you to model rich content types. You can use RWF to distribute field identifier-value pair data (similar to Marketfeed), self-describing data (similar to Qform), as well as more complex, nested hierachal content.

5.1.3 Domain Message Model

A Domain Message Model (DMM) describes a specific arrangement of OMM message and data constructs. A DMM defines any:

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

Unless a DMM 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 Reuters Domain Model

A **Reuters Domain Model (RDM)** is a domain message model typically provided or consumed by a Thomson Reuters product (i.e., the Enterprise Platform, Data Feed Direct, or Elektron). Some currently-defined RDMs 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). Thomson Reuters's defined models have a domain value of less than 128. For extended definitions of the currently-defined Reuters Domain Models, refer to the *Transport API RDM Usage Guide*.

5.1.3.2 User-Defined Domain Model

A **User-Defined Domain Model** is a DMM 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 an RDM. Any user-defined model must use a domain value between 128 and 255.

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

5.2 Packages

The 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 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 import from the `com.thomsonreuters.upa.transport` package.

The Transport Package is described in more detail in Chapter 9.

5.2.2 Codec Package

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

- Expose data types (primitive and container types) and manage their RWF binary representation. These data types in turn make up components of OMM 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 RWF binary-encoded representation. The Codec defines message header elements that flow between various applications in the Enterprise Platform (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 import from the `com.thomsonreuters.upa.codec` package.

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

Chapter 6 Building an OMM Consumer

6.1 Overview

This chapter provides an overview of how to create an OMM consumer application. An OMM consumer application can establish a connection to other OMM interactive provider applications, including the TREP, Data Feed Direct, and Elektron. After connecting successfully, an OMM consumer can then consume (i.e., send data requests and receive responses) and publish data (i.e., post data).

The general process can be summarized by the following steps:

- Establish network communication
- Log in
- Obtain source directory information
- Load or download all necessary dictionary information
- Issue requests, process responses, and/or post information
- Log out and shut down

The **Consumer** example application, included with the Transport API products, provides an example implementation of an OMM consumer application. The application is written with simplicity in mind and demonstrates the uses of the 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 Transport API consumer application is to establish a network connection with its peer component (i.e., another application with which to interact). An OMM 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 `Channel`. The connection can be terminated if ping heartbeats are not sent or received within the expected time frame. Thomson Reuters recommends sending ping messages at intervals one third the size of the ping timeout.

Detailed information and use case examples for using RSSL Transport are provided in Chapter 9, Transport Package Detailed View.

6.3 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM 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 12, Message Package Detailed View) and the Data Package (described in Chapter 11, Data Package Detailed View). Further information about Login domain usage and messaging is available in the *Transport API RDM Usage Guide*.

6.4 Obtain Source Directory Information

The Source Directory domain model conveys information about all available services in the system. An OMM consumer typically requests a Source Directory to retrieve information about available services and their capabilities. This includes information about supported domain types, the service's state, the quality of service (QoS), and any item group information associated with the service. At minimum, Thomson Reuters 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 OMM 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 Section 13.4.

Content is encoded and decoded using the Transport API's Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View). Information about the Source Directory domain and its associated filter entry content is available in the *Transport API RDM 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 Thomson Reuters **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 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 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 RDM Usage Guide*.

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).

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 Thomson Reuters are defined in the *Transport API RDM Usage Guide*.

At this point, an OMM 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 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 or posting content, it should close all open streams and shut down the network connection. Issuing an **CloseMsg** 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 obtained from **Channel.getBuffer**. Calling **channel.close** terminates the connection to the provider application. Detailed information and transport code examples are provided in Chapter 9, Transport Package Detailed View.

6.8 Additional Consumer Details

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

- The consumer application demonstrates one way of implementing of an OMM 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 10, Encoding and Decoding Conventions.
- 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 a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 9, Transport Package Detailed View.
- For specific information about the DMMs required by this application type, refer to the *Transport API RDM Usage Guide*.

Chapter 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 OMM 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
- Disconnect consumers and shut down

The **Provider** example application included with the 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 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 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 an OMM consumer application attempts to connect, the provider uses the `Server.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 `Channel` 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. Thomson Reuters recommends sending ping messages at intervals one-third the size of the ping timeout.

For detailed information and use cases for the RSSL Transport, refer to Chapter 9, 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 **RefreshMsg** 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 **StatusMsg**, 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 Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View). For further information on Login domain usage and messaging, refer to the *Transport API RDM Usage Guide*.

7.4 Provide Source Directory Information

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

- The Source Directory Info filter contains the name and **serviceId** for each available service. The interactive provider should populate the filter with information specific to the services it provides.
- The Source Directory State filter contains status information for the service informing the consumer whether the service is Up (available) or Down (unavailable).
- The Source Directory Group filter conveys item group status information, including information about group states, as well as the merging of groups. If a provider determines that a group of items is no longer available, it can convey this information by sending either individual item status messages (for each affected stream) or a Directory message containing the item group status information. Additional information about item groups is available in Section 13.4.

Content is encoded and decoded using the Transport API's Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View). For details on the Source Directory domain and all of its associated filter entry content, refer to the *Transport API RDM 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 Thomson Reuters **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 ADH, a provider application may also download the RWFFId 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 ADH supporting 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 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. For available Dictionary utility methods, refer to the *Transport API Java Edition RDM Usage Guide*.

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).

Information about the Login and Dictionary domains, their expected content and formatting, and dictionary utility functions, is available in the *Transport API RDM 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 `msgKey` identification information received against the content available from the provider
- Determining whether it can provide the requested QoS
- Ensuring that the consumer does not already have a stream open for the requested information

If a provider can service a request, it should send appropriate responses. However, if the provider cannot satisfy the request, the provider should send a `StatusMsg` to indicate the reason and close the stream. All requests and responses should follow specific formatting as defined in the domain model specification. For details on all domains provided by Thomson Reuters, refer to the *Transport API RDM 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 Section 13.9.

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

7.7 Disconnect Consumers and Shut Down

When shutting down, the provider application should close the listening socket by calling `Server.close method`. 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 `StatusMsg` 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 `Channel.getBuffer` and call `Channel.close` for each connected client.

For detailed information and use case examples for the transport, refer to Chapter 9, Transport Package Detailed View.

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.

7.8 Additional Interactive Provider Details

For specific details about OMM Interactive Providers and the 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 have additional information about specific implementation and behaviors.
- To review high-level encoding and decoding concepts, refer to Chapter 10, Encoding and Decoding Conventions.
- 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 a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 9, Transport Package Detailed View.
- For specific information about DMMs required by this application type, refer to the *Transport API Java Edition RDM Usage Guide*.

Chapter 8 Building an OMM NIP

8.1 Overview

This chapter provides an outline of how to create an OMM NIP application which can establish a connection to an ADH server. Once connected, an OMM NIP can publish information into the ADH cache without needing to handle requests for the information. The ADH can cache the information and along with other Enterprise Platform components, provide the information to any OMM 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 Transport API package, the **NIPProvider** example application provides an implementation of an NIP written with simplicity in mind and demonstrates the use of the 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 NIP application is to establish network communication with an ADH server. To do so, the OMM NIP typically creates an outbound connection to the well-known hostname and port of an ADH. The NIP 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 **Channel**. If ping heartbeats are not sent or received within the expected time frame, the connection can be terminated. Thomson Reuters recommends sending ping messages at intervals one-third the size of the ping timeout.

For detailed information on RSSL Transport and associated use case examples, refer to Chapter 9, Transport Package Detailed View.

8.3 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM NIP must register with the system using a Login request¹ prior to providing any content.

After receiving a Login request, the ADH determines whether the NIP is permissioned to access the system. The ADH sends a Login response to the NIP which indicates whether the ADH has granted it access. If the application is denied, the ADH closes the Login stream and the NIP application cannot perform any additional communication. If the application gains access to the ADH, 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 *Transport API RDM Usage Guide*.

8.4 Perform Dictionary Download

If connected to an ADH that support dictionary downloads, an OMM NIP can download the RWFFId and RWFEnum dictionaries to retrieve appropriate information when providing field list content. An OMM NIP can use this feature to ensure they are using the correct version of the dictionary or to encode data. The ADH supporting the Provider Dictionary Download feature sends a Login response message containing the **SupportProviderDictionaryDownload** login element. The dictionary request is send using the Dictionary domain model².

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 *Transport API RDM 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 NIP must provide a Source Directory refresh³ indicating:

- Service, service state, QoS, and capability information associated with the NIP
- Supported domain types and any item group information associated with the service.

At a minimum, Thomson Reuters recommends that the NIP 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 NIPs typically provide data on only one service.
- The Source Directory State filter contains status information for service. This informs the ADH 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 *Transport API RDM Usage Guide*.

1. Because this is done in an interactive manner, the NIP should assign a **streamId** with a positive value (which the ADH will reference) when sending its response.

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 NIP should use a **streamId** with a negative value.

8.6 Provide Content

After providing a Source Directory, the NIP application can begin pushing content to the ADH. Each unique information stream should begin with an **RefreshMsg**, conveying all necessary identification information for the content⁴. The initial identifying refresh can be followed by other status or update messages. Some ADH functionality, such as cache rebuilding, may require that NIP applications publish the message key on all **RefreshMsgs**. For more information, refer to component-specific documentation.

Note: Some components, depending on their specific functionality and configuration, require that NIP applications publish the **msgKey** in **UpdateMsgs**. To avoid component or transport migration issues, NIP 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 NIP 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 obtained from **Channel.getBuffer**. Calling **Channel.getBuffer** terminates the connection to the ADH. Detailed information for transport and associated use cases are provided in Chapter 9, Transport Package Detailed View.

8.8 Additional NIP Details

For specific details about OMM Non-Interactive Providers and the Transport API use, refer to the following locations:

- The **NIPProvider** application demonstrates one implementation of an OMM NIP 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 10, Encoding and Decoding Conventions.
- 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 a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 9, Transport Package Detailed View.
- For specific information about the DMMs required by the application, refer to the *Transport API Java Edition RDM Usage Guide*.

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.

Chapter 9 Transport Package Detailed View

9.1 Concepts

The Transport API offers a Transport Package capable of communicating with other OMM-based components, including but not limited to TREP, Elektron, EDF Direct, and other TREP API OMM-based applications. The Transport Package efficiently sends and receives data across TCP/IP-based networks, leverages HTTP or HTTPS 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 option¹ to the ability for an application to turn off all protective locking². 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 9.2.3.

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.

1. When this option is enabled, RSSL 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.

9.1.1 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.

9.1.1.1 Socket Transport

The Transport API provides a transport for efficiently distributing messages across a TCP/IP-based reliable network (**RSSL_CONN_TYPE_SOCKET**). This transport is capable of connecting to various OMM-based components, including but not limited to Enterprise Platform, Elektron, RDF Direct, and other Transport API or RFA OMM-based applications. On specific platforms, applications can also leverage tunneling through HTTP (**RSSL_CONN_TYPE_HTTP**) or HTTPS (**RSSL_CONN_TYPE_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 ADS or OMM Interactive Provider, or by OMM Non-Interactive Provider applications to connect to an ADH. Listening sockets are typically created by OMM Interactive Provider applications to allow OMM Consumer applications or ADSs to instantiate connections to it and request data.

9.1.1.2 Reliable Multicast Transport

The Transport API provides an efficient transport for exchanging messages over a UDP Multicast-based network (**RSSL_CONN_TYPE_RELIABLE_MCAST**). This transport leverages the same technology used on the Enterprise Platform Backbone to improve reliability of message delivery and automatically re-sequence out-of-order messages.

OMM Non-Interactive Provider applications may create multicast connections for publishing to an ADH. OMM Consumer applications may leverage the Transport API Reactor and its watchlist feature to create connections to an ADS. For more information on the Transport API Reactor, refer to the *Transport API C Edition Value Added Components Developers Guide*.

9.1.1.3 Sequenced Multicast Transport

The Transport API provides an efficient transport for reading messages over the UDP Multicast-based network (**RSSL_CONN_TYPE_SEQ_MCAST**). This transport is intended for use with Thomson Reuters's ultra-low-latency OMM multicast feed, Elektron Direct Feed (EDF). The Sequenced Multicast protocol is a special UDP multicast with sequencing that allows the user to ensure order and identify gaps.

For more information, refer to the [Elektron Direct Feed User Guide](#) (you must have a Customer Zone account to access this document online).

9.1.2 Channel Object

The `channel` 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 `Channel` structure, and the application does not need to create nor free memory (associated with the channel). The `Channel` 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 `Channel` use within the transport.

The following table describes the methods of the `channel` object.

METHOD	DESCRIPTION
SelectableChannel	Returns the <code>java.nio.channels.SelectableChannel</code> object that can be used in an I/O notification mechanism (e.g. to register a Selector). This is the <code>SelectableChannel</code> associated with this end of the network connection.
oldSelectableChannel	It is possible for a <code>SelectableChannel</code> 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 <code>TransportReturnCodes.READ_FD_CHANGE</code> (for further information, refer to Section 9.6). The previous <code>SelectableChannel</code> is stored in <code>oldSelectableChannel</code> so the application can properly unregister and then register the new <code>SelectableChannel</code> with their I/O notification mechanism.
blocking	A boolean representing the blocking mode of the <code>Channel</code> .
state	The state associated with the <code>Channel</code> . Until the channel has completed its initialization handshake and has transitioned to an active state, no reading or writing can be performed. Section 9.1.2.1 describes channel state values.
connectionType	A <code>ConnectionType</code> that indicates the type of underlying connection being used. For more information, refer to Section 9.1.2.2.
pingTimeout	When a <code>Channel</code> 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 <code>pingTimeout/3</code> seconds. For more information, refer to Section 9.12.
protocolType	When a <code>Channel</code> becomes active for a client or server, this is populated with the <code>protocolType</code> associated with the content being sent on this connection. If the <code>protocolType</code> indicated by a server does not match the <code>protocolType</code> that a client specifies, the connection will be rejected. The transport layer is data-neutral and allows the flow of any type of content. <code>protocolType</code> is provided to help client and server applications manage the information they communicate. For more details, refer to Section 10.5.1.
majorVersion	When a <code>Channel</code> 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 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 10.5.1.
minorVersion	When a <code>Channel</code> 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 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 10.5.1.

Table 4: `Channel` Methods

METHOD	DESCRIPTION
userSpecObject	A reference that can be set by the user of the Channel . 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 Channel with other user created information, such as a watch list associated with this connection.
init	Continues Channel initialization for non-blocking channels. For more details, refer to Section 9.5.
info	Gets information about this Channel . For more details, refer to Section Section 9.14.2.
ioctl	Set or get some I/O values programmatically. For more details, refer to Section 9.14.6.
bufferUsage	Gets the total number of used buffers for this Channel .
getBuffer	Retrieves a TransportBuffer for use. For more details, refer to Section 9.8.
releaseBuffer	Releases a TransportBuffer . Should only be used if the buffer could not be successfully written.
read	Read on this Channel . For more details, refer to Section 9.6.
write	Write on this Channel . For more details, refer to Section 9.9
packBuffer	For more details, refer to Section 9.11.
flush	Flush this Channel . For more details, refer to Section 9.10.2.
ping	Send a ping (i.e. heart beat) message to the far end of the connection.
close	Close the Channel .
reconnectClient	Used for tunneling solution to reconnect and bridge connections. This only applies to http and encrypted connections, where it might be needed to keep connections alive through proxy servers.

Table 4: **Channel** Methods (Continued)

9.1.2.1 Channel State Values

ENUMERATED NAME	DESCRIPTION
INACTIVE	Indicates that an <code>Channel</code> is inactive. This channel cannot be used. This state typically occurs after a channel is closed by the user.
INITIALIZING	Indicates that an <code>Channel</code> requires additional initialization. This initialization is typically additional connection handshake messages that need to be exchanged. When using blocking I/O, an <code>Channel</code> is typically active when it is returned and no additional initialization is required by the user.
ACTIVE	Indicates that an <code>Channel</code> is active. This channel can perform any connection-related actions, such as reading or writing.
CLOSED	Indicates that an <code>Channel</code> 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 <code>Error</code> information should be available for the user.

Table 5: Channel State Values

9.1.2.2 ConnectionTypes 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 9.3). Additionally, after a connection is established, the `Channel.connectionType` will indicate the connection type being used.

CONNECTIONTYPE	DESCRIPTION
SOCKET	Indicates that the <code>Channel</code> uses a standard, TCP-based socket connection. This type can be used to connect between any Transport-based applications.
HTTP	Indicates that the <code>Channel</code> tunnels using HTTP. This type can be used to connect between any Transport-based applications. For more information, refer to Section 4.6.
ENCRYPTED	Indicates that the <code>Channel</code> 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). Because data will already be decrypted when it arrives at the server, a <code>Channel</code> may indicate that a connection type is HTTP or SOCKET, even if the connection was established by specifying ENCRYPTED. For more information, refer to Section 4.6.

Table 6: ConnectionType Values

CONNECTIONTYPE	DESCRIPTION
RELIABLE_MCAST	<p>Indicates that the <code>Channel</code> uses a UDP-based, reliable multicast connection type.</p> <p>This connection type is available only to applications using the <code>Transport.connect</code> function to establish their connection. The reliable multicast connection type ensures proper ordering of content across the network and, through the use of an acknowledgment and retransmission mechanism, backfills recent packet gaps. In situations where a packet gap cannot be filled, the application is notified of the gap situation.</p> <p>The default behavior for this connection type is to stay connected to the multicast, even in a gap situation. This allows the application to attempt recovery in a manner that might minimize any affect on the network. You can control this behavior via the <code>disconnectOnGaps</code> option described in Table 17.</p>
SEQUENCED_MCAST	<p>Indicates that the <code>Channel</code> uses a UDP-based, sequenced multicast connection type.</p> <p>This connection type is available only to applications using the <code>Transport.connect</code> function to establish their connection. Though this connection type uses sequence numbers which enables gap detection, it only ensures the proper ordering of content across the network; it does not acknowledge or retransmit packets to fill a gap.</p> <p>The default behavior for this connection type is to stay connected to the multicast, even in a gap situation. This allows for the application to attempt recovery in a manner that might minimize any affect on the network. You can control this behavior via the <code>disconnectOnGaps</code> option described in Table 17.</p>
UNIDIR_SHMEM	<p>Indicates that the <code>Channel</code> is using a shared memory connection type.</p> <p>This connection type offers a one-way data flow from a single server to multiple clients using a shared memory segment for content delivery. However, the server and clients must run on the same machine.</p> <p>For compatibility purposes, this connection type provides a <code>Channel.SelectableChannel</code> to the application. This <code>SelectableChannel</code> will always indicate that something is available to read, even when there is not. This ensures that the application is reading content with as little latency as possible. If needed, the application can implement alternate approaches that would allow for a less CPU intensive read algorithm.</p> <p> Warning! Transport API applications using this connection type require appropriate run-time permissions to create and lock memory on the system (e.g. <code>mlock()</code>). See operating system-specific information for details on ensuring applications have proper system access rights.</p>

Table 6: ConnectionType Values (Continued)

9.1.3 Server Object

The `server` object is used to represent a server that is listening for incoming connection requests. Any memory associated with a `Server` structure is internally managed by the Transport Package, and the application does not need to create nor destroy this type. The `Server` is typically used to accept or reject incoming connection attempts. See the subsequent sections for more information about `server` use within the transport.

The following table describes `Server` methods.

METHOD	DESCRIPTION
accept	Accepts an incoming connection. For more details, refer to Section 9.4.2.
bufferUsage	Returns the total number of used buffers for the <code>Server</code> .
close	Closes a <code>Server</code> . Active <code>Channels</code> accepted from this <code>Server</code> will not be closed.
info	Gets information about the <code>server</code> . For more details, refer to section Section 9.14.5 (check link)
ioctl	Allows change some I/O values programmatically for a <code>Server</code> . For more details, refer to Section 9.14.5.
SelectableChannel	Represents a <code>java.nio.channels.SelectableChannel</code> that can be used in some kind of I/O notification mechanism (e.g. Selector). This is the <code>SelectableChannel</code> associated with listening socket. When triggered, this typically indicates that there is an incoming connection and <code>Server.accept</code> should be called.
state	The <code>ChannelState</code> associated with the <code>Server</code> . A server is typically returned as active unless an error occurred during the <code>Transport.bind</code> call or the <code>Close</code> method was called. Table 6 describes possible state values.
portNumber	The port number that this <code>Server</code> is bound to and listening for incoming connections on.
userSpecObject	A reference that can be set by the user of the <code>Server</code> . 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 <code>Server</code> with other user created information, such as a list of associated <code>Channel</code> objects.

Table 7: `Server` Methods

9.1.4 Transport Error Handling

Many Transport Package methods take a parameter for returning detailed error information. This `Error` object is 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 (e.g. `Transport.connect`), 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. 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 <code>Channel</code> on which the error occurred.
errord	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 <code>errno</code> or 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.

Table 8: `Error` Methods

METHOD	DESCRIPTION
text ^a	Detailed text describing the error. This can include Transport API- specific error information, underlying library-specific error information, or a combination of both.
clear	Clears the <code>Error</code> object.

Table 8: Error Methods (Continued)

a. `Error` text information is limited to 1,200 bytes in length.

9.1.5 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:

- `Channel.init` return codes: Section 9.5.4.
- `Channel.read` return codes: Section 9.6.3.
- `Channel.write` return codes: Section 9.9.5.
- `Channel.flush` return codes: Section 9.10.3.

TRANSPORT RETURN CODE	DESCRIPTION
SUCCESS	Indicates successful completion of the operation.
FAILURE	Indicates that initialization has failed and cannot progress. The <code>channel.state</code> should be CLOSED . See the <code>Error</code> content for more information.
INIT_NOT_INITIALIZED	Indicates that the Transport has not been initialized. See the <code>Error</code> content for more details. For details on initializing, refer to Section 9.2.

Table 9: General Transport Return Codes

9.1.6 Application Lifecycle

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

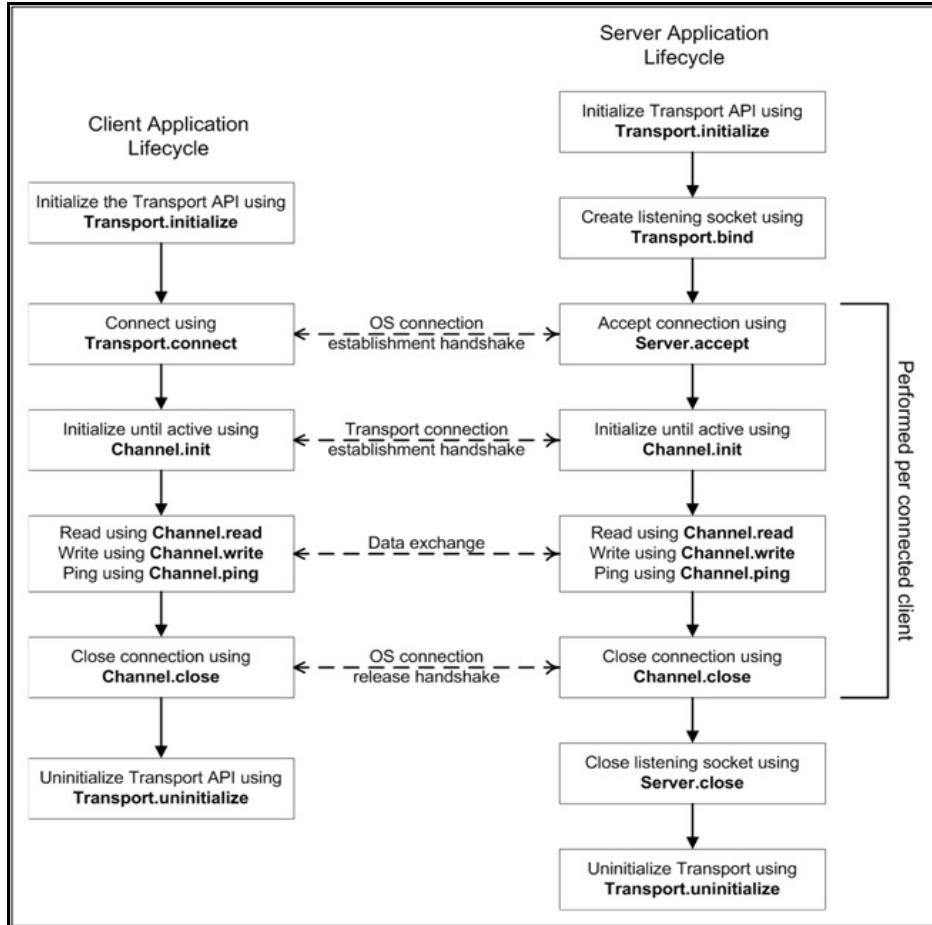


Figure 27. Transport Application Lifecycle

9.2 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. During this process, the transport also performs any necessary boot strapping associated with its underlying dependencies, such as WinSock or WinINET if on a Windows platform.

Similarly, when an application has completed its usage of the Transport, it must uninitialized it. The uninitialization process will release internal resources. These resources will eventually be garbage collected.

9.2.1 Initialization and Uninitialization Method

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 call. This creates and initializes internal memory (i.e., objects). The <code>initialize</code> method also allows the user to specify the locking model they want applied to the Transport. For more information, refer to Section 9.2.3.
uninitialize	The last Transport method that an application should call. This uninitialized internal resources. These resources will eventually be garbage collected.

Table 10: Initialization and Uninitialization Methods

9.2.2 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 / object creation 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`.

```
Error error = TransportFactory.createError();
InitArgs initArgs = TransportFactory.createInitArgs();
initArgs.globalLocking(true);
/* Starting Transport use, must call initialize first */
if (Transport.initialize(initArgs, error) != TransportReturnCodes.SUCCESS)
{
    System.out.println("Initialize(): failed <" + 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 1: Transport Initialization and Uninitialization

9.2.3 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:

- Global locking is controlled by `InitArgs.globalLocking(boolean)`, with `InitArgs` as a parameter to the `Transport.initialize()` method. After global locking is chosen, it cannot be changed without uninitialized and reinitializing the transport. This behavior ensures that a locking change is not pushed onto pre-established connections.
- For client connections, Channel locking is controlled on a per channel basis via `ConnectOptions.channelReadLocking(Boolean)` and `ConnectOptions.channelWriteLocking(Boolean)`, 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(Boolean)` and `AcceptOptions.channelWriteLocking(Boolean)`, with `AcceptOptions` as a parameter to the `Server.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(boolean)`, with `BindOptions` as a parameter to the `Transport.bind()` method (for more information, refer to Section 9.4.1.1).

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 <code>Channel</code> from one thread and write to the same <code>Channel</code> 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 11: Locking Types

9.3 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 or multicast network, often to some type of Provider running on a well known port number or multicast group address and port.

9.3.1 Network Topologies

The Transport API supports two types of network topologies:

- **unified**: A **unified** network topology is one where the **Channel** uses the same connection information (**address:port**) to send and receive all content.
- **segmented**: A **segmented** network topology is one where the **Channel** uses different connection information for sending and receiving. In the case of a **segmented** network, this allows for sent content and received content to be on different underlying **address:port** combinations.

On TCP-based networks, the Transport API supports only a **unified** topology (**ConnectionTypes.SOCKET**, **ConnectionTypes.HTTP**, and **ConnectionTypes.ENCRYPTED**), but on multicast-based networks, the Transport API supports both **unified** and **segmented** topologies (**ConnectionTypes.RELIABLE_MCAST** and **ConnectionTypes.SEQUENCED_MCAST**).

For configuration information on network topologies, refer to Table 14.

9.3.1.1 TCP-based Networks

If an application needs to communicate with multiple devices using a **ConnectionTypes.SOCKET**, **ConnectionTypes.HTTP**, or **ConnectionTypes.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:

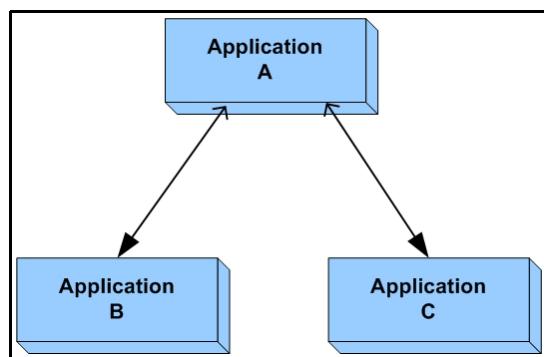


Figure 28. Unified TCP Network

In Figure 28, 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, OMM consumer and NIP applications connect as shown in Figure 29. The arrows used in the figure depict the directions in which connections are established. OMM consumers typically connect to a well known port number associated with some kind of Interactive Provider (e.g. ADS, Elektron), while OMM Non-Interactive Providers typically connect to a well known port on the ADH.

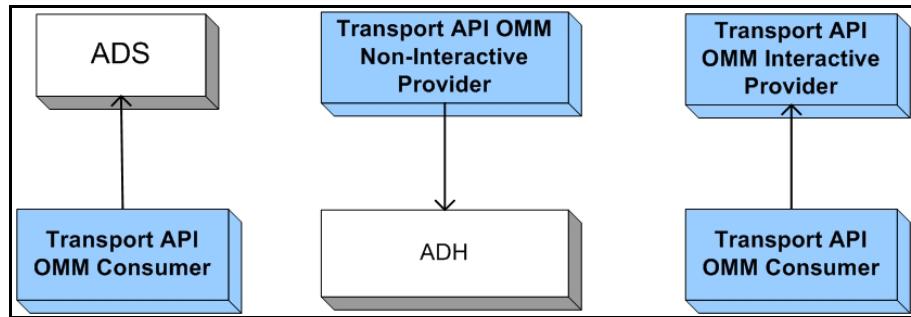


Figure 29. TCP Connection Creation

9.3.1.2 Multicast-based Networks: Unified

If an application wishes, it can communicate with multiple devices using a single connection to a multicast network (presuming the other devices access the same multicast network). In this case, a single transmission is sufficient to send data to all connected devices.

In the following diagram (Figure 30), all applications send and receive content on the same multicast network. Because the same network is used for sending and receiving traffic, all traffic is seen by all applications. Anything sent by one application will be received by all other applications on the network.

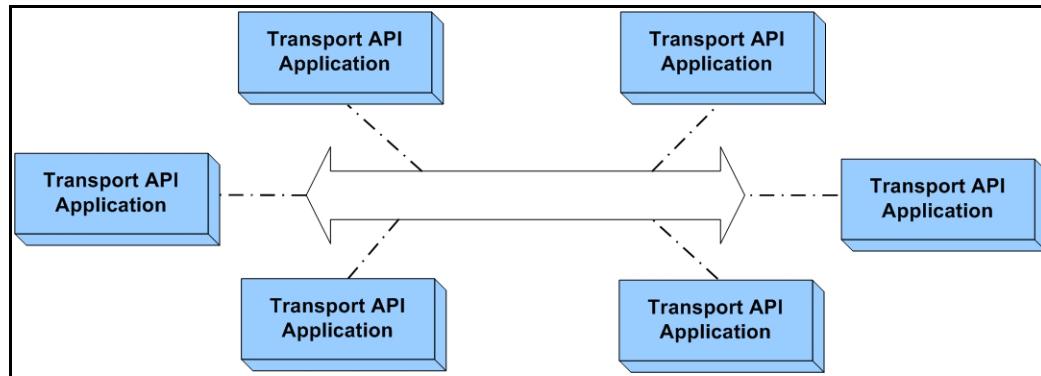


Figure 30. Unified Multicast Network

9.3.1.3 Multicast-based Networks: Segmented

In segmented multicast networks, applications transmit and receive data over different networks allowing users to separate applications based on the content they need to send or receive

In the following diagram (Figure 31):

- Applications A - C only send content on Network 1; they do not receive content from Network 1 (i.e., Application A does not see content sent by applications B or C). Applications A - C receive only the content sent on Network 2 (by applications D - F).
- Applications D - F only send content on Network 2; they do not receive content from Network 2 (i.e., Application D does not see content sent by applications E or F). Applications D - F receive only the content sent on Network 1 (by applications A - C).

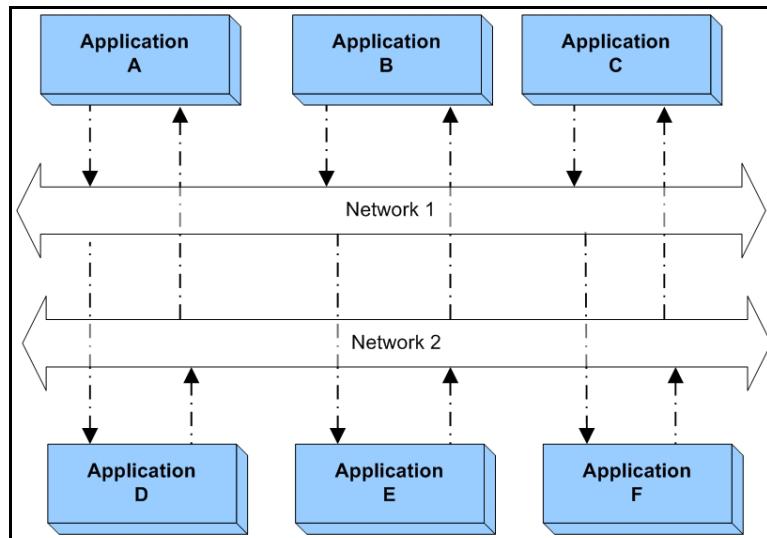


Figure 31. Segmented Multicast Network

The following diagram (Figure 32) illustrates OMM NIP applications using outbound multicast connections leveraging a segmented connection type. This allows the ADH to receive only content published by NIP applications (via the NIProv Send Network).

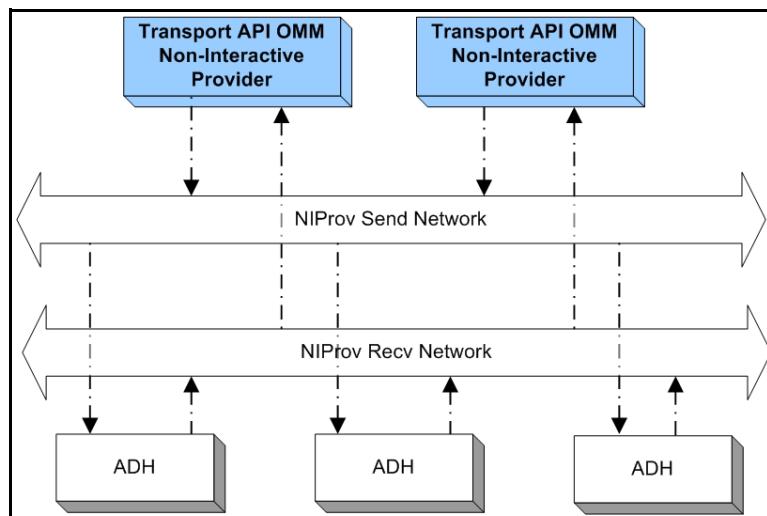


Figure 32. Multicast Connection Creation

9.3.2 Creating the Outbound Connection: `Transport.connect` Method

An application can create an outbound connection by using the `Transport.connect` method.

METHOD	DESCRIPTION
Transport.connect	<p>Establishes an outbound connection, which can leverage standard sockets, HTTP, or HTTPS. Returns an <code>Channel</code> that represents the connection to the user. In the event of an error, NULL is returned and additional information can be found in the <code>Error</code> structure.</p> <p>Connection options are passed in via an <code>ConnectOptions</code> object described in Table 13.</p> <p>Once a connection is established and transitions to the <code>ConnectionTypes.ACTIVE</code> state, this <code>Channel</code> can be used for other transport operations. For more information about channel initialization, refer to Section 9.5.</p>

Table 12: `Transport.connect` Method

9.3.2.1 `ConnectOptions` Methods

METHOD	DESCRIPTION
unifiedNetworkInfo	<p><code>unifiedNetworkInfo</code> object representing connection parameters used when sending and receiving on same network. This is typically used with <code>ConnectionTypes.SOCKET</code>, <code>ConnectionTypes.HTTP</code>, <code>ConnectionTypes.ENCRYPTED</code>, and fully connected/mesh multicast networks.</p> <p><code>unifiedNetworkInfo</code> is described in more detail in Section 9.3.2.2.</p>
segmentedNetworkInfo	Connection parameters when sending and receiving on different networks. This is typically used with multicast networks that have different groups of senders and receivers (e.g., NIProvider can send on one network and receive on another). <code>segmentedNetworkInfo</code> is described in more detail in Section 9.3.2.5.
connectionType	Type of connection to establish. Creation of encrypted, TCP-based socket, HTTP, and UDP-based multicast connection types are available across all supported platforms.
connectionType	<code>connectionType</code> s are described in more detail in Section 9.1.2.2.
guaranteedOutputBuffers	A guaranteed number of buffers made available for this <code>channel</code> to use while writing data. Guaranteed output buffers are allocated at initialization time.
guaranteedOutputBuffers	For more information, refer to Section 9.8.
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.
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 <code>pingTimeout</code> value on the <code>Channel</code> . When determining the desired ping timeout, the typically used rule of thumb is to send a heartbeat every <code>pingTimeout/3</code> seconds.
pingTimeout	For more information, refer to Section 9.12.
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.
compressionType	For more information about supported compression types and compression negotiation, refer to Section 9.4.3.

Table 13: `ConnectOptions` Methods

METHOD	DESCRIPTION
tunnelingInfo	tunnelingInfo object representing tunneling connection specific options. This is only valid for HTTP and Encrypted connection types. <ul style="list-style-type: none"> For more information on ConnectionTypes, refer to Section 9.1.2.2. For more information on TunnelingInfo, refer to Section 9.15.1.1.
credentialsInfo	credentialsInfo object representing Proxy credentials. For more information, refer to Section 9.15.2.3.
blocking	If set to true , blocking I/O will be used for this channel . When I/O is used in a blocking manner on a Channel , 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 Channel 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.
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 a Channel becomes active for a client or server, this information becomes available via the protocolType on the Channel . 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 10.5.1.
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 Channel . Typically, a major version increase is associated with the introduction of incompatible change. 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 10.5.1.
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 Channel . Typically, a minor version increase is associated with a fully backward compatible change or extension. 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 10.5.1.
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 Channel returned from the Transport.connect method. This information can be useful for coupling this Channel with other user created information, such as a watch list associated with this connection.
tcpOpts	TcpOpts object representing TCP-based connection type-specific options. These settings are used for ConnectionTypes of SOCKET , HTTP , and ENCRYPTED . For information about specific options, refer to Section 9.3.2.4.

Table 13: ConnectOptions Methods (Continued)

METHOD	DESCRIPTION
multicastOpts	A substructure containing multicast-based connection type-specific options. These settings are used for <code>ConnectionTypes.RELIABLE_MCAST</code> . For information about specific options, refer to Section 9.3.2.5.
shmemOpts	A substructure containing shared memory-based connection type-specific options. These settings are used for <code>ConnectionTypes.UNIDIR_SHMEM</code> . For information about specific options, refer to Section 9.3.2.6.
seqMCastOpts	A substructure containing multicast-based, connection type-specific options. These settings are used for <code>ConnectionTypes.SEQUENCED_MCAST</code> . For information about specific options, refer to Section 9.3.2.7.
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.
sysRecvBufSize	Accessor method, used to set or get the system's receive buffer size used for this connection. Not setting or setting of <code>0</code> indicates to use the default size of 64 KB. This can also be set or changed via <code>channel.ioctl</code> for values less than or equal to 64 KB. For values larger than 64KB, you must use this method to set <code>sysRecvBufSize</code> prior to the connect system call.
sysSendBufSize	Accessor method, used to set or get the system's send buffer size used for this connection. No setting or a setting of <code>0</code> indicates to use the default size of 64KB.
clear	Clears this object, so that it can be reused.

Table 13: `ConnectOptions` Methods (Continued)

9.3.2.2 UnifiedNetworkInfo Method 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 <code>address:serviceName</code> pair.
serviceName	Configures the numeric port number or service name (as defined in etc/services file) to use in a unified network configuration. All content will be sent and received on this <code>address:serviceName</code> 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 not specified the default network interface will be used.
unicastServiceName	Configures the numeric port number or service name (as defined in the etc/services file) to use for all unicast UDP traffic in a unified network configuration. This parameter is only required for multicast connection types (<code>ConnectionTypes.RELIABLE_MCAST</code> and <code>ConnectionTypes.SEQUENCED_MCAST</code>). If multiple connections or applications are running on the same host, this must be unique for each connection.

Table 14: `UnifiedNetworkInfo` Method Options

9.3.2.3 SegmentedNetworkInfo Method Options

METHOD	DESCRIPTION
recvAddress	Configures the receive address or hostname to use in a segmented network configuration. All content is received on this <code>recvAddress:recvServiceName</code> pair.
recvServiceName	Configures the receive network's numeric port number or service name (as defined in the etc/services file) to use in a segmented network configuration. All content is received on this <code>recvAddress:recvServiceName</code> pair.
sendAddress	Configures the send address or hostname to use in a segmented network configuration. All content is sent on this <code>sendAddress:sendServiceName</code> pair.
sendServiceName	Configures the send network's numeric port number or service name (as defined in the etc/services file) to use in a segmented network configuration. All content is sent on this <code>sendAddress:sendServiceName</code> 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 not specified the default network interface is used.
unicastServiceName	Configures the numeric port number or service name (as defined in the etc/services file) to use for all unicast UDP traffic in a unified network configuration. This parameter is only required for multicast connection types (<code>ConnectionTypes.RELIABLE_MCAST</code> and <code>ConnectionTypes.SEQUENCED_MCAST</code>). If multiple connections or applications are running on the same host, this must be unique for each connection.

Table 15: `SegmentedNetworkInfo` Method Options

9.3.2.4 TcpOpts Method Option

METHOD	DESCRIPTION
tcpNodelay	If set to <code>true</code> , this disables Nagle's Algorithm 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 small packet.

Table 16: `TcpOpts` Method Option

9.3.2.5 MCastOpts Method Options

METHOD	DESCRIPTION
disconnectOnGaps	Defaults to false, so if any multicast gap situation occur the underlying connection will not be closed. This allows the application to perform any item level recovery it may be able to do in order to reduce unnecessary bandwidth of full recovery on the multicast network. If set to true, the underlying connection will be closed when any multicast gap situation occurs. A multicast gap situation is reported as a return value of PACKET_GAP_DETECTED , SLOW_READER , or CONGESTION_DETECTED from <code>Channel.read</code> .
packetTTL	Controls the maximum number of components (network switches, etc.) a multicast datagram can traverse before it is removed from the network. Setting this to 0 , prevents packets from leaving the sending machine. When set to 255, the packet is not limited in the number of components it can traverse and is not removed from the network.
tcpControlPort	Specifies the port number that rrdump (a monitoring tool available in the TREP Infrastructure Tools package) should use. If set to or left as NULL, tcpControlPort uses the same port number as the unicastServiceName setting. If set to -1 , a control port will not be opened.
portRoamRange	Specifies the number of port numbers on which to attempt binding if the unicastServiceName fails to bind. The unicastServiceName is used as the starting point and will increment by 1 until it reaches the number specified in portRoamRange or successfully binds. If set to 0 , port roaming is disabled and the connection will attempt to bind only to the unicastServiceName .

Table 17: **MCastOpts** Method Options

9.3.2.6 ShmemOpts Method

METHOD	DESCRIPTION
maxReaderLag	Maximum number of messages that the client can have waiting to be read. If the client "lags" the server by more than this amount, the client will be disconnected on its next attempt to read. The default is equal to 75% of the number of buffers in the shared memory segment.

Table 18: **ShmemOpts** Method Option

9.3.2.7 SeqMCastOpts Method

METHOD	DESCRIPTION
maxMsgSize	Sets the maximum amount of data (in bytes) that can be sent and received on any packet over a ConnectionType.SEQUENCED_MCAST connection. Defaults to 3000 bytes.
instanceId	The originating IP address and port and the instanceId identify the sequenced multicast channel. When multiple applications run on the same host, unique instanceId values allow them to operate independently.

Table 19: **SeqMCastOpts** Method Option

9.3.3 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` object and then attempts to connect. If the connection succeeds, the application then registers the `Channel.SelectableChannel` with the I/O notification mechanism and continues with connection initialization (as described in Section 9.5).

```

Channel channel;
Selector selector;
Error error = TransportFactory.createError();
ConnectOptions cOpts = TransportFactory.createConnectOptions();

/* populate connect options, then pass to Transport.connect method - Transport should already be
   initialized */

cOpts.connectionType(ConnectionTypes.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(CompressionTypes.NONE); /* client does not desire compression for this
   connection */

/* populate version and protocol with RWF information */
cOpts.protocolType(Codec.protocolType());
cOpts.majorVersion(Codec.majorVersion());
cOpts.minorVersion(Codec.minorVersion());

if ((channel = Transport.connect(cOpts, error)) == null)
{
    System.out.println("Connection failure: " + error.text() + ", errorId=" + error.errorId()
        + " sysError=" + error.sysError());

    /* End application, uninitialized to clean up first */
    Transport.uninitialize();
    return;
}

/* Connection was successful, add SelectableChannel to I/O notification mechanism and initialize
   connection */
try
{
    /* register for read and write select */
    selector = Selector.open();
    channel.SelectableChannel().register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE,
        channel);

}
Catch (Exception e)
{

```

```
/* Selector.open() and SelectableChannel.register() 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 9.5 for more details. */
```

Code Example 2: Creating a Connection Using `Transport.connect`

9.3.4 Tunneling Connection Keep Alive

A client connection that is leveraging a connection type of `ConnectionTypes.HTTP` or `ConnectionTypes.ENCRYPTED` may be connecting through proxy devices as it tunnels through the Internet. Some proxy devices will force-close connections after certain elapsed time or time of day requirements are met. If one of these proxy devices is in a tunneling connections path, it can result in periodic connection loss. The Transport API Transport provides the `Channel.reconnectClient` method which allows a tunneling client application to pro-actively create another connection and bridge data flow from the existing connection, which will be closed, to the new connection. An application can use this, along with knowledge of the proxy device's time requirements, to keep an applications connection alive beyond the time limits enforced by the proxy which helps to avoid data recovery scenarios. This method is not used to perform any kind of connection or data recovery after a connection is closed or disconnected or for any non-tunneled connection types.

9.4 Server Creation and Accepting Connections

9.4.1 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 the `Transport.connect` method. 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).

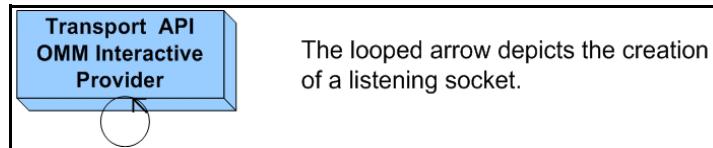


Figure 33. Transport API Server Creation

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

METHOD	DESCRIPTION
Transport.bind	<p>Establishes a listening socket connection, which supports connections from standard socket and HTTP <code>Transport.connect</code> users. Returns a <code>Server</code> 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 <code>Error</code> structure.</p> <p>Options are passed in via a <code>BindOptions</code> object described in Section 9.4.1.1.</p> <p>Once a listening socket is established, this <code>Server</code> can begin accepting connections. For more information, refer to Section 9.4.2.</p>

Table 20: `Transport.bind` Method

9.4.1.1 BindOptions Methods

METHOD	DESCRIPTION
connectionType	The type of connection to establish. <code>ConnectionTypes</code> are described in more detail in Table 7.
serviceName	A character representation of a numeric port number or service name (as defined in the <code>etc/services</code> file) on which to bind and open a listening socket.
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 ^a . If the loopback address (127.0.0.1) is specified, connections can be accepted only when instantiating from the local machine ^b .
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 <code>maxFragmentSize</code> buffers. This is different from application level message fragmentation done via the Message Package (as discussed in Section 13.1). 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 9.9.

Table 21: `BindOptions` Methods

METHOD	DESCRIPTION
numInputBuffers	The number of sequential input buffers used by each <code>Channel</code> 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 <code>maxFragmentSize</code> bytes. Input buffers are allocated at initialization time.
guaranteedOutputBuffers	A guaranteed number of buffers made available for each <code>Channel</code> to use while writing data. Each buffer is created to contain <code>maxFragmentSize</code> bytes. Guaranteed output buffers are allocated at initialization time. For more information, refer to Section 9.8.
	Note: For <code>ConnectionTypes.UNIDIR_SHMEM</code> , this parameter determines the number of buffers in the shared memory segment. The size of the shared memory segment will approximate <code>guaranteedOutputBuffers * maxFragmentSize</code> .
maxOutputBuffers	The maximum number of output buffers allowed for use by each <code>Channel</code> . (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) is equal to the number of shared pool buffers that each <code>Channel</code> is allowed to use. Shared pool buffers are only used if all <code>guaranteedOutputBuffers</code> are unavailable. If equal to the <code>guaranteedOutputBuffers</code> value, no shared pool buffers are available.
sharedPoolSize	<p>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 <code>Channel</code>, where each channel is allowed to use up to (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) number of buffers. Each shared pool buffer will be created to contain <code>maxFragmentSize</code> bytes.</p> <p>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 9.8.</p> <p>Note: It is considered an invalid configuration to allow more shared pool buffers (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) than the <code>sharedPoolSize</code>. If this happens, an error is returned from the <code>Transport.bind</code> method.</p>
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 9.2.3.
pingTimeout	<p>The server's maximum allowable ping timeout value. This is the largest possible value allowed in the negotiation between the client and the server's <code>pingTimeout</code> value. After the connection becomes active, the actual negotiated value becomes available through the <code>pingTimeout</code> value on the <code>Channel</code>. When determining the desired ping timeout, the rule of thumb is to send a heartbeat every <code>pingTimeout/3</code> seconds.</p> <p>For more information, refer to Section 9.12.</p>
minPingTimeout	<p>The server's lowest allowable ping timeout value. This is the lowest possible value allowed in the negotiation between client and servers <code>pingTimeout</code> values. After the connection becomes active, the actual negotiated value becomes available through the <code>pingTimeout</code> value on the <code>Channel</code>. When determining the desired ping timeout, the rule of thumb is to send a heartbeat every <code>pingTimeout/3</code> seconds.</p> <p>For more information, refer to Section 9.12.</p>
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. TREP and other Thomson Reuters components typically require this to be set to true .
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. TREP and other Thomson Reuters components typically require this to be set to true .

Table 21: BindOptions Methods (Continued)

METHOD	DESCRIPTION
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 9.4.3.
compressionLevel	Sets the level of compression to apply. Allowable values are 0 to 9 . <ul style="list-style-type: none"> 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 9.4.3.
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 9.4.3.
serverBlocking	If set to true , blocking I/O will be used for this Server . When I/O is used in a blocking manner on a Server , the server.accept method will complete any initialization on the Channel 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.
channelsBlocking	If set to true , blocking I/O will be used for all connected Channels . When I/O is used in a blocking manner on a Channel , 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.
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 . When a Channel becomes active for a client or server, this information becomes available via the protocolType on the Channel . 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 10.5.1.
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 Channel . Typically, the major version increases with the introduction of a 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 10.5.1.

Table 21: BindOptions Methods (Continued)

METHOD	DESCRIPTION
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 Channel . 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 to Section 10.5.1.
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 Server returned from the Transport.bind method if a userSpecObject was not specified in the AcceptOptions . This information can be useful for coupling this Server with other user-created information, such as a list of connected Channels .
tcpOpts	An object containing options specific to TCP-based connection types. For information about specific options, refer to Section 9.3.2.4.
clear	Clears this object so that it can be reused.
componentVersion	An optional, user-defined component version string appended behind the standard UPA component version information. If the combined component version length exceeds the maximum supported by the Transport API, the user-defined information will be truncated.

Table 21: BindOptions Methods (Continued)

- a. **INADDR_ANY** is used
- b. **INADDR_LOOPBACK** is used

9.4.1.2 Transport.bind Listening Socket Connection Creation Example

The following example demonstrates basic `Transport.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 `Server.SelectableChannel` 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 9.4.2.

```

Server srvr = null;
BindOptions b0pts = TransportFactory.createBindOptions();
Selector selector = null;

/* populate bind options, then pass to bind method - UPA should already be initialized */

b0pts.serviceName("14002"); /* server is running on port number 14002 */
b0pts.pingTimeout(45); /* servers desired ping timeout is 45 seconds, pings should be sent every 15 */
b0pts.minPingTimeout(30); /* min acceptable ping timeout is 30 seconds, pings should be sent every 10 */

/* set up buffering, configure for shared and guaranteed pools */
b0pts.guaranteedOutputBuffers(1000);
b0pts.maxOutputBuffers(2000);
b0pts.sharedPoolSize(50000);
b0pts.sharedPoolLock(true);

b0pts.serverBlocking(false); /* perform non-blocking I/O */
b0pts.channelsBlocking(false); /* perform non-blocking I/O */
b0pts.compressionType(CompressionTypes.NONE); /* server does not desire compression for this
connection */

/* populate version and protocol with RWF information or protocol specific info */
b0pts.protocolType(Codec.protocolType());
b0pts.majorVersion(Codec.majorVersion());
b0pts.minorVersion(Codec.minorVersion());

if ((srvr = Transport.bind(b0pts, error)) == null)
{
    System.out.printf("Error (%d) (errno: %d) encountered with bind. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());

    /* End application, uninitialize to clean up first */
    Transport.uninitialize();
    return null;
}

/* Bind was successful, register Selector and wait for connections */
try
{
    selector = Selector.open();
}
catch (Exception e)
{

```

```
        System.out.println("Open Selector Exception: " + e.getMessage());
    }
try
{
    srvr.SelectableChannel().register(selector, SelectionKey.OP_ACCEPT, srvr);
}
catch (Exception e)
{
    System.out.println("Register Selector Exception: " + e.getMessage());
}

/* Use accept for incoming connections, read and write data to established connections, etc */
```

Code Example 3: Creating a Listening Socket Using [Transport.bind](#)

9.4.2 Accepting Connection Requests

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

METHOD NAME	DESCRIPTION
Server.accept	<p>Uses the <code>Server</code> that represents the listening socket connection and begins the process of accepting the incoming connection request. Returns a <code>Channel</code> that represents the client connection. In the event of an error, NULL is returned and additional information can be found in the <code>Error</code> object.</p> <p>The <code>Transport.accept</code> method can also begin the rejection process for a connection through the use of the <code>AcceptOptions</code> object as described in Section 9.4.2.1.</p> <p>Once a connection is established and transitions to <code>ChannelState.ACTIVE</code>, this <code>Channel</code> can be used for other transport operations. For more information about channel initialization, refer to Section 9.5.</p>

Table 22: `Server.accept` Method

9.4.2.1 AcceptOptions Methods

METHOD	DESCRIPTION
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 9.5.
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 <code>userSpecObject</code> of the <code>Channel</code> returned from the <code>Server.accept</code> method. If this value is not set, the <code>Channel.userSpecObject</code> will be set to the <code>userSpecObject</code> associated with the <code>Server</code> that is accepting this connection.
channelReadLocking	Sets or checks whether the connection will use locking on reading.
channelWriteLocking	Sets or checks whether the connection will use locking on writing.
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). <code>sysRecvBufSize</code> is set via the <code>BindOptions</code> (for details, refer to Section 9.4.1.1).
clear	Clears the object for reuse.

Table 23: `AcceptOptions` Methods

9.4.2.2 Server.accept Accepting Connection Example

The following example demonstrates basic `Server.accept` use. The application first populates `AcceptOptions` and then attempts to accept the incoming connection request. If the accept succeeds, the application registers the new `Channel.SelectableChannel` with the I/O notification mechanism and continues with connection initialization, described in Section 9.5.

```

/* Accept is typically called when servers socketId indicates activity */
Channel chnl = null;
AcceptOptions aOpts = TransportFactory.createAcceptOptions();

/* populate accept options, then pass to accept method - UPA should already be initialized */
aOpts.nakMount(false); /* allow the connection */

if ((chnl = srvr.accept(aOpts, error)) == null)
{
    System.out.printf("Error (%d) (errno: %d) encountered with accept. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());

    /* End application, uninitialized to clean up first */
    Transport.uninitialize();
    return null;
}

/* Accept was successful, register Selector and wait for connections */
Selector selector = null;
try
{
    selector = Selector.open();
}
catch (Exception e)
{
    System.out.println("Open Selector Exception: " + e.getMessage());
}
try
{
    chnl.SelectableChannel().register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE,
                                      chnl);
}
catch (Exception e)
{
    System.out.println("Register Selector Exception: " + e.getMessage());
}

/* Continue the connection initialization process, for details, refer to Section 9.5 */

```

Code Example 4: Accepting Connection Attempts using `Server.accept`

9.4.3 Compression Support

As mentioned, the Transport supports the use of data compression. The following table identifies supported `CompressionTypes`.

COMPRESSION TYPE	DESCRIPTION
NONE	Do not compress data on the connection.
ZLIB	Use zlib compression on the connection. Zlib, an open source utility, employs a variation of the LZ77 algorithm while compressing and decompressing data.
LZ4	Use LZ4 compression on the connection. LZ4 is a lossless data compression algorithm that is focused on speed of compression and decompression. It belongs to the LZ77 family of byte-oriented compression schemes.

Table 24: `CompressionTypes` Values

The use of compression is negotiated during the connection establishment process. If the client's configured `ConnectOptions.compressionType` and the server's `BindOptions.compressionType` match, compression will be leveraged for the connection. The server's specified `compressionLevel` will determine the quality of the compression, where a lower value favours less time consumed when compressing and a higher number compresses data into a smaller size. It is possible for a server to force the use of compression, regardless of the client's configuration. This can be achieved by setting the `BindOptions.forceCompression` parameter, in which case both the server's `compressionType` and `compressionLevel` will be used.

Though compression may be enabled on a connection, it is still possible for individual buffers to be uncompressed. For efficiency, the Transport uses a default compression threshold of thirty bytes. Any message larger than the threshold will be compressed. This threshold can be changed via the `Channel.ioctl` method (refer to Section 9.14). If a message is larger than the compression threshold, it is still possible to be uncompressed by calling `Channel.write` with `WriteArgs.flags` set to `WriteArgs.flagWriteFlags.DO_NOT_COMPRESS`. For more information, refer to Section 9.9.

9.5 Channel Initialization

After a `Channel` is returned from the client's `Transport.connect` or server's `Server.accept` call, the channel may need to continue the initialization process.

Note: For both client and server channels, to complete the channel initialization process, more than one call to `Channel.init` might be required.

Additional initialization is required as long as the `channel.state` is `ChannelState.INITIALIZING`.

- If using a non-blocking I/O, this is the typical state from which a `Channel` starts and multiple initialization calls might be needed to transition to active.
- If using a blocking I/O, when successful, `Transport.connect` and `Server.accept` return a completely initialized channel in an active state.

Internally, the initialization process involves several actions. The initialization includes any necessary Transport API 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 `channel.state` will transition.

- If the connection is accepted (i.e., all negotiations were successful), the `channel.state` will become `ChannelState.ACTIVE`.
- If the connection is rejected (i.e., due to either failed negotiation or a `Server` rejection of the connection by setting `AcceptOptions.nakMount` to `true`), the `Channel.state` will become `ChannelState.CLOSED`, and the application should close the channel to clean up any associated resources.

9.5.1 Channel.init Method

METHOD NAME	DESCRIPTION
Channel.init	<p>Continues initialization of a <code>Channel</code>. This channel could originate from <code>Transport.connect</code> or <code>Server.accept</code>. This method exchanges various messages to perform necessary Transport API negotiations and handshakes to complete channel initialization. If using blocking I/O, this method is typically not used because <code>Transport.connect</code> and <code>Server.accept</code> return active channels.</p> <p>Requires the use of the <code>InProgInfo</code> object, refer to Section 9.5.2.</p> <p>The <code>Channel</code> can be used for all additional transport functionality (e.g. reading, writing) after the <code>state</code> transitions to <code>ChannelState.ACTIVE</code>. If a connection is rejected or initialization fails, the state transitions to <code>ChannelState.CLOSED</code>, and the application should close the channel to clean up any associated resources.</p> <p>The return values are described in Section 9.5.4.</p>

Table 25: `Channel.init` Method

9.5.2 InProgInfo Object

Use the `InProgInfo` object with the `Channel.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 `SelectableChannel` and register the new `SelectableChannel` with the I/O notification mechanism in use with associated information being conveyed by `InProgInfo` and `InProgFlags`.

METHOD	DESCRIPTION
flags	<p>Combination of bit values to indicate special behaviors and presence of optional <code>InProgInfo</code> content.</p> <p><code>flags</code> uses the following enumeration values:</p> <ul style="list-style-type: none"> • NONE: Indicates that channel initialization is still in progress and subsequent calls to <code>Channel.init</code> are needed for completion. The call did not change the <code>SelectableChannel</code>. • SCKT_CHNL_CHANGE: Indicates that the call changed the <code>SelectableChannel</code>. The previous <code>SelectableChannel</code> is now stored in <code>InProgInfo.oldSelectableChannel</code> so it can be unregistered with the I/O notification mechanism. The new <code>SelectableChannel</code> is stored in <code>InProgInfo.newSelectableChannel</code> so it can be registered with the I/O notification mechanism. However, channel initialization is still in progress and subsequent calls to <code>Channel.init</code> are needed to complete it.
oldSelectableChannel	Populated if flags indicate that <code>Channel.init</code> needs to perform a socket change. If this occurs, the <code>oldSelectableChannel</code> contains the <code>java.nio.channels.SelectableChannel</code> associated with the previous connection so the application can unregister this with the I/O notification mechanism.
newSelectableChannel	Populated if flags indicate that <code>Channel.init</code> needs to perform a socket change. If this occurs, the <code>newSelectableChannel</code> contains the <code>java.nio.channels.SelectableChannel</code> associated with the new connection so the application can register this with the I/O notification mechanism.

Table 26: `InProgInfo` Methods

9.5.3 Calling Channel.init

Typically, calls to `Channel.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 `Channel.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 `Channel` with a `Selector` by calling `Channel.SelectableChannel.register` method with a `Selector` and `SelectionKey` of `OP_READ`, `OP_WRITE`, and `OP_CONNECT`. After the user calls the `Selector.select` method and the `Channel` is ready for writing (or ready to complete its connection sequence), the `Channel.init` method is called (this sends the initial connection handshake message). When the `Channel` has data to read, `init` is called - this typically reads the next portion of the handshake. This process continues until the connection is active.

A server application would typically register the `Server` with a `Selector` by calling the `Server.SelectableChannel.register` method with a `Selector` and `SelectionKey` of `OP_ACCEPT`. After the user calls the `Selector.select` method, and the `Server` is ready to accept a new connection, and the `Server.accept` method should be called. `accept` returns a `Channel`. Register the `Channel` with a `Selector` and `SelectionKey` of `OP_READ`. After the user calls the `Selector.select` method and the `Channel` has data to read, the `Channel.init` method is called (this typically reads the initial portion of the handshake and will send out any necessary response). This process continues until the connection is active.

9.5.4 Channel.init Return Codes

The following table defines the `TransportReturnCodes` that can occur when using `Channel.init`.

RETURN CODE	DESCRIPTION
SUCCESS	Indicates the initialization process completed successfully. The <code>Channel.state</code> should be <code>ChannelState.ACTIVE</code> .
FAILURE	Indicates that initialization has failed and cannot progress. The <code>Channel.state</code> should be <code>ChannelState.CLOSED</code> , and the application should close the channel to clean up associated resources. For more details, refer to the <code>Error</code> content.
CHAN_INIT_IN_PROGRESS	Indicates that initialization is still in progress. Check <code>InProgInfo.flags</code> to determine whether the <code>SelectableChannel</code> changed. The <code>Channel.state</code> should be <code>ChannelState.INIALIZING</code> .
CHAN_INIT_REFUSED	Indicates the connection was rejected. For more details, refer to the <code>Error</code> content.
INIT_NOT_INITIALIZED	Indicates that the Transport is not initialized. For more details, refer to the <code>Error</code> content. For information on initializing, refer to Section 9.2.

Table 27: `Channel.init` `TransportReturnCodes`

9.5.5 Channel.init Example

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

```
/* Channel.init() is typically called based on activity on the selector, though a timer or looping
   can be used - the Channel.init() method should continue to be called until the connection becomes
   active, at which point reading and writing can begin. */
InProgInfo inProgInfo = TransportFactory.createInProgInfo();
if (chnl.state() == ChannelState.INIALIZING)
{
    if ((retCode = chnl.init(inProgInfo, error)) < TransportReturnCodes.SUCCESS)
    {
        System.out.printf("Error (%d) (errno: %d) encountered with init. Error Text: %s\n",
                          error.errorId(), error.sysError(), error.text());
        /* The application should close the channel to clean up any associated resources. */
    }
    else
    {
        /* Handle return code appropriately */
        switch (retCode)
        {
            case TransportReturnCodes.CHAN_INIT_IN_PROGRESS:
                /* Initialization is still in progress, check the InProgInfo for additional information */
                if (inProgInfo.flags() == InProgFlags.SCKT_CHNL_CHANGE)
                {
                    System.out.println("\nSession In Progress - New Channel: " + chnl.SelectableChannel() +
                                      " Old Channel: " + inProgInfo.SelectableChannel());
                    /* cancel old channel read select */
                    try

```

```
{  
    SelectionKey key = inProgInfo.SelectableChannel().keyFor(selector);  
    key.cancel();  
}  
catch (Exception e) {} // old channel may be null so ignore  
/* add new channel read select */  
try  
{  
    chnl.SelectableChannel().register(selector, SelectionKey.OP_READ |  
        SelectionKey.OP_WRITE, chnl);  
}  
catch (Exception e)  
{  
    System.out.println("register select Exception: " + e.getMessage());  
}  
}  
else  
{  
    System.out.println("\nChannel " + chnl.SelectableChannel() + " In Progress...");  
}  
break;  
case TransportReturnCodes.SUCCESS:  
System.out.printf("Channel on port %d is now active - reading and writing can begin.\n",  
    chnl.SelectableChannel().socket().getLocalPort());  
break;  
default:  
System.out.println("\nBad return value portno=" +  
    chnl.SelectableChannel().socket().getLocalPort() + "<" + error.text() + ">");  
/* Likely unrecoverable, connection should be closed */  
break;  
}  
}
```

Code Example 5: Channel Initialization Process Using `Channel . i ni t`

9.6 Reading Data

When a client or server `Channel.state` is `ChannelState.ACTIVE`, an application can receive data from the connection. The arrival of this data is often announced by the I/O notification mechanism with which the `Channel.SelectableChannel` is registered. The Transport reads data from the network as a byte stream, after which it determines `TransportBuffer` 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 a `TransportBuffer` is returned from `Channel.read`, the contents are only valid until the next call to `Channel.read`.

To reduce potentially unnecessary copies, returned information simply points into the internal `Channel` input buffer. If the application requires the contents of the buffer beyond the next `Channel.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 `Channel.read` method will perform any necessary decompression prior to returning information to the application. For available compression types, refer to Section 9.4.3.

It is possible for `Channel.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 `Channel.read`.

If a packed buffer is received, each call to `Channel.read` returns an individual message (i.e., portion of contents) from the packed buffer. Every subsequent call to `channel.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 9.11.

9.6.1 Channel.read Method

METHOD	DESCRIPTION
Channel	<p>Provides the user with data received from the connection. This method expects the <code>Channel</code> to be in the active state. When data is available, a <code>TransportBuffer</code> referring to the information is returned, which is valid until the next call to <code>Channel.read</code>. If a blocking I/O is used, the <code>Channel.read</code> method will not return until there is information to return or an error has occurred.</p> <p>A <code>ReadArgs</code> 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 <code>Channel</code> input buffer. <code>ReadArgs</code> also conveys the number of bytes and uncompressed bytes read. The <code>ReadArgs.readRetVal</code> method is used to get the return code.</p> <p>An Error parameter passed into the method is used to convey error information if the <code>ReadArgs.readRetVal</code> value indicates an error.</p> <p>Return values are described in Section 9.6.3.</p>

Table 28: `Channel` Method

9.6.2 ReadFlags Values

FLAG VALUE	DESCRIPTION
NO_FLAGS	Channel data does not have associated read flags.
READ_NODE_ID	Channel data includes a valid node ID.
READ_SEQNUM	Channel data includes a sequence number.
READ_INSTANCE_ID	The message includes an instance ID.
READ_RETRANSMIT	Channel data is a retransmission of previous content.

Table 29: ReadFlags Values

9.6.3 Channel.read Return Codes

The following table defines `TransportReturnCodes` that can occur when using `Channel.read`.

RETURN CODE	BUFFER CONTENTS	DESCRIPTION
SUCCESS	Populated if the full buffer is available, NULL otherwise. The buffer's <code>length</code> indicates the number of bytes to which the <code>data</code> refers.	Indicates that the <code>Channel.read</code> 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 9.12.
Any positive value > 0	Populated if full buffer is available, NULL otherwise. The buffer's <code>length</code> indicates the number of bytes to which the <code>data</code> refers.	Indicates that the <code>Channel.read</code> 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 <code>Channel.read</code> method should be called again to ensure that the remaining bytes are processed. The ping timer should be updated (for details, refer to Section 9.12). Note: If there are additional bytes to process, you should call <code>Channel.read</code> 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.
READ_WOULD_BLOCK	NULL	Indicates that the <code>Channel.read</code> call has nothing to return to the user.
READ_PING	NULL	Indicates that a heartbeat message was received. The ping timer should be updated (for details, refer to Section 9.12).
FAILURE	NULL	Indicates a failure condition, often that the connection is no longer available. The <code>Channel</code> should be closed (for details, refer to Section 9.13). For more details, refer to <code>Error</code> content.

Table 30: Channel . read TransportReturnCodes

RETURN CODE	BUFFER CONTENTS	DESCRIPTION
PACKET_GAP_DETECTED	NULL	Indicates that a packet gap was detected in the inbound transport content. This may be recoverable above the transport layer, so the <code>Channel</code> is left in a connected state. If needed, an application can configure the transport to disconnect whenever this occurs by using the <code>disconnectOnGaps</code> option. For details on this option, refer to Section 9.3.2.5.
SLOW_READER	NULL	Indicates that the reader is not keeping up with the data rate and a packet gap was detected in the inbound transport content. This may be recoverable above the transport layer, so the <code>Channel</code> is left in a connected state. If needed, an application can configure the transport to disconnect whenever this occurs by using the <code>disconnectOnGaps</code> option. For details on this option, refer to Section 9.3.2.5.
CONGESTION_DETECTED	NULL	Indicates network congestion and that a gap was detected in the inbound transport content. This may be recoverable above the transport layer, so the <code>Channel</code> is left in a connected state. If needed, an application can configure the transport to disconnect whenever this occurs by using the <code>disconnectOnGaps</code> option. For details on this option, refer to Section 9.3.2.5.
READ_FD_CHANGE	NULL	Indicates that the connections <code>SelectableChannel</code> has changed. This can occur as a result of internal connection keep-alive mechanisms. The previous <code>SelectableChannel</code> is stored in the <code>Channel.oldSelectableChannel</code> so it can be removed from the I/O notification mechanism. The <code>Channel.oldSelectableChannel</code> contains the new file descriptor, which should be registered with the I/O notification mechanism.
READ_IN_PROGRESS	NULL	Indicates that a <code>channel.read</code> call on the <code>Channel</code> is already in progress. This can be due to another thread performing the same operation.
INIT_NOT_INITIALIZED	NULL	Indicates that the Transport has not been initialized. See the <code>Error</code> content for more details. For information on initializing, refer to Section 9.2.

Table 30: `Channel.read` TransportReturnCodes (Continued)

9.6.4 Channel.read Example

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

```

/* Channel.read() use, be sure to keep track of the return values from read so data is not
   stranded in the input buffer */
ReadArgs readArgs = TransportFactory.createReadArgs();
TransportBuffer buffer = null;

if ((buffer = chnl.read(readArgs, error)) != null)
{
    /* if a buffer is returned, we have data to process and code is success */
    /* Process data and update ping monitor (Section 9.8) since data was received */

    /* Process data and update ping monitor (Section 9.8) since data was received */
    if (readArgs.readRetVal() > TransportReturnCodes.SUCCESS)
    {
        /* There is more data to read and process and I/O notification may not trigger for it */
        /* Either schedule another call to read or loop on read until */
        /* retCode == TransportReturnCodes.SUCCESS and there is no data left in internal input buffer */
    }
}
else
{
    /* Handle return codes appropriately, not all return values are failure conditions */
    int retCode = readArgs.readRetVal();
    switch(retCode)
    {
        case TransportReturnCodes.READ_PING:
            /* Update ping monitor (for details, refer to Section 9.12) */
            break;
        case TransportReturnCodes.READ_FD_CHANGE:
        {
            System.out.println("\nRead() Channel Change - Old Channel: " + chnl.oldSelectableChannel()
+
                " New Channel: " + chnl.SelectableChannel());
            /* File descriptor changed, typically due to tunneling keep-alive */
            /* Unregister old socketId and register new socketId */
            try
            {
                SelectionKey key = chnl.SelectableChannel().keyFor(selector);
                key.cancel();
            }
            catch (Exception e) {} // old channel may be null so ignore
            /* Up to application whether to register with write set - depends on need for write
               notification */
            try
            {
                chnl.SelectableChannel().register(selector, SelectionKey.OP_READ |

```

```
        SelectionKey.OP_WRITE, chnl);
    }
    catch (Exception e)
    {
        System.out.println("\nregister select Exception: " + e.getMessage());
    }
}
break;
case TransportReturnCodes.READ_WOULD_BLOCK: /* Nothing to read */
case TransportReturnCodes.READ_IN_PROGRESS: /* Reading from multiple threads: this is
                                             dangerous*/
    /* Handle as application sees fit, output warning, etc */
    break;
case TransportReturnCodes.INIT_NOT_INITIALIZED:
case TransportReturnCodes.FAILURE:
    System.out.printf("Error (%d) (errno: %d) encountered with read. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
    /* Connection should be closed */
    break;
default:
    System.out.printf("Unexpected return code (%d) encountered!", retCode);
    /* Likely unrecoverable, connection should be closed */
}
}
```

Code Example 6: Receiving Data Using Channel . read

9.7 Writing Data: Overview

When a client or server `Channel.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 `TransportBuffer` from the Transport buffer pool (refer to Section 9.8).

After a buffer is acquired, the user can populate the `TransportBuffer` directly or use the Transport API to encode.

At this point, the user can choose to pack additional information into the same buffer (refer to Section 9.11) or add the buffer to the transports outbound queue (refer to Section 9.9). 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 9.10.2). An I/O notification mechanism can be used to help with determining when the network is able to accept additional bytes for writing. The RSSL 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.

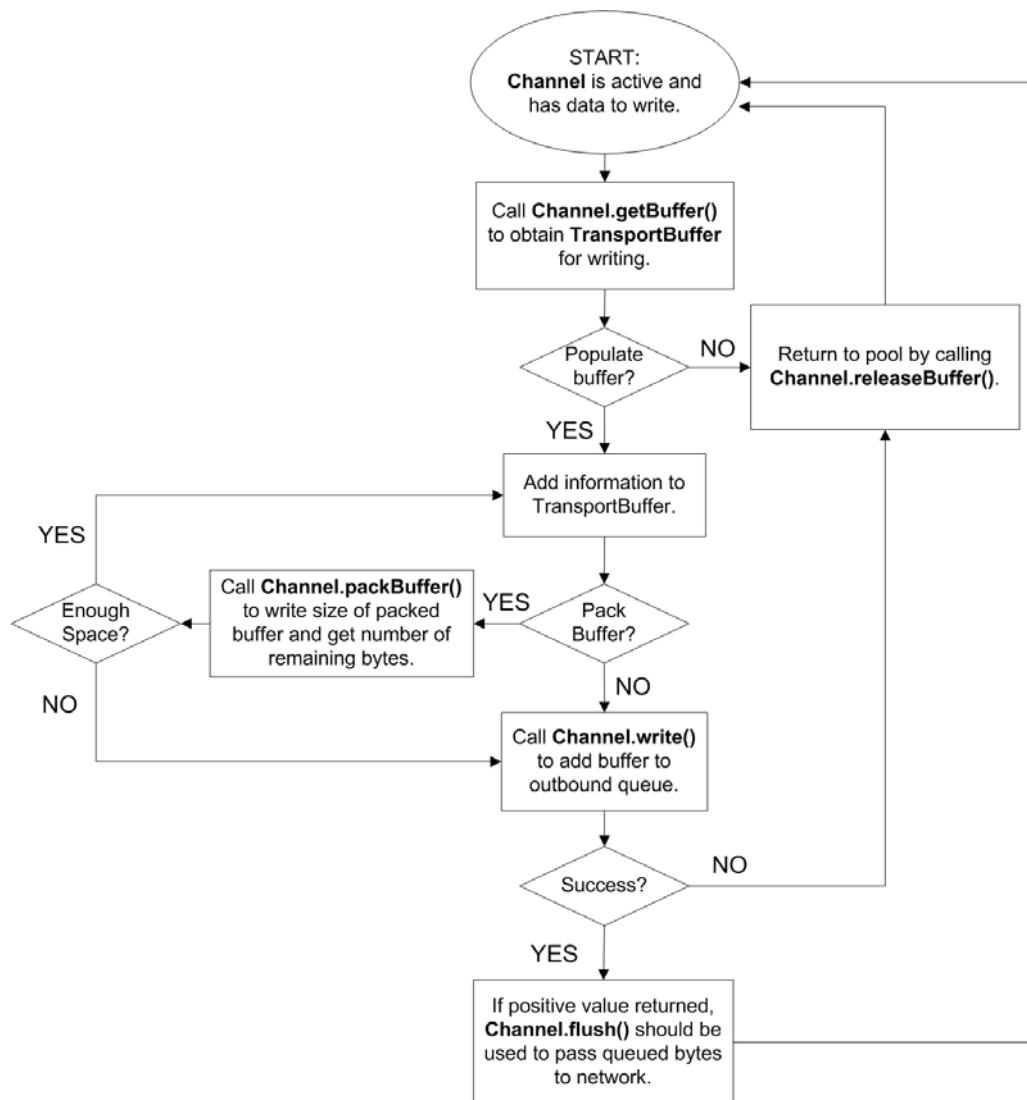


Figure 34. Transport API Writing Flow Chart

9.8 Writing Data: Obtaining a Buffer

To write information, the user must obtain a `TransportBuffer` from the Transport buffer pool. This buffer can originate from the guaranteed output buffer pool associated with the `Channel` or the shared buffer pool associated with the `Server`. A `TransportBuffer` is backed by a Java `ByteBuffer`. After acquiring a buffer, the user can populate the `TransportBuffer` directly by using the `ByteBuffer` reference from the `TransportBuffer.data` method, or by using the Transport API to encode data (refer to Chapter 10, Encoding and Decoding Conventions). If the buffer is not used or the `Channel.write` method call fails, the buffer must be released back into the pool, using `Channel.releaseBuffer`, to ensure proper reuse and cleanup. If the buffer is successfully passed to `Channel.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 `Channel` is configurable through `ConnectOptions` or `BindOptions`. When connecting, the `guaranteedOutputBuffers` setting controls the number of available buffers. When connections are accepted by a `Server`, 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` options, refer to Table 13 and Table 21.

9.8.1 Transport Buffer Management Channel Methods

FUNCTION NAME	DESCRIPTION
getBuffer	<p>Obtains a <code>TransportBuffer</code> of the requested size from the guaranteed or shared buffer pool. After populating the buffer, the <code>length</code> method can be used to get the number of bytes encoded.</p> <p>If the requested size is larger than the <code>maxFragmentSize</code>, the transport will create and return the buffer to the user. When written, this buffer will be fragmented by the <code>Channel.write</code> method (refer to Section 9.9).</p> <p>Because of some additional book keeping required when packing, the application must specify whether a buffer should be 'packable' when calling <code>Channel.getBuffer</code>. For more information on packing, refer to Section 9.11.</p> <p>For performance purposes, an application is not permitted to request a buffer larger than <code>maxFragmentSize</code> and have the buffer be 'packable.'</p> <p>If the buffer is not used or the <code>Channel.write</code> call fails, the buffer must be returned to the pool using <code>Channel.releaseBuffer</code>. If the <code>Channel.write</code> call is successful, the buffer will be returned to the correct pool by the transport.</p> <p>An <code>Error</code> parameter passed into the method conveys error information if a buffer request cannot be satisfied. Return values are described in Table 33.</p> <p>Note: For shared memory connection types (<code>UNIDIR_SHMEM</code>) only one buffer can be obtained at a time. The application must release or write the buffer it has before the application can obtain another buffer.</p>
releaseBuffer	Releases a <code>TransportBuffer</code> back to the correct pool. This should only be called with buffers that originate from <code>Channel.getBuffer</code> and are not successfully passed to <code>channel.write</code> .
bufferUsage	Returns the number of buffers currently in use by the <code>Channel</code> , this includes buffers that the application holds and buffers internally queued and waiting to be flushed to the connection.

Table 31: Buffer Management Channel Methods

9.8.2 Transport Buffer Management Server Method

METHOD	DESCRIPTION
bufferUsage	Returns the number of shared pool buffers currently in use by all channels connected to the Server , this includes shared pool buffers that the application holds and shared pool buffers internally queued and waiting to be flushed.

Table 32: Buffer Management Server Methods

9.8.3 Channel.getBuffer Return Values

The following table defines `TransportReturnCodes` and `Error.errorId` values that can occur while using `Channel.getBuffer`.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case: <code>TransportReturnCodes.SUCCESS</code>	A <code>TransportBuffer</code> is returned to the user. <code>TransportBuffer.data</code> refers to the underlying ByteBuffer.
NULL buffer returned Error Code: <code>TransportReturnCodes.NO_BUFFERS</code>	NULL is returned to the user. This value indicates that there are no buffers available to the user. See <code>Error</code> content for more details. This typically occurs because all available buffers are queued and pending flushing to the connection. The application can use <code>Channel.flush</code> to attempt releasing buffers back to the pool (refer to Section 9.10.2). Additionally, the <code>Channel.ioctl</code> method can be used to increase the number of <code>guaranteedOutputBuffers</code> (refer to Section 9.14).
NULL buffer returned Error Code: <code>TransportReturnCodes.FAILURE</code>	NULL is returned to the user. This value indicates that some type of general failure has occurred. The <code>Channel</code> should be closed, refer to Section 9.13. See <code>Error</code> content for more details.
NULL buffer returned Error Code: <code>TransportReturnCodes.INIT_NOT_INITIALIZED</code>	Indicates that the Transport has not been initialized. See the <code>Error</code> content for more details. For information on initializing, refer to Section 9.2.

Table 33: `Channel.getBuffer` `TransportReturnCodes`

9.9 Writing Data to a Buffer

After a `TransportBuffer` is obtained from `Channel.getBuffer` and populated with the user's data, the buffer can be passed to the `Channel.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 `Channel.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 `TransportBuffer` that has not been successfully passed to `Channel.write` should be released to the pool using `Channel.releaseBuffer`. The following table describes the `Channel.write` method as well as some additional parameters associated with it.

The example in Section 9.9.6 demonstrates the use of `Channel.getBuffer` and `Channel.releaseBuffer`.

9.9.1 Channel.write Method

METHOD	DESCRIPTION
write	<p>Performs any writing or queuing of data. This method expects the <code>channel</code> 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 <code>write</code> method will not return until data was written to the connection or an error has occurred.</p> <p>The <code>WriteArgs</code> parameter passed into the method specifies the <code>WriteFlags</code>, <code>WritePriorities</code>, and conveys the number of bytes written and also uncompressed bytes written. <code>WriteArgs</code> allows for several modifications to be specified for this call. For more information, refer to Section 9.9.2.</p> <p>The Transport supports writing data at different priority levels (for more details, refer to Section 9.10.1).</p> <ul style="list-style-type: none"> • The <code>WriteArgs.uncompressedBytesWritten</code> method returns the number of bytes to be written, including any transport header overhead but not taking into account any compression. • The <code>WriteArgs.bytesWritten</code> method returns the number of bytes to be written, including any transport header overhead and taking into account any compression. • The <code>WriteArgs.seqNum</code> method returns the message's sequence number. <p>If compression is disabled, <code>uncompressedBytesWritten</code> and <code>bytesWritten</code> should match. The number of bytes saved through the compression process can be calculated by <code>(bytesWritten - uncompressedBytesWritten)</code>.</p> <p>Return values are described in Section 9.9.5.</p> <p>Note: Before passing a buffer to <code>Channel.write</code>, 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.</p>

Table 34: Channel.write Function

9.9.2 WriteFlags Values

WRITE FLAG	DESCRIPTION
NO_FLAGS	No modification will be performed to this <code>Channel.write</code> 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 <code>TransportBuffer</code> passed in with this <code>Channel.write</code> call.
DIRECT_SOCKET_WRITE	When set, the <code>Channel.write</code> method will attempt to pass the contents of the <code>TransportBuffer</code> directly to the socket write operation, bypassing any internal Transport API transport queuing. If any information is currently queued, this buffer will also be queued and the <code>Channel.flush</code> 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 <code>Channel.write</code> and <code>Channel.flush</code> indicate that data is not queued.
WRITE_SEQNUM	Indicates that the writer wants to attach a sequence number to this message
WRITE_RETRANSMIT	Indicates that this message is a retransmission of previous content and requires a user-supplied sequence number to indicate which packet is being retransmitted.

Table 35: WriteFlags

9.9.3 Compression

The `Channel.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 9.4.3.

9.9.4 Fragmentation

In addition to compression, the `Channel.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 `Channel`. If the fragmentation cannot fully complete, often due to a shortage of pool buffers, this is indicated by `TransportReturnCodes.WRITE_CALL AGAIN`. In this situation, the application should use `Channel.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 `Channel.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.

Currently, shared memory (`ConnectionTypes.UNIDIR_SHMEM`) connections do not support fragmentation.

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 `channel.write` method.

9.9.5 Channel.write Return Codes

The following table lists all `TransportReturnCodes` that can occur when using the `Channel.write` method.

TRANSPORT RETURN CODE	DESCRIPTION
SUCCESS	<p>Indicates that the <code>Channel.write</code> method was successful and additional bytes have not been internally queued. The <code>Channel.flush</code> method does not need to be called.</p> <p>The application should not release the <code>TransportBuffer</code>; the Transport API will release it.</p>
Any positive value > 0	<p>Indicates that the <code>Channel.write</code> method has succeeded and there is information internally queued by the transport. To pass internally queued information to the connection, the <code>Channel.flush</code> 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 <code>Channel</code> has write availability.</p> <p>The application should not release the <code>TransportBuffer</code>; the Transport API will release it.</p>
WRITE_FLUSH_FAILED	<p>Indicates that the <code>Channel.write</code> 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 <code>Channel.state</code> will transition to <code>ChannelState.CLOSED</code>. If the connection closes, <code>Error</code> information will be populated.</p> <p>The application should not release the <code>TransportBuffer</code>; the Transport API will release it.</p>
WRITE_CALL AGAIN	<p>Indicates that a large buffer could not be fully fragmented with this <code>Channel.write</code> call. This is typically due to all pool buffers being unavailable. An application can use <code>Channel.flush</code> to free up pool buffers or use <code>Channel.ioctl</code> to increase the number of available pool buffers. After pool buffers become available again, the same buffer should be used to call <code>Channel.write</code> 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.</p> <p>If the application does not subsequently pass the <code>TransportBuffer</code> to <code>Channel.write</code>, the buffer should be released by calling <code>Channel.releaseBuffer</code>.</p>
FAILURE	<p>Indicates that a general write failure has occurred. The <code>channel</code> should be closed (refer to Section 9.13). For more details, refer to any <code>Error</code> content.</p> <p>The application should release the <code>TransportBuffer</code> by calling <code>Channel.releaseBuffer</code>.</p>

Table 36: `Channel.write` TransportReturnCodes

TRANSPORT RETURN CODE	DESCRIPTION
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 TransportBuffer was obtained from Channel.getBuffer , the application should release it by calling Channel.releaseBuffer .
INIT_NOT_INITIALIZED	Indicates that the Transport has not been initialized. <ul style="list-style-type: none"> For more details, refer to any Error content. For information on initializing, refer to Section 9.2. The application's attempt to call Channel.getBuffer should have failed for the same reason, so a TransportBuffer should not be present.

Table 36: [Channel.write](#) TransportReturnCodes (Continued)

9.9.6 Channel.getBuffer and Channel.write Example

The following example shows typical use of [Channel.getBuffer](#) and [Channel.write](#). This code would be similar for client or server based [Channels](#).

```
/* Channel.getBuffer() and Channel.write() use, be sure to keep track of the return values from write so
   data is not stranded in the output buffer - Channel.flush() may be required to continue attempting to
   pass data to the connection */
TransportBuffer buffer = null;
EncodeIterator encIter = CodecFactory.createEncodeIterator();
RequestMsg msg = (RequestMsg)CodecFactory.createMsg();
WriteArgs writeArgs = TransportFactory.createWriteArgs();

/* Ask for a 500 byte non-packable buffer to write into */
if ((buffer = chnl.getBuffer(500, false, 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 message encoding information, refer to Section 12.2.9.1 */
    retCode = msg.encode(encIter);

    /* Now write the data - keep track of return code */
    /* this example writes buffer as high priority and no write modification flags */
    writeArgs.priority(WritePriorities.HIGH);
    retCode = chnl.write(buffer, writeArgs, error);

    if (retCode > TransportReturnCodes.SUCCESS)
    {
        /* The write was successful and there is more data queued in the Transport. The flush method
           (discussed in Section 9.10.2) should be used to continue attempting to flush data to the
           connection. UPA will release buffer.*/
    }
    else
    {

```

Code Example 7: Writing Data Using `Channel.write`, `Channel.getBuffer`, and `Channel.releaseBuffer`

9.10 Managing Outbound Queues

Because it may not be possible for the `Channel.write` method to pass all data to the underlying socket, some data may be queued by the Transport. Applications can use the `Channel.flush` method to continue attempting to pass queued data to the connection.

9.10.1 Ordering Queued Data: WritePriorities

Using the `Channel.write` method, an application can associate a priority with each `TransportBuffer`. 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 `Channel.write` first. Only queued data will incur any ordering changes due to priority, and data directly written to the socket by `Channel.write` will not be impacted.

Priority ordering occurs as part of the `Channel.flush` call (refer to Section 9.10.2), 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, `Channel.flush` will move to the next priority in the `priorityFlushStrategy`. The `priorityFlushStrategy` can be changed for each `Channel` by using the `Channel.ioctl` method (refer to Section 9.14).

9.10.1.1 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 `Channel.write` priority values. As the user calls `Channel.write` and assigns priorities to their buffers, they will be queued at the appropriate priority level. As the `Channel.flush` method is called, buffers are removed from the queues in a manner that follows the `priorityFlushStrategy`.

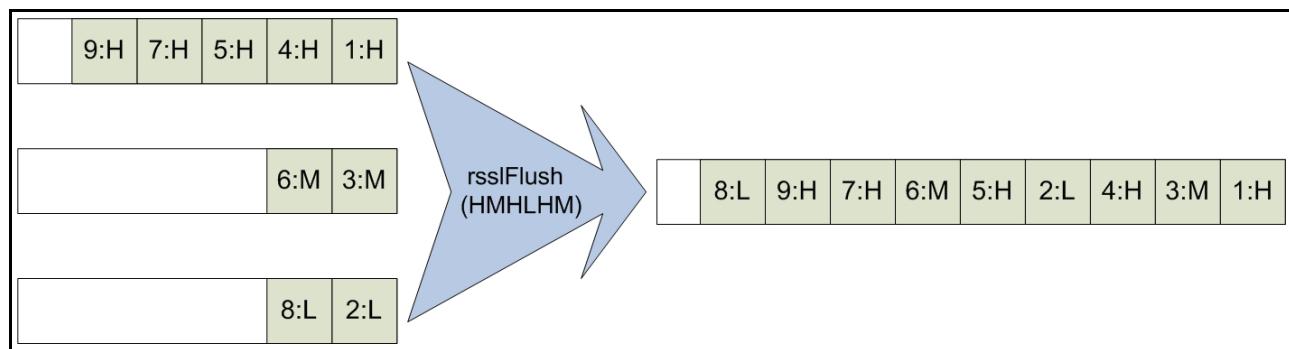


Figure 35. `Channel.write` Priority Scenario

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 `Channel.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 `Channel.write`, so the buffer marked 1 was the first buffer into `Channel.write`, the buffer marked 5 was the 5th buffer into `Channel.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 `Channel.flush` call) in the order specified by the `priorityFlushStrategy`, which by default is HMHLHM. In Figure 35, 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 `Channel.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.

9.10.1.2 WritePriorities Values

PRIORITYVALUE	MEANING
HIGH	If not directly written to the socket, this <code>TransportBuffer</code> will be flushed at the high priority.
MEDIUM	If not directly written to the socket, this <code>TransportBuffer</code> will be flushed at the medium priority.
LOW	If not directly written to the socket, this <code>TransportBuffer</code> will be flushed at the low priority.

Table 37: WritePriorities Values

9.10.2 Channel.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 `Channel.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 `Channel.flush` until all bytes have been written.

FUNCTION NAME	DESCRIPTION
Channel.flush	<p>Writes queued data to the connection. This method expects the <code>Channel</code> to be in the active state. If data is not queued, the <code>Channel.flush</code> method is not required and should return immediately.</p> <p>This method performs any buffer reordering that might occur due to priorities passed in on the <code>Channel.write</code> method. For more information about priority writing, refer to Section 9.10.1.</p> <p>Return values are described in Table 39.</p>

Table 38: Channel . flush Method

9.10.3 Channel.flush Return Codes

The following table defines the return `TransportReturnCodes` that can occur when using `Channel.flush`.

RETURN CODE	DESCRIPTION
SUCCESS	Indicates that the <code>Channel.flush</code> method has succeeded and additional bytes are not internally queued. The <code>Channel.flush</code> method need not be called.
Any positive value > 0	Indicates that the <code>Channel.flush</code> method has succeeded, however data is still internally queued by the transport. The <code>Channel.flush</code> 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 <code>SelectableChannel</code> has write availability.
FAILURE	Indicates that a general failure has occurred, often because the underlying connection is unavailable or closed. The <code>Channel</code> should be closed (refer to Section 9.13). For more details, refer to the <code>Error</code> content.
INIT_NOT_INITIALIZED	Indicates that the Transport is not initialized. For more details, refer to the <code>Error</code> content. For information on initializing, refer to Section 9.2.

Table 39: Channel . flush TransportReturnCodes

9.10.4 Channel.flush Example

The following example shows typical use of `channel.flush`. This example assumes the use of an I/O notification mechanism. This code would be similar for client- or server-based `ChannelS`.

```

/* 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 the
   connection */

/* Assuming this section of code was called because of a write selector notification */
if ((retCode = chnl.flush(error)) > TransportReturnCodes.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 TransportReturnCodes.SUCCESS:
            /* Everything has been flushed, no data is left to send - unset write notification */
            SelectionKey key = chnl.SelectableChannel().keyFor(selector);
            try
            {
                chnl.SelectableChannel().register(selector, key.interestOps() - SelectionKey.OP_WRITE,
                                                chnl);
            }
            catch (Exception e)
            {
                System.out.println("\nregister select Exception: " + e.getMessage());
            }
            break;
        case TransportReturnCodes.INIT_NOT_INITIALIZED:
        case TransportReturnCodes.FAILURE:
            System.out.printf("Error (%d) (errno: %d) encountered with flush. Error Text: %s\n",
                             error.errorId(), error.sysError(), error.text());
            /* Connection should be closed, return failure */
            break;
        default:
            System.out.printf("Unexpected return code (%d) encountered!", retCode);
            /* Likely unrecoverable, connection should be closed */
    }
}
}

```

Code Example 8: Channel . flush Use

9.11 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 `Channel.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 `Channel.packBuffer` method packs multiple messages into one `TransportBuffer`.

METHOD	DESCRIPTION
Channel.packBuffer	Packs the contents of a passed-in <code>TransportBuffer</code> and returns the number of bytes remaining in the <code>TransportBuffer</code> . 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 <code>channel.getBuffer</code> as 'packable' and cannot exceed the <code>maxFragmentSize</code> . Packing is not supported for shared memory (<code>ConnectionTypes.UNIDIR_SHMEM</code>) connections.

Table 40: `Channel . packBuffer` Method

9.11.1 Channel.packBuffer Return Values

The following table defines return and error code values that can occur when using `Channel.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 <code>Channel</code> should be closed (refer to Section 9.13). For more details, refer to <code>Error</code> content.

Table 41: `Channel . packBuffer` Return Values

9.11.2 Example: Channel.getBuffer, Channel.packBuffer, and Channel.write

The following example shows typical use of `Channel.getBuffer`, `Channel.packBuffer`, and `Channel.write`. This code is similar for client- or server-based `Channel` structures.

```
/* Channel.getBuffer(), Channel.packBuffer() and Channel.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 */
TransportBuffer buffer = null;
EncodeIterator encIter = CodecFactory.createEncodeIterator();
RequestMsg msg = (RequestMsg)CodecFactory.createMsg();
```

```

WriteArgs writeArgs = TransportFactory.createWriteArgs();

/* Ask for a 6000 byte packable buffer to write multiple messages into */
if ((buffer = chnl.getBuffer(6000, true, 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 12.2.9.1 */
    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, error)) < TransportReturnCodes.SUCCESS)
    {
        System.out.printf("Error (%d) (errno: %d) encountered with packBuffer. Error Text: %s\n",
                          error.errorId(), error.sysError(), error.text());
        /* Buffer must be released - return code from releaseBuffer can be checked */
        chnl.releaseBuffer(buffer, 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 12.2.9.1 */
    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, error)) < TransportReturnCodes.SUCCESS)
    {
        System.out.printf("Error (%d) (errno: %d) encountered with packBuffer. Error Text: %s\n",
                          error.errorId(), error.sysError(), error.text());
        /* Buffer must be released - return code from releaseBuffer can be checked */
        chnl.releaseBuffer(buffer, 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, error);
}

```

```

    /* For a full, write error-handling example, refer to the Example in Section Section 9.9.6. */
}
else
{
    /* Use the flush method (Section 9.10.2) to free buffers back to the pool */
}

```

Code Example 9: Message Packing Using `Channel.packBuffer`

9.12 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 an OMM 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.

9.12.1 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 9.3.2.1 and Section 9.4.1.1. 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 `Channel.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 = clientPingTimeout;
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 10: Ping Negotiation Calculation

9.12.2 Channel.ping Function

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 `Channel.read` returns `TransportReturnCodes.READ_PING` (refer to Section 9.6), 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 `Channel.pingTimeout` value, the connection should be closed.

The following table describes the `Channel.ping` method, used to send heartbeat messages.

METHOD	DESCRIPTION
Channel.ping	<p>Attempts to write a heartbeat message on the connection. This method expects an active <code>Channel</code>.</p> <p>If an application calls the <code>Channel.ping</code> 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.</p> <p>When using a shared memory (<code>UNIDIR_SHMEM</code>) connection type, pings can only be sent from server to client.</p> <p>Return values are described in Table 43.</p>

Table 42: `Channel.ping` method

9.12.3 Channel.ping Return Values

The following table defines the `TransportReturnCodes` that can occur when using `Channel.ping`.

TRANSPORT RETURN CODE	DESCRIPTION
SUCCESS	Indicates that the <code>Channel.ping</code> 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 <code>Channel.flush</code> 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 <code>socketId</code> has write availability.
FAILURE	This value indicates that some type of general failure has occurred. The <code>Channel</code> should be closed (refer to Section 9.13). For more details, refer to the <code>Error</code> content.

Table 43: `Channel.ping` `TransportReturnCodes`

9.12.4 Channel.ping Example

The following example shows typical use of `Channel.ping`. This example assumes the use of some kind of timer mechanism to execute when necessary. This code would be similar for client or server based `Channels`.

```
/* Channel.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(error)) > TransportReturnCodes.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 TransportReturnCodes.SUCCESS:
            /* Ping message has been sent successfully */
            break;
        case TransportReturnCodes.INIT_NOT_INITIALIZED:
        case TransportReturnCodes.FAILURE:
            System.out.printf("Error (%d) (errno: %d) encountered with ping. Error Text: %s\n",
                error.errorId(), error.sysError(), error.text());
            /* Connection should be closed, return failure */
            break;
        default:
            System.out.printf("Unexpected return code (%d) encountered!", retCode);
            /* Likely unrecoverable, connection should be closed */
    }
}
```

Code Example 11: Channel . ping Use

9.13 Closing Connections

9.13.1 Functions for Closing Connections

When an error occurs on a connection or a `Channel` is being disconnected, the `Channel.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 `Channel.getBuffer`, they should be released using `Channel.releaseBuffer` prior to closing the channel.

If a server is being shut down, use the `Server.close` method to close the listening socket and perform any necessary cleanup. All currently connected `Channels` 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 `Channel.close` to shut down any currently connected applications.

METHOD	DESCRIPTION
<code>Channel.close</code>	Closes a client- or server-based <code>Channel</code> . 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 <code>Channel.close</code> .
<code>Server.close</code>	Closes a listening socket associated with a <code>Server</code> . <code>Server.close</code> 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 <code>Channel.close</code> to shutdown any remaining connections.

Table 44: Connection Closing Functionality

9.13.2 Close Connections Example

The following example shows typical use of `Channel.close` and `Server.close`.

```
/* Channel.close() */
if (chnl.close(error) < TransportReturnCodes.SUCCESS)
{
    System.out.printf("Error (%d) (errno: %d) encountered with channel close. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

/* Server.close() */
if (srvr.close(error) < TransportReturnCodes.SUCCESS)
{
    System.out.printf("Error (%d) (errno: %d) encountered with server close. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}
```

Code Example 12: Closing a Connection Using `Channel.close` and `Server.close`

9.14 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 **Channel** or **Server** parameters during run-time. These methods are described in the following tables.

9.14.1 General Transport Utility Methods

METHOD NAME	DESCRIPTION
Channel.info	Allows the application to query Channel negotiated parameters and settings and retrieve all current settings. This includes maxFragmentSize and negotiated compression information as well as many other values. This populates a ChannelInfo object. For a full list of available settings, refer to Section 9.14.2.
Server.info	Allows the application to query Server related values, such as current and peak shared pool buffer usage statistics. This populates a ServerInfo object, defined in Section 9.14.5.
Channel.ioctl	Allows the application to change various settings associated with the Channel . Available IoctlCodes are defined in Section 9.14.6.
Server.ioctl	Allows the application to change various settings associated with the Server . Available IoctlCodes are defined in Section 9.14.7.
Transport.hostByName	Takes a Java String populated with a hostname, looks up and returns a Java InetSocketAddress .
Transport.userName	Queries the username associated with the owner of the current process, and returns a String .

Table 45: Transport Utility Methods

9.14.2 ChannelInfo Methods

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

METHOD	DESCRIPTION
maxFragmentSize	<p>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 <code>maxFragmentSize</code>.</p> <p>This is the largest size a user can request while still being ‘packable.’</p>
numInputBuffers	<p>The number of sequential input buffers into which the <code>Channel</code> 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 <code>maxFragmentSize</code> bytes. Input buffers are allocated at initialization time.</p>
guaranteedOutputBuffers	<p>The guaranteed number of buffers which this <code>Channel</code> can use while writing data. Each buffer can contain <code>maxFragmentSize</code> bytes. Guaranteed output buffers are allocated at initialization time. For more details on obtaining a buffer, refer to Section 9.8.</p> <p>You can configure <code>guaranteedOutputBuffers</code> using <code>Channel.ioctl</code>, as described in Section 9.14.6.</p>
maxOutputBuffers	<p>The maximum number of output buffers which this <code>Channel</code> can use. (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) is equal to the number of shared pool buffers that this <code>Channel</code> can use. Shared pool buffers are only used if all <code>guaranteedOutputBuffers</code> are unavailable. If <code>maxOutputBuffers</code> is equal to the <code>guaranteedOutputBuffers</code> value, shared pool buffers are unavailable.</p> <p>You can configure <code>maxOutputBuffers</code> using <code>Channel.ioctl</code>, as described in Section 9.14.6.</p>
pingTimeout	<p>The negotiated ping timeout value. Typically, the rule of thumb in handling heartbeats is to send a heartbeat every <code>pingTimeout</code>/3 seconds.</p> <p>For more details on <code>pingTimeout</code>, refer to Section 9.12.1.</p>
serverToClientPings	<p>Sets whether server is expected to send heartbeat messages:</p> <ul style="list-style-type: none"> If set to <code>true</code>, heartbeat messages must flow from server to client. If set to <code>false</code>, the server is not required to send heartbeats. <p>TREP and other Thomson Reuters components typically require this value to be set to <code>true</code>.</p>
clientToServerPings	<p>Sets whether the client is expected to send heartbeat messages:</p> <ul style="list-style-type: none"> If set to <code>true</code>, heartbeat messages must flow from client to server. If set to <code>false</code>, the client is not required to send heartbeats. <p>TREP and other Thomson Reuters components typically require this value to be set to <code>true</code>.</p>
sysSendBufSize	<p>Sets the size of the send or output buffer associated with the underlying transport. The Transport has additional output buffers, controlled by <code>maxOutputBuffers</code> and <code>guaranteedOutputBuffers</code>. For some connection types, you can configure <code>sysSendBufSize</code> using <code>Channel.ioctl</code>, as described in Section 9.14.6.</p>
sysRecvBufSize	<p>Sets the size of the receive or input buffer associated with the underlying transport. The Transport has an additional input buffer controlled by <code>numInputBuffers</code>.</p> <p>For some connection types, you can configure <code>sysRecvBufSize</code> using <code>Channel.ioctl</code>, as described in Section 9.14.6.</p>

Table 46: `Channel.info` Methods

METHOD	DESCRIPTION
compressionType	Sets the type of compression to use on this connection. Refer to Section 9.4.3 for more information about supported compression types.
compressionThreshold	Sets the compression threshold. Messages smaller than the threshold are not compressed; messages larger than the threshold are compressed.
priorityFlushStrategy ^a	The currently priority level order used when flushing buffers to the connection, where H = High priority, M = Medium priority, and L = Low priority. When passed to <code>Channel.write</code> , each buffer is associated with the priority level at which it should be written. The default <code>priorityFlushStrategy</code> 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 <code>priorityFlushStrategy</code> using <code>Channel.ioctl</code> , as described in Section 9.14.6.
clientHostname	The host name of the connecting client. Valid only for <code>Channels</code> that were accepted (i.e., returned from <code>Transport.accept</code>) and whose <code>ChannelState</code> is <code>ACTIVE</code> .
clientIP	The IP address of the connecting client. Valid only for Channels that were accepted (i.e., returned from <code>Transport.accept</code>) and whose <code>ChannelState</code> is <code>ACTIVE</code> .
multicastStats	If using a connection type of <code>ConnectionTypes.RELIABLE_MCAST</code> , this substructure reports information about sent and received packets, including any gap or retransmission information. For details on options used with <code>multicastStats</code> , refer to Section 9.14.3.
componentInfo	Retrieves a Java <code>List</code> of <code>ComponentInfos</code> . One <code>ComponentInfo</code> object will be present for each connected device that supports connected component versioning. For more detailed information on the <code>ComponentInfo</code> structure, refer to Section 9.14.4.

Table 46: Channel Info Methods (Continued)

a. Allows for up to 32 one-byte characters to be represented. 'H' = high priority, 'M' = medium priority, and 'L' = low priority.

9.14.3 multicastStats Methods

METHOD	DESCRIPTION
mcastSent	The number of multicast packets sent by this <code>Channel</code> .
mcastRcvd	The number of multicast packets received by this <code>Channel</code> .
unicastSent	The number of unicast UDP packets sent by this <code>Channel</code> .
unicastRcvd	The number of unicast UDP packets received by this <code>Channel</code> .
retransReqSent	The number of retransmission requests sent by this <code>Channel</code> . Retransmission requests are sent in an attempt to recover a missed packet and may indicate a network problem if gaps are also detected. This is populated only for reliable multicast type connections.
retransReqRcvd	This is the number of retransmission requests received by this <code>channel</code> . Retransmission requests are received if another component on the network missed a packet sent by this channel and may indicate a network problem if gaps are also being detected. This is populated only for reliable multicast type connections.
retransPktsSent	The number of retransmitted packets sent by this <code>Channel</code> . Packets are retransmitted in response to retransmission requests. If a packet cannot be retransmitted, this results in a gap occurring and indicates a network problem, which applications are notified of via <code>Channel.read</code> . This is populated only for reliable multicast type connections.
retransPktsRcvd	The number of retransmitted packets received by this <code>channel</code> . This is populated only for reliable multicast type connections.
gapsDetected	Returns a count of the number of detected packet gaps detected and reported to the application. This is a result of packet loss on the network and may indicate a more serious network problem.

Table 47: `multicastStats` Methods

9.14.4 componentInfo Method

METHOD	DESCRIPTION
componentVersion	A <code>TransportBuffer</code> containing an ASCII string that indicates the product version of the connected component.

Table 48: `componentInfo` Options

9.14.5 ServerInfo Methods

The following table describes values available to the user through the use of the `Server.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 <code>Server</code> .
peakBufferUsage	The maximum achieved number of used shared pool buffers across all users connected to the <code>Server</code> . This value can be reset through the use of <code>Server.ioctl</code> , as described in Section 9.14.7.
Clear	Clears this object, so that it can be reused.

Table 49: `ServerInfo` Methods

9.14.6 Channel.ioctl IoctlCodes

The following table provides a description of the ioctlCodes available for use with the `Channel.ioctl` method.

OPTION ENUMERATION	DESCRIPTION
MAX_NUM_BUFFERS	Allows a <code>Channel</code> to change its <code>maxOutputBuffers</code> setting. Value is an <code>int</code> .
NUM_GUARANTEED_BUFFERS	Allows a <code>Channel</code> to change its <code>guaranteedOutputBuffers</code> setting. Value is an <code>int</code> .
HIGH_WATER_MARK	Allows a <code>Channel</code> to change the internal Transport API output queue depth water mark, which has a default value of 6,144 bytes. When the Transport API output queue exceeds this number of bytes, the <code>Channel.write</code> method internally attempts to flush content to the network. Value is an <code>int</code> .
SYSTEM_READ_BUFFERS	Allows a <code>Channel</code> to change the TCP receive buffer size associated with the connection. Value is an <code>int</code> , which defaults to 64 (KB). If the value is larger than 64KB, the value needs to be specified before the socket connects to the remote peer. <ul style="list-style-type: none"> For servers and <code>SYSTEM_READ_BUFFERS</code> larger than 64 KB, use <code>BindOptions.sysRecvBufferSize</code> to set the receive buffer size prior to calling <code>Transport.bind</code>. For clients and <code>SYSTEM_READ_BUFFERS</code> larger than 64 KB, use <code>ConnectOptions.sysRecvBufferSize</code> to set the receive buffer size prior to calling <code>Transport.connect</code>.
SYSTEM_WRITE_BUFFERS	Allows a <code>Channel</code> to change the TCP send buffer size associated with the connection. Value is an <code>int</code> .
COMPRESSION_THRESHOLD	Allows a <code>Channel</code> to change the size (in bytes) at which buffer compression occurs, must be greater than 30 bytes. Value is an <code>int</code> .
PRIORITY_FLUSH_ORDER	Allows a <code>Channel</code> to change its <code>priorityFlushStrategy</code> . Value is a <code>string</code> , where each character is either: <ul style="list-style-type: none"> H for high priority M for medium priority L for low priority The <code>string</code> should not exceed 32 entries. 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.

Table 50: `Channel.ioctl` `IoctlCodes`

9.14.7 Server.ioctl IoctlCodes

The following table provides a description of the `IoctlCodes` available for use with the `server.ioctl` method.

IOCTL CODE	DESCRIPTION
SERVER_NUM_POOL_BUFFERS	Allows a <code>Server</code> to change its <code>sharedPoolSize</code> setting. Value is an <code>int</code> .
SERVER_PEAK_BUF_RESET	Allows a <code>Server</code> to reset the <code>peakBufferUsage</code> statistic. Value is not required.

Table 51: `Server.ioctl` `IoctlCodes`

9.15 Tunneling

Consumer applications can establish Internet connections via HTTP and HTTPS tunneling. This functionality is supported across all platforms.

9.15.1 Configuration

An HTTP tunneling connection uses a connection type of `ConnectionTypes.HTTP`, while an HTTPS tunneling connection uses a connection type of `ConnectionTypes.ENCRYPTED`. Additional configuration is required on an HTTPS tunneling connection, which can be specified using the `TunnelingInfo` method.

A consumer application needs to configure additional parameters, in addition to `tunnelingType`. By setting the `HTTPproxy` configuration parameter to `true`, the application will be connected via a proxy. The configuration parameters `HTTPproxyHostname` and `HTTPproxyPort` specify the proxy host name and port. A client connection that leverages connection type `ConnectionTypes.HTTP` or `ConnectionTypes.ENCRYPTED` might be connecting through proxy devices as it tunnels through the Internet.

If a consumer application uses the connection configured for connection type `ConnectionTypes.ENCRYPTED`, additional configuration parameters apply. The parameters specify security settings, such as: `KeystoreType`, `KeystoreFile`, `KeystorePasswd`, `SecurityProvider`, `KeyManagerAlgorithm`, `TrustManagerAlgorithm`. The Transport API uses JDK `java.security` package. If the parameters `KeystoreType`, `SecurityProvider`, `KeyManagerAlgorithm`, and `TrustManagerAlgorithm` are not specified, the JDK `java.security` package provides default settings.

9.15.1.1 TunnelingInfo Methods

METHOD	DESCRIPTION
tunnelingType	The Tunneling type. Possible values are "None", "http", or "encrypted". For HTTP Tunneling, <code>tunnelingType</code> has to be set to <code>http</code> or <code>encrypted</code> .
HTTPproxy	Set whether the tunneling application is going through an HTTP proxy server.
HTTPproxyHostName	Configures the address or hostname of the HTTP proxy server to connect to. <code>HTTPproxy</code> has to be <code>true</code> .
HTTPproxyPort	Configures the Port Number of the HTTP proxy server to which the consumer application connects. If you configure this method, <code>HTTPproxy</code> must also be set to <code>true</code> .
objectName	Configures the object name for load balancing to the various ADSs as part of a hosted solution.
KeystoreType	Configures the type of <code>keystore</code> for the certificate file. Defaults to the property <code>keystore.type</code> in the JDK security properties file (<code>java.security</code>). The Oracle JDK default is <code>JKS</code> .
KeystoreFile	Configures the <code>keystore</code> file that contains your own private keys and public key certificates you received from someone else.
KeystorePasswd	Configures the password for the <code>keystore</code> file.
SecurityProtocol	The cryptographic protocol used (the Oracle JDK default is <code>TLS</code>). Available options are: <ul style="list-style-type: none"> • TLSv1.2 • TLSv1.1 • TLSv1 • TLS • SSLv3 • SSLv2 • SSL

Table 52: TunnelingInfo Methods

METHOD	DESCRIPTION
SecurityProvider	The Java Cryptography Package provider used. The Oracle JDK default is SunJSSE .
KeyManagerAlgorithm	The Java Key Management algorithm. Defaults to the property ssl.KeyManagerFactory algorithm in the JDK security properties file (java.security). The Oracle JDK is SunJX509 .
TrustManagerAlgorithm	The Java Trust Management algorithm. Defaults to the property ssl.TrustManagerFactory.algorithm in the JDK security properties file (java.security). The Oracle JDK default is PKIX .

Table 52: TunnelingInfo Methods (Continued)

9.15.1.2 Configuration Example

The following procedure describes how to provide the required authentication credentials to the Transport API. The following procedure illustrates how to modify the **Consumer** example.

1. Open **Consumer.java** located in **Examples/com/thomsonreuters/upa/examples/consumer**.
2. For a connection type of **ConnectionTypes.ENCRYPTED**, edit the following code in the **setEncryptedConfiguration** method with the proxy server hostname and port to which you will connect:

```
options.tunnelingInfo().HTTPproxyHostName( "myProxy" );
options.tunnelingInfo().HTTPproxyPort(8443);
```

3. In the **setEncryptedConfiguration** method, edit the following code for the Keystore file and its password:

```
options.tunnelingInfo().KeystoreFile("myKeystore.jks");
options.tunnelingInfo().KeystorePasswd("myKeystorePasswd");
```

4. For a connection type of **ConnectionTypes.HTTP**, edit the following code in the **setHTTPconfiguration** method with the proxy server hostname and port to which you will connect.

```
options.tunnelingInfo().HTTPproxyHostName( "myProxy" );
options.tunnelingInfo().HTTPproxyPort(8080);
```

9.15.2 Proxy Authentication

You can configure some proxy servers to authenticate client applications before they pass through the proxy to their destination. The Transport API supports Negotiate(Kerberos), Kerberos, NTLM, and Basic authentication schemes.

Note: A consumer application needing NTLM authentication should add the Apache jar files (in the load's **Libs/ApacheClient** directory) to the **CLASSPATH**.

Authentication schemes:

- 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.

This section:

- Provides an overview of the proxy authentication process.
- Details how to programmatically supply consumer applications with the user credentials required to authenticate with a proxy.
- Provides tips for troubleshooting proxy authentication failures.

9.15.2.1 The Proxy Authentication Process

If a Transport API consumer’s connection is configured to connect to a provider via a proxy server (using HTTP or Encrypted tunneling) which requires authentication, the Transport API will automatically participate in the authentication process. The application must supply Transport API-valid credentials (described in Section 9.15.2.2). Specifically, the Transport API automatically parses the list of supported authentication schemes (sent by the proxy), selects the most appropriate scheme, re-connects to the proxy (if necessary), and exchanges the messages required by the selected authentication scheme.

A typical proxy authentication adheres to the following process:

- The consumer application uses either HTTP or HTTPS protocol to connect to a provider (e.g., an ADS, or a Transport API C provider application, not shown) via a proxy server.
- Because authentication is enabled on the proxy server, the proxy server sends an HTTP response to the application indicating authentication is required. This response contains the HTTP error code# 407, and includes a list of authentication schemes enabled on the proxy.

The initial response sent from the proxy server may also indicate that the connection between it and the consumer application will be closed.

- If the Transport API supports at least one of the authentication schemes specified in the list sent by the proxy server, it reconnects to the proxy (if necessary) and sends a message containing:
 - The name of the authentication scheme it will use.
 - The user credentials (e.g. a username and a password) encoded in the format prescribed by the authentication scheme.

- The proxy server attempts to authenticate the user credentials provided by the application. Depending on the configuration of the proxy server and the authentication scheme, the proxy server may attempt to authenticate the credentials against an LDAP server, a Microsoft ActiveDirectory™ server, or its own credentials datastore.
 - If the authentication scheme requires only that the application send a single message containing the user credentials (e.g., the BASIC authentication scheme), and the proxy server was able to successfully authenticate these credentials, then the proxy sends a response to the Transport API with the HTTP “OK” error code# 200, indicating a successful authentication.
 - If the authentication scheme is NTLM (illustrated in Section 9.15.2.5), then the authentication process requires a negotiation (i.e., multiple messages sent back and forth). After the initial message, the proxy server again sends a message to the Transport API containing HTTP error code #407, and (typically) additional handshaking details to be processed by the application. The Transport API uses this information to send a message back to the proxy server to continue the authentication process until authentication ultimately succeeds (or fails). When successful, the proxy sends a response to the Transport API with HTTP “OK” error code# 200.
 - If the authentication scheme is Negotiate/Kerberos, (illustrated in Section 9.15.2.6) then the authentication process requires additional handshaking with a Domain Controller. After the proxy server sends a message to the Transport API containing HTTP error code #407, the Transport API does all the necessary Kerberos handshaking with the Domain Controller (which for Kerberos is the Key Distribution Center or KDC) to obtain the needed Kerberos service ticket. The Transport API uses this ticket to authenticate. When successful, the proxy sends a response to the Transport API with HTTP “OK” error code #200.
- If authentication fails, the proxy server sends a response with an HTTP error code and text/HTML indicating the failure.

9.15.2.2 Supplying the Transport API with Credentials for Proxy Authentication

When the Transport API connects to a proxy server that requires authentication, the proxy server sends a response to the Transport API containing HTTP error code# 407, indicating that authentication is required and includes a list of authentication schemes enabled on the proxy server. Authentication schemes are listed in order from ‘most secure’ (i.e., Negotiate/Kerberos) to ‘least secure’ (i.e., Basic). The Transport API attempts to use the first provided authentication scheme (i.e., Negotiate) and if that fails, the Transport API attempts to use the next authentication scheme (and so on) in the order provided. So in Code Example 13, the order of attempted authentication schemes is: Negotiate(Kerberos) -> Kerberos -> NTLM -> Basic.

All authentication schemes require a username and a password during the authentication process. Negotiate, Kerberos, and NTLM also include additional details while authenticating (described in Section 9.15.2.3).

The following sample “407” response includes a highlighted list of authentication schemes enabled on the proxy:

```
HTTP/1.1 407 Proxy Authentication Required (Forefront TMG requires authorization to fulfill the
request. Access to the Web Proxy filter is denied.)
Via: 1.1 OAKLPC101
Proxy-Authenticate: Negotiate
Proxy-Authenticate: Kerberos
Proxy-Authenticate: NTLM
Proxy-Authenticate: Basic realm="hostname.ntdomain.company.com"
Connection: close
Proxy-Connection: close
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 726
```

Code Example 13: Sample 407 Proxy Response Listing the Authentication Schemes Enabled on the Proxy

9.15.2.3 CredentialInfo Methods

METHOD	DESCRIPTION
HTTPproxyUsername	The username to authenticate. Needed for all authentication protocols.
HTTPproxyPasswd	The password to authenticate. Needed for all authentication protocols.
HTTPproxyDomain	The domain of the user to authenticate. Needed for NTLM, Negotiate/Kerberos, or Kerberos authentication protocols. For Negotiate/Kerberos or Kerberos authentication protocols, HTTPproxyDomain should be the same as the domain in the 'realms' and 'domain_realm' sections of the Kerberos configuration file, see the HTTPproxyKRB5configFile .
HTTPproxyLocalHostname	The local hostname of the client. Needed for NTLM authentication protocol only.
HTTPproxyKRB5ConfigFile	The complete path of the Kerberos5 configuration file (krb5.ini , krb5.conf , or some other custom file). Needed for Negotiate/Kerberos and Kerberos authentications.

Table 53: CredentialInfo Methods

9.15.2.4 Providing Credentials and Modifying the Consumer Example

The following procedure describes how to provide the required authentication credentials to the Transport API and how to modify the **Consumer** example.

Open **Consumer.java** located in [Examples/com/thomsonreuters/upa/examples/consumer](#).

For a connection type of **ConnectionTypes.ENCRYPTED**, edit in method **setCredentials** the following code with the username, password, and domain you will use:

```
options.credentialsInfo().HTTPproxyUsername("firstName.lastName");
options.credentialsInfo().HTTPproxyPasswd("myPasswd");
options.credentialsInfo().HTTPproxyDomain("myDomain");
```

Also in method **setCredentials** you may need to change the Kerberos configuration file location:

```
options.credentialsInfo().HTTPproxyKRB5configFile("C:\\WINDOWS\\krb5.ini");
```

9.15.2.5 Proxy Authentication using NTLM

The following diagram illustrates proxy authentication between a Transport API consumer and a proxy server using Windows Authentication (i.e., NTLM).

Note: In the following diagram, UPA is synonymous with the Transport API.

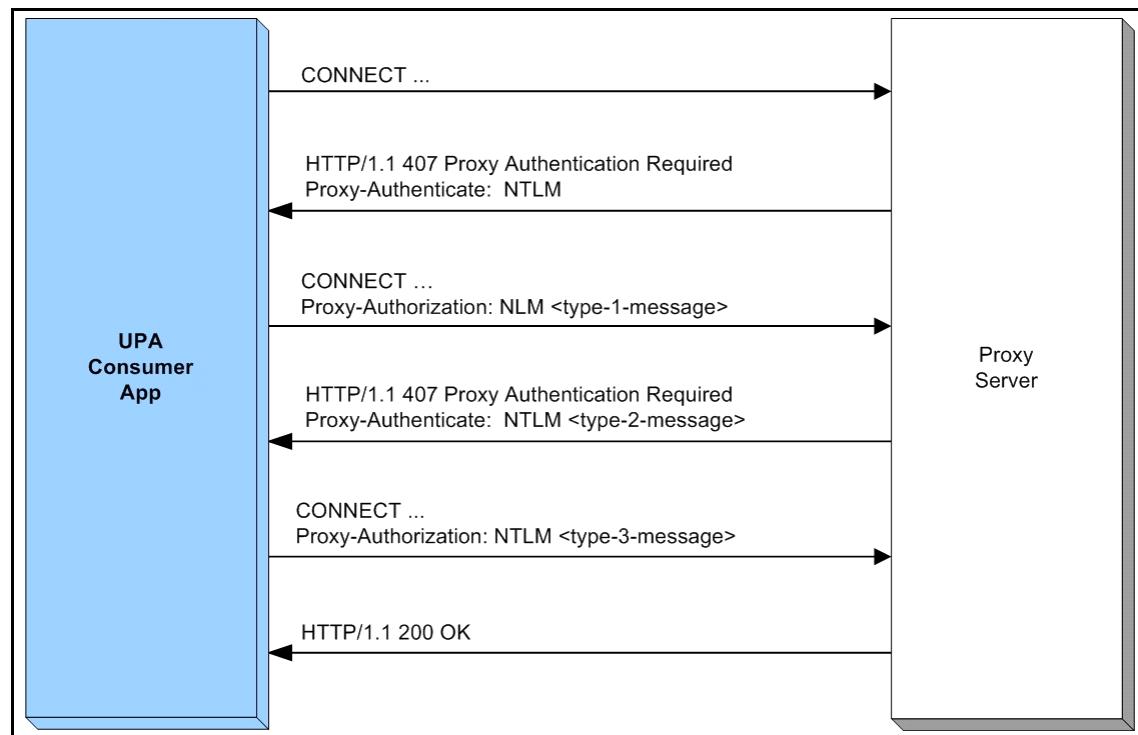


Figure 36. Transport API Consumer Application authenticating with a Proxy Server using NTLM

9.15.2.6 Proxy Authentication using Negotiate/Kerberos

In the following diagram, from the perspective of the consumer application, the only additional “work” required to support proxy authentication is to programmatically supply the Transport API with the credentials required for authentication.

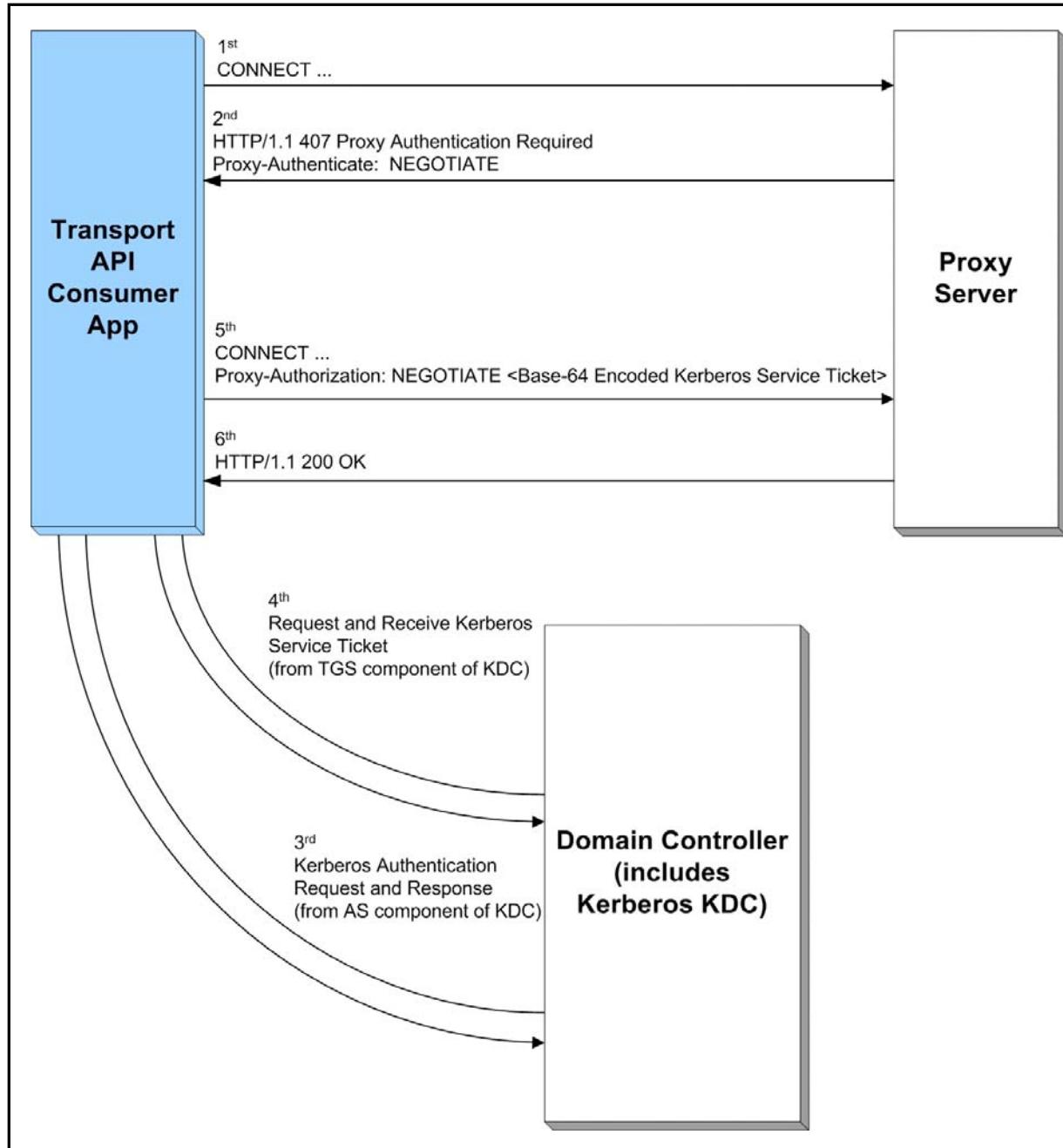


Figure 37. Transport API Consumer Application Authenticating with a Proxy Server using Negotiate/Kerberos

Chapter 10 Encoding and Decoding Conventions

10.1 Concepts

The 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 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 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, Thomson Reuters 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.

You can use the Codec package with the Transport package and user-defined transports:

- To use the Codec package with the Transport package, obtain `TransportBuffers` from the Transport package and encode and/or decode using the Codec package.
- To use the Codec package with user-defined transports, obtain `Buffers` from the Codec package, and encode and/or decode using the Codec package.

The rest of this chapter refers to `TransportBuffers`, unless `Buffers` are explicitly specified. However, `Buffers` can be used wherever `transportBuffers` are specified.

10.1.1 Data Types

The 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 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 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.

10.1.2 Composite Pattern and Nesting

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

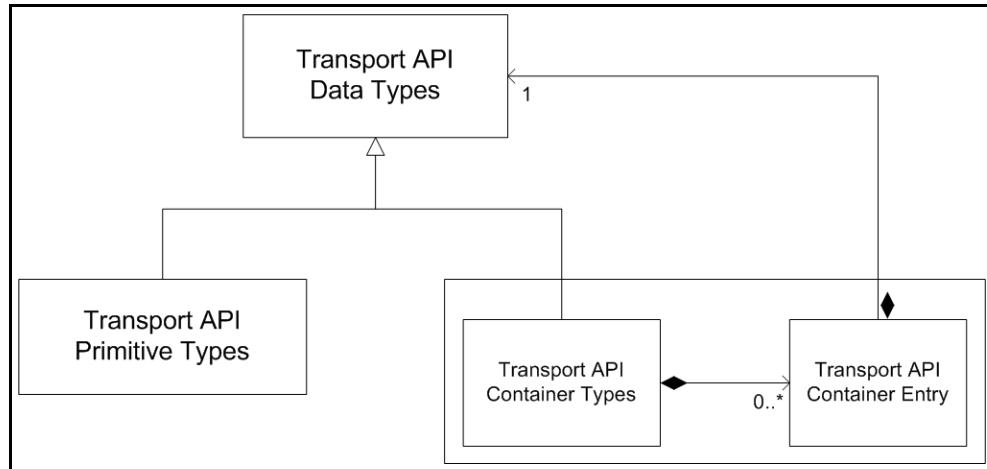


Figure 38. Transport API and the Composite Pattern

The diagram highlights the following:

- Being made up of both primitive and container types, Transport API data type values mirror the composite pattern's component.
- Primitive types mimic the composite pattern's leaf, conveying concrete information for the user.
- The 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.

10.2 Encoding Semantics

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

10.2.1 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).

10.2.2 The Encode Iterator: `EncodeIterator`

To encode content you must use an `EncodeIterator` and can use a single encode iterator to manage the entire encoding process¹ (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. Create an `EncodeIterator` by calling the `CodecFactory.createEncodeIterator` method. After creating the iterator, clear it using the `EncodeIterator.clear` method. Each `EncodeIterator` requires a `TransportBuffer` (into which it encodes) and the RWF version information (to ensure that the proper version of the wire format is encoded). Use the `EncodeIterator.setBufferAndRWFVersion` method to set the `TransportBuffer` and RWF version (refer to Section 10.5.1).

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 RDM 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.

10.2.2.1 **EncoderIterator** Functions

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

The following table describes methods that you can use with **EncoderIterator**. The methods listed below that take **TransportBuffers** also support **Buffers** (from the Codec package). Whereas **TransportBuffers** are used with the Transport package, **Buffers** (from the Codec package) are available for use with user-defined transports.

METHOD	DESCRIPTION
EncoderIterator.clear	Clears members necessary for encoding and readies the iterator for reuse. You must clear the EncoderIterator prior to starting any encoding process. For performance purposes, only those members necessary for proper functionality are cleared.
EncoderIterator.setBufferAndRWFVersion	Associates an EncoderIterator with the TransportBuffer into which it encodes and RWF versioning information. TransportBuffer.data should refer to sufficient space for encoding. RWF Versioning information ensures that the Transport API uses the appropriate wire format version while encoding. Wire format information is typically available on the connection between applications. Refer to Section 10.5.1.
EncoderIterator.realignBuffer	If an encoding process exceeds the space allocated in the current TransportBuffer , this method dynamically associates a new, larger buffer with the encoding process, allowing encoding to continue.

Table 54: **EncoderIterator** Utility Methods

10.2.2.2 **EncodeIterator**: Basic Use Example

The following example illustrates how to initialize **EncodeIterator** in a typical fashion.

```
/* create and clear iterator to prepare for encoding */
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
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)
 */
if (encodeIter.setBufferAndRWFVersion(buffer, chnl.majorVersion(), chnl.minorVersion()) <
    CodecReturnCodes.SUCCESS)

{
    System.out.printf("Error (%d) (errno: %d) encountered with setBufferAndRWFVersion. Error Text:
                      %s\n", error.errorId(), error.sysError(), error.text());
}

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

Code Example 14: **EncodeIterator** Usage Example

10.2.3 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 15: Encoding Rollback Example

10.3 Decoding Semantics

Using the 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 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 `CodecReturnCodes.END_OF_CONTAINER` is returned.

10.3.1 The Decode Iterator: `Decodelterator`

All decoding requires the use of an `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 an `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 an `DecodeIterator` consists of several steps. Create a `DecodeIterator` by calling the `CodecFactory.createDecodelterator` method. After the iterator is created, use `DecodeIterator.clear` to clear `DecodeIterator`. Each `DecodeIterator` requires an `TransportBuffer` from which it decodes and RWF version information, which ensures decoding of the appropriate version of the wire format (refer to Section 10.5.1).

Note: Additional concrete decoding examples are provided throughout this manual as well as in the example applications provided with the Transport API package.

10.3.2 Functions for use with `Decodelterator`

The following table describes the methods that you can use with `DecodeIterator`. The methods listed below that take `TransportBuffers` also support `Buffers` (from the Codec package). `TransportBuffers` are used with the Transport package. `Buffers` (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 <code>DecodeIterator</code> before decoding content. For performance purposes, only those members required for proper functionality are cleared.
setBufferAndRWFVersion	Associates the <code>DecodeIterator</code> with the <code>TransportBuffer</code> from which to decode and RWF version information. Set <code>TransportBuffer.data</code> to refer to the content to be decoded. RWF Version information ensures that the Transport API uses the appropriate wire format version when encoding. Wire format information is typically available on the connection between applications. Refer to Section 10.5.1.

Table 55: `Decodelterator` Utility Methods

METHOD	DESCRIPTION
finishDecodeEntries	The decoding process typically runs until the end of each container, indicated by <code>CodecReturnCodes.END_OF_CONTAINER</code> . 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 55: `DecodeIterator` Utility Methods (Continued)

10.3.3 `DecodeIterator`: Basic Use Example

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

```
/* create and clear iterator to prepare for decoding */
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
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)
*/
if (decodeIter.setBufferAndRWFVersion(buffer, chnl.majorVersion(), chnl.minorVersion()) <
    CodecReturnCodes.SUCCESS)
{
    System.out.printf("Error (%d) (errno: %d) encountered with setBufferAndRWFVersion. Error
                      Text: %s\n", error.errorId(), error.sysError(), error.text());
}

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

Code Example 16: `DecodeIterator` Usage Example

10.4 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 Codec package, return codes greater than or equal to `CodecReturnCodes.SUCCESS` indicate some type of specific success code, while codes less than `CodecReturnCodes.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 9, Transport Package Detailed View.

10.4.1 Success Codes

The following table describes success values of `CodecReturnCodes` associated with the Codec package.

CODEC RETURN CODE	DESCRIPTION
SUCCESS	Indicates operational success. Does not indicate next steps, though additional encoding or decoding might be required.
ENCODE_MSG_KEY_ATTRIB	Indicates that initial message encoding was successful and now the application should encode <code>msgKey</code> attributes. This return occurs if the application indicates that the message should include <code>msgKey</code> attributes when calling <code>Msg.encodeInit (MsgKeyFlags.HAS_ATTRIB)</code> without populating pre-encoded data into <code>msgKey.encAttrib</code> . For further details, refer to Section 12.1.2 and Code Example 42.
ENCODE_EXTENDED_HEADER	Indicates that initial message encoding (and <code>msgKey</code> attribute encoding) was successful, and the application should now encode <code>extendedHeader</code> content. This return occurs if an application indicates that the message should include <code>extendedHeader</code> content when calling <code>Msg.encodeInit</code> without populating pre-encoded data into the <code>extendedHeader</code> . For further details on message encoding information, refer to Chapter 12, Message Package Detailed View.
ENCODE_CONTAINER	Indicates that initial encoding succeeded and that the application should now encode the specified <code>containerType</code> . <ul style="list-style-type: none"> For details on container types, refer to Section 11.3. For details on encoding messages, refer to Chapter 12, Message Package Detailed View.
SET_COMPLETE	Indicates that <code>FieldList</code> or <code>ElementList</code> 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.

Table 56: Codec Package Success `CodecReturnCodes`

CODEC RETURN CODE	DESCRIPTION
DICT_PART_ENCODED	<p>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).</p> <ul style="list-style-type: none"> For specific information about the Dictionary domain and the utility functions provided by the Transport API, refer to the <i>Transport API Java Edition RDM Usage Guide</i>. For more details on message fragmentation, refer to Section 13.1.
BLANK_DATA	<p>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.</p> <p>For further details on primitive types, refer to Section 11.2.</p>
NO_DATA	<p>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).</p>
END_OF_CONTAINER	<p>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.</p> <p>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.</p> <p>For details on container types, refer to Section 11.3.</p>
SET_SKIPPED	<p>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.</p> <p>For further information on set definitions, refer to Section 11.6.</p>
SET_DEF_DB_EMPTY	<p>Indicates that decoding of a set definition database completed successfully, but the database was empty.</p> <p>For further information, refer to Section 11.6.</p>

Table 56: Codec Package Success CodecReturnCodes (Continued)

10.4.2 Failure Codes

RETURN CODE	DESCRIPTION
FAILURE	Indicates a general failure, used when no specific details are available.
BUFFER_TOO_SMALL	Indicates that the <code>TransportBuffer</code> on the <code>EncodeIterator</code> lacks sufficient space for encoding.
INVALID_ARGUMENT	Indicates an invalid argument was provided to an encoding or decoding method.
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.
UNSUPPORTED_DATA_TYPE	Indicates that the type is not supported for the operation being performed. This might indicate a <code>primitiveType</code> is used where a <code>containerType</code> is expected or the opposite.
UNEXPECTED_ENCODER_CALL	Indicates that encoding functionality was used in an unexpected sequence or the called method is not expected in this encoding.
INCOMPLETE_DATA	Indicates that the <code>TransportBuffer</code> on the <code>DecodeIterator</code> does not have enough data for proper decoding.
INVALID_DATA	Indicates that invalid data was provided to the invoked method.
ITERATOR_OVERRUN	Indicates that the application is attempting to nest more levels of content than is supported by a single <code>EncodeIterator</code> ^a . If this occurs, you should use multiple iterators for encoding.
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.
SET_DEF_NOT_PROVIDED	Indicates that <code>FieldList</code> or <code>ElementList</code> encoding requires a set definition database which was not provided. For more information, refer to Section 11.6.
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.
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.
ILLEGAL_LOCAL_SET_DEF	Indicates that the <code>setId</code> associated with a contained definition exceeds the allowable value. Currently <code>setId</code> values up to 15 are allowed. For more information, refer to Section 11.6.

Table 57: Codec Package Failure CodecReturnCodes

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

10.4.3 CodecReturnCodes Methods

CodecReturnCodes contains the following methods:

METHOD	DESCRIPTION
toString	Returns a Java String representation for a CodecReturnCodes value (e.g. “INCOMPLETE_DATA” for CodecReturnCodes.INCOMPLETE_DATA).
Info	Returns a Java String representation of the meaning associated with a CodecReturnCodes value (e.g. “Failure: Not enough data was provided.” for CodecReturnCodes.INCOMPLETE_DATA).

Table 58: **CodecReturnCodes** Methods

10.5 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 **EncoderIterator** and **DecodeIterator** to ensure the proper handling and use of the appropriate wire format version.

Note: Thomson Reuters strongly recommends that you write all 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.

10.5.1 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:

- 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 **Channel** from both client and server and can be used for encoding and decoding:

- To populate version information on a **EncoderIterator**, call the **EncoderIterator.setBufferAndRWFVersion** method.
- To populate version information on a **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 9, Transport Package Detailed View.

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

The Codec Package contains several defined values that you can access via **Codec** methods and use with protocol versioning:

METHOD	DESCRIPTION
protocolType	Defines the protocolType value associated with RWF. Define other protocols using different protocolType values.
majorVersion	Sets the value associated with the current major version. If incompatible changes are introduced, this value is incremented.
minorVersion	Sets the value associated with the current minor version. If extensions or compatible changes are introduced, this value is incremented.

Table 59: Codec Methods

10.5.2 Library Versioning

The Transport API library version information is contained in the **MANIFEST.MF** of the **upa.jar** file. The **MANIFEST.MF** contains the Transport API version data, the internal Thomson Reuters build version data, and the product date.

There are several ways in which you can obtain this data. From a console, you can use the following **jar** command to extract the **MANIFEST.MF** then examine the contents, which provides the Transport API package version data and internal version (which provides the internal Thomson Reuters build version data). Any issues raised to support should include this version data.

```
jar xf upa.jar META-INF/MANIFEST.MF
type META-INF/MANIFEST.MF
```

Code Example 17: Extract Package information from **MANIFEST.MF**

Additionally, each Transport API library includes a utility method, defined in Table 60, to programmatically extract library version information. Each method populates a **LibraryVersionInfo** object, as defined in Table 61.

METHOD	DESCRIPTION
Codec.queryVersion	Retrieves version data associated with the Codec Package library.
Transport.queryVersion	Retrieves version data associated with the Transport Package library.

Table 60: Library Version Utility Methods

METHOD	DESCRIPTION
productVersion	Returns the Package version as specified by the Specification-Version in the MANIFEST.MF .
internalVersion	Returns the internal Thomson Reuters build data as specified by the Implementation-Version in the MANIFEST.MF .

Table 61: LibraryVersionInfo Methods

METHOD	DESCRIPTION
productDate	Returns the build date for the product release as specified by the Implementation-Vendor in the MANIFEST.MF .

Table 61: LibraryVersionInfo Methods

Chapter 11 Data Package Detailed View

11.1 Concepts

The Codec 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 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 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** constants 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.

Each type represented with a constant has a corresponding class definition used when encoding or decoding that type.

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 constant 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 62.

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 62) returns **CodecReturnCodes.BLANK_DATA**. To encode blank data into a **FieldEntry** or **ElementEntry**, refer to Section 11.3.1 and Section 11.3.2.

- 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 **DataType.INT**). 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 in the **DataTypes** class, 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	ENCODE INTERFACE	DECODE INTERFACE
UNKNOWN	None	Indicates that the type is unknown. <code>DataTypes.UNKNOWN</code> 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.	None	None
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).	Int.encode	Int.decode
UINT	UInt ^b	An unsigned integer type. Can currently represent an unsigned value with precision of up to 64 bits.	UInt.encode	UInt.decode
FLOAT	Float	A four-byte, floating point type. Can represent the same range of values allowed with the Java <code>Float</code> type. Follows IEEE 754 specification.	Float.encode	Float.decode
DOUBLE	Double	An eight-byte, floating point type. Can represent the same range of values allowed with the Java <code>Double</code> type. Follows IEEE 754 specification.	Double.encode	Double.decode
REAL	Real ^c	An optimized RWF representation of a decimal or fractional value which typically requires less bytes on the wire than <code>Float</code> or <code>Double</code> 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 11.2.1.	Real.encode	Real.decode
DATE	Date	Defines a date with month, day, and year values. For more details on this type, refer to Section 11.2.2.	Date.encode	Date.decode
TIME	Time	Defines a time with hour, minute, second, millisecond, microsecond, and nanosecond values. For more details on this type, refer to Section 11.2.3.	Time.encode	Time.decode
DATETIME	DateTime	Combined representation of date and time. Contains all members of <code>Date</code> and <code>Time</code> . For more details on this type, refer to Section 11.2.4.	DateTime.encode	DateTime.decode

Table 62: Transport API Primitive Types

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION	ENCODE INTERFACE	DECODE INTERFACE
QOS	Qos	Defines QoS information such as data timeliness (e.g., real time) and rate (e.g., tick-by-tick). Allows a user to send QoS 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 11.2.5.	Qos.encode	DecodeQos.decode
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 11.2.6.	State.encode	State.decode
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, constant definition (e.g., those contained in the com.thomsonreuters.upa.rdm package).	Enum.encode	Enum.decode
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 11.2.7.	For more information, refer to Section 11.2.7.2.	For more information, refer to Section 11.2.7.5.
BUFFER	Buffer ^e	Represents a raw byte buffer type. Any semantics associated with the data in this buffer is provided from outside of the Transport API, either via a field dictionary (e.g., RDMFieldDictionary) or a DMM definition. For more details on this type, refer to Section 11.2.8.	Buffer.encode	Buffer.decode

Table 62: Transport API Primitive Types (Continued)

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION	ENCODE INTERFACE	DECODE INTERFACE
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 Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.8.	Buffer.encode	Buffer.decode
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 Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.8.	Buffer.encode	Buffer.decode
RMTES_STRING	Buffer ^e	Represents an RMTES (Reuters Multilingual Text Encoding Standard) string which should follow the RMTES encoding standard and contain only characters valid within that set. The Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.8.	Buffer.encode	Buffer.decode

Table 62: 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 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 Transport API, before encoding or after decoding this type. This type allows for a length of up to 65,535 bytes.

DataTypes contains the following methods.

METHOD	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	<ul style="list-style-type: none"> If the dataType represents a primitive type, returns true. If the dataType represents a container type, returns false.
isContainerType	<ul style="list-style-type: none"> If the dataType represents a container type, returns true. If the dataType represents a primitive type, returns false.
toString	Returns a Java String representation for a DataTypes value.

Table 63: DataTypes Methods

11.2.1 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.

11.2.1.1 Methods

Real contains the following methods:

METHOD	DESCRIPTION
isBlank	<p>Returns a Boolean value. Indicates whether the data is considered blank.</p> <ul style="list-style-type: none"> If true, the value and hint should be ignored If false, value and hint determine the resultant value. <p>This allows State to be represented as blank when used either as a primitive type or a set-defined primitive type.</p>
hint	<p>Returns a RealHints value (hint) which defines how to interpret the value contained in State. Hint values can add or remove up to seven trailing zeros, 14 decimal places, or fractional denominators up to 256.</p> <p>For more information about hint values, refer to Section 65.</p>
toLong	<p>The raw value represented by the state (omitting any decimal or denominator). Typically requires the application of hint before interpreting or performing any calculations. This member can currently represent up to 63 bits and a one-bit sign (positive or negative).</p> <p>Its value can range from (-2^{63}) to ($2^{63} - 1$).</p>
toDouble	Uses the formulas described in Section 11.2.1.3 to convert a State to a Java Double type.

Table 64: Real Methods

METHOD	DESCRIPTION
toString	Converts a state type to a numeric String representation. Blank is output as an empty zero length String .
value(<i>long</i> , <i>hint</i>)	Sets the raw long value and hint .
value(<i>double</i> , <i>hint</i>)	Uses the formulas described in Section 11.2.1.4 to convert a float and hint to a state type.
value(<i>float</i> , <i>hint</i>)	Uses the formulas described in Section 11.2.1.4 to convert a float and hint to a state type.
value(String)	Converts a numeric String with denominator or decimal information to a state type. Interprets a String of +0 as a blank State .
encode	Encodes a state into a buffer.
decode	Decodes a state from a buffer.
equals	Compares one state to another specified state . Returns true if equal, false otherwise.
copy	Performs a deep copy of one state into another specified state .
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 64: Real Methods

11.2.1.2 hint Values

The following table defines the available **hint RealHints** values for use with **Real**. The conversion routines described in Section 11.2.1.3 use **Real**'s **hint** and **toLong** value **value**.

ENUM	DESCRIPTION
RSSL_RH_EXPONENT_14	Negative exponent operation, equivalent to 10^{-14} . Shifts decimal by 14 positions.
RSSL_RH_EXPONENT_13	Negative exponent operation, equivalent to 10^{-13} . Shifts decimal by 13 positions.
RSSL_RH_EXPONENT_12	Negative exponent operation, equivalent to 10^{-12} . Shifts decimal by 12 positions.
RSSL_RH_EXPONENT_11	Negative exponent operation, equivalent to 10^{-11} . Shifts decimal by 11 positions.
RSSL_RH_EXPONENT_10	Negative exponent operation, equivalent to 10^{-10} . Shifts decimal by ten positions.
RSSL_RH_EXPONENT_9	Negative exponent operation, equivalent to 10^{-9} . Shifts decimal by nine positions.
RSSL_RH_EXPONENT_8	Negative exponent operation, equivalent to 10^{-8} . Shifts decimal by eight positions.
RSSL_RH_EXPONENT_7	Negative exponent operation, equivalent to 10^{-7} . Shifts decimal by seven positions.
RSSL_RH_EXPONENT_6	Negative exponent operation, equivalent to 10^{-6} . Shifts decimal by six positions.
RSSL_RH_EXPONENT_5	Negative exponent operation, equivalent to 10^{-5} . Shifts decimal by five positions.
RSSL_RH_EXPONENT_4	Negative exponent operation, equivalent to 10^{-4} . Shifts decimal by four positions.
RSSL_RH_EXPONENT_3	Negative exponent operation, equivalent to 10^{-3} . Shifts decimal by three positions.
RSSL_RH_EXPONENT_2	Negative exponent operation, equivalent to 10^{-2} . Shifts decimal by two positions.

Table 65: RsslRealHints Enumeration Values

ENUM	DESCRIPTION
RSSL_RH_EXPONENT_1	Negative exponent operation, equivalent to 10^{-1} . Shifts decimal by one position.
RSSL_RH_EXPONENT0	Exponent operation, equivalent to 10^0 . value does not change.
RSSL_RH_EXPONENT1	Positive exponent operation, equivalent to 10^1 . Depending on the type of conversion, this adds or removes one trailing zero.
RSSL_RH_EXPONENT2	Positive exponent operation, equivalent to 10^2 . Depending on the type of conversion, this adds or removes two trailing zeros.
RSSL_RH_EXPONENT3	Positive exponent operation, equivalent to 10^3 . Depending on the type of conversion, this adds or removes three trailing zeros.
RSSL_RH_EXPONENT4	Positive exponent operation, equivalent to 10^4 . Depending on the type of conversion, this adds or removes four trailing zeros.
RSSL_RH_EXPONENT5	Positive exponent operation, equivalent to 10^5 . Depending on the type of conversion, this adds or removes five trailing zeros.
RSSL_RH_EXPONENT6	Positive exponent operation, equivalent to 10^6 . Depending on the type of conversion, this adds or removes six trailing zeros.
RSSL_RH_EXPONENT7	Positive exponent operation, equivalent to 10^7 . Depending on the type of conversion, this adds or removes seven trailing zeros.
RSSL_RH_FRACTION_1	Fractional denominator operation, equivalent to 1/1. Value does not change.
RSSL_RH_FRACTION_2	Fractional denominator operation, equivalent to 1/2. Depending on the type of conversion, this adds or removes a denominator of two.
RSSL_RH_FRACTION_4	Fractional denominator operation, equivalent to 1/4. Depending on the type of conversion, this adds or removes a denominator of four.
RSSL_RH_FRACTION_8	Fractional denominator operation, equivalent to 1/8. Depending on the type of conversion, this adds or removes a denominator of eight.
RSSL_RH_FRACTION_16	Fractional denominator operation, equivalent to 1/16. Depending on the type of conversion, this adds or removes a denominator of 16.
RSSL_RH_FRACTION_32	Fractional denominator operation, equivalent to 1/32. Depending on the type of conversion, this adds or removes a denominator of 32.
RSSL_RH_FRACTION_64	Fractional denominator operation, equivalent to 1/64. Depending on the type of conversion, this adds or removes a denominator of 64.
RSSL_RH_FRACTION_128	Fractional denominator operation, equivalent to 1/128. Depending on the type of conversion, this adds or removes a denominator of 128.
RSSL_RH_FRACTION_256	Fractional denominator operation, equivalent to 1/256. Depending on the type of conversion, this adds or removes a denominator of 256.
INFINITY	Value should be interpreted as infinity (Inf).
NEG_INFINITY	Value should be interpreted as negative infinity (-Inf).
NOT_A_NUMBER	Value should be interpreted as not a number (NaN).

Table 65: **RsslRealHints** Enumeration Values (Continued)

11.2.1.3 Hint Use Case: Converting an Real to a Float or a Double

An application can convert between an **Real** and a Java **float** or **double** as needed. Converting a **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 18: Real Conversion to Double/Float

11.2.1.4 Hint Use Case: Converting Double or Float to an 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 19: Real Conversion from Double/Float

11.2.2 Date

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 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 Java String . Returns a String as " DD MM YYYY ".
value	Converts a Java String date from " DD MMM YYYY " (e.g., 01 JUN 2003) or " MM/DD/YYYY " (e.g., 6/1/2003) format to Date .
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 in 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 66: **Date** Methods

11.2.3 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 otherwise¹.

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 Java String . Returns the String as " hour:minute:second:milli:micro:nano " (e.g. 15:24:54:627:843:143).
value	Converts a Java String time from " HH:MM " (13:01) or " HH:MM:SS " (15:23:54) format to Time .
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 67: **Time** Methods

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

11.2.4 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 otherwise².

DateTime provides convenient methods to set or get **Date** and **Time** values directly, or **Date** and **Time** can be retrieved and used independently.

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 11.2.2.
time	Returns the Time portion of the DateTime and conforms to the behaviors described in Section 11.2.3.
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 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.

Table 68: **DateTime** Methods

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

METHOD	DESCRIPTION
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.
millisSinceEpoch	Returns the date-time value as milliseconds since the January 1, 1970 (midnight UTC/GMT) epoch.
toString	Convert DateTime to a Java string . Returns a string as “ %d %b %Y hour:minute:second:milli:micro:nano ” (e.g., 30 NOV 2010 15:24:54:627:843:143).
value(String)	Converts a Java string representation of a date and time to a DateTime . This method supports: <ul style="list-style-type: none"> • 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 68: **DateTime** Methods (Continued)

11.2.5 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 11.2.5.1, Section 11.2.5.2, and Section 11.2.5.3.

If present in a data payload, specific handling and interpretation associated with QoS information is provided from outside of the Transport API, possibly via the specific DMM definition.

Several Transport API message headers also contain QoS data. When present, this data is typically used to request or convey the QoS associated with a particular stream. For more information about QoS 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.

11.2.5.1 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 11.2.5.2.
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 11.2.5.3.
dynamic	Describes the changeability of the QoS within the requested range, typically over the life of a data stream. <ul style="list-style-type: none"> • If set to false, the QoS should not change following the initial establishment. • If set to true, the QoS can change over time to other values within the requested range. QoS can change due to permissioning information, stream availability, network congestion, or other reasons. Specific information about dynamically changing QoS should be described in documentation for components that support this behavior.
isDynamic	Returns true if the QoS 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.TIME_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 . <ul style="list-style-type: none"> • Returns true if the values contained in the structure are identical. • Returns false if the values contained in the structure differ.

Table 69: **Qos** Methods

METHOD	DESCRIPTION
isBetter	Compares this <code>Qos</code> with a specified <code>Qos</code> to determine which has better overall quality. <ul style="list-style-type: none"> Returns <code>true</code> if this <code>Qos</code> is better. Returns <code>false</code> if this <code>Qos</code> is not better.
isInRange	Determines whether this <code>Qos</code> lies within a range from best <code>Qos</code> to worst <code>Qos</code> . <ul style="list-style-type: none"> Returns <code>true</code> if this <code>Qos</code> falls between best and worst <code>Qos</code> Returns <code>false</code> if this <code>Qos</code> falls outside of the best or worst <code>Qos</code> range.
blank	Clears this object and sets it to blank.
isBlank	Returns <code>true</code> if <code>Qos</code> is blank, otherwise <code>false</code> .
toString	Returns a Java String representation for this <code>Qos</code> .
copy	Performs a deep copy of the <code>Qos</code> to another specified <code>Qos</code> .
encode	Encodes <code>Qos</code> into a buffer.
decode	Decodes <code>Qos</code> from a buffer.
clear	Clears this object, so that you can reuse it. Sets all members in <code>Qos</code> to an initial value of 0. This includes setting <code>rate</code> and <code>timeliness</code> to their unspecified values (not intended to be encoded or decoded).

Table 69: `Qos` Methods (Continued)

11.2.5.2 Qos Timeliness Values

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

Table 70: `QosTimeliness` Values

11.2.5.3 QosRates Values

QOS RATE	DESCRIPTION
UNSPECIFIED	<code>rate</code> is unspecified. Typically used by QoS initialization methods and not intended to be encoded or decoded.
TICK_BY_TICK	<code>rate</code> is tick-by-tick (i.e., data is sent for every update). This is the highest quality <code>rate</code> value. The best overall QoS is a tick-by-tick <code>rate</code> with a <code>timeliness</code> of <code>QosTimeliness.REALTIME</code> .
JIT_CONFLATED	<code>rate</code> 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 <code>TICK_BY_TICK</code> . 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 <code>TICK_BY_TICK</code> . However, when the application cannot keep up with data rates, it results in a rate similar to <code>TIME_CONFLATED</code> , where <code>rateInfo</code> is determined by the provider. Specific information about <code>conflationTime</code> or <code>conflationCount</code> might be present in an <code>UpdateMsg</code> . For further details, refer to Section 12.2.3.
TIME_CONFLATED	<code>rate</code> is time-conflated. The interval of time (usually in milliseconds) over which data are conflated is provided in <code>Qos.rateInfo</code> . This is lower quality than <code>TICK_BY_TICK</code> and at times even lower than <code>JIT_CONFLATED</code> . Specific information about the <code>conflationTime</code> or <code>conflationCount</code> might be present in the <code>UpdateMsg</code> . For more details, refer to Section 12.2.3.

Table 71: QosRates Values

11.2.6 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 DMM.

Several 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.

11.2.6.1 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 11.2.6.2.
dataState	Sets or gets the dataState , which conveys data about the health of data flowing within a stream. DataState values are described in Section 11.2.6.4.
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 11.2.6.6. 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 . <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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.

Table 72: **State** Methods

METHOD	DESCRIPTION
clear	Clears this object for reuse. Sets all members in <code>state</code> to an initial value. This includes setting <code>streamState</code> to its unspecified value (not intended to be encoded or decoded).

Table 72: **State** Methods (Continued)

11.2.6.2 StreamStates Values

STREAM STATE	DESCRIPTION
UNSPECIFIED	<code>streamState</code> is unspecified. Typically used as a structure initialization value and is not intended to be encoded or decoded.
OPEN	<code>streamState</code> is open. This typically means that data is streaming: as data changes, they are sent on the stream.
NON_STREAMING	<code>streamState</code> is non-streaming. After receiving a final <code>RefreshMsg</code> or <code>StatusMsg</code> , 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.
CLOSED_RECOVER	<code>streamState</code> is closed, however data can be recovered on this service and connection at a later time. This state can occur via either a <code>RefreshMsg</code> or a <code>StatusMsg</code> . Single Open behavior can modify this state (continuing to indicate a stream state of <code>OPEN</code>) and attempt to recover data on the user's behalf. For further details on Single Open behavior, refer to Section 13.5.
CLOSED	<code>streamState</code> 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 <code>RefreshMsg</code> or an <code>StatusMsg</code> .
REDIRECTED	<code>streamState</code> 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 a <code>RefreshMsg</code> or a <code>StatusMsg</code> . For further details, refer to Section 12.1.3.2.

Table 73: **StreamStates** Values

11.2.6.3 StreamStates Methods

METHOD	DESCRIPTION
toString	Returns a Java <code>String</code> representation for a <code>StreamStates</code> value (e.g. "CLOSED_RECOVER" for <code>StreamStates.CLOSED_RECOVER</code>).
info	Returns a Java <code>String</code> representation of any information associated with a <code>StreamStates</code> value (e.g. "Closed, Recoverable" for <code>StreamStates.CLOSED_RECOVER</code>).

Table 74: **StreamStates** Methods

11.2.6.4 DataStates Values

DATA STATE	DESCRIPTION
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 <code>OK</code> or <code>SUSPECT</code>) instead of <code>NO_CHANGE</code> . This typically conveys <code>code</code> or <code>text</code> information associated with an item group, but no change to the group's previous data and stream state has occurred.
OK	<code>dataState</code> is <code>OK</code> . All data associated with the stream is healthy and current.
SUSPECT	<code>dataState</code> is <code>SUSPECT</code> (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 75: DataStates Values

11.2.6.5 DataStates Methods

METHOD	DESCRIPTION
toString	Returns a Java <code>String</code> representation for a <code>Datastates</code> value (e.g. "NO_CHANGE" for <code>DataStates.NO_CHANGE</code>).
info	Returns a Java String representation of any information associated with a <code>Datastates</code> value (e.g. "No Change" for <code>DataStates.NO_CHANGE</code>).

Table 76: DataStates Methods

11.2.6.6 StateCodes Values

STATE CODE	DESCRIPTION
NONE	Indicates that additional state code information is not required, nor present.
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.
TIMEOUT	Indicates that a timeout occurred somewhere in the system while processing requested data.
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 QoS.
INVALID_ARGUMENT	Indicates that the request includes an invalid or unrecognized parameter. Specific information should be contained in the <code>text</code> .

Table 77: StateCodes Values

STATE CODE	DESCRIPTION
USAGE_ERROR	Indicates invalid usage within the system. Specific information should be contained in the <code>text</code> .
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 <code>text</code> .
JIT_CONFLATION_STARTED	Indicates that JIT conflation has started on the stream. User is notified when JIT Conflation ends via <code>StateCodes.REALTIME_RESUMED</code> .
REALTIME_RESUMED	Indicates that JIT conflation on the stream has finished.
FAILOVER_STARTED	Indicates that a component is recovering due to a failover condition. User is notified when recovery finishes via <code>StateCodes.FAILOVER_COMPLETED</code> .
FAILOVER_COMPLETED	Indicates that recovery from a failover condition has finished.
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 <code>seqNum</code> present in Transport API messages.
NO_RESOURCES	Indicates that no resources are available to accommodate the stream.
TOO_MANY_ITEMS	Indicates that a request cannot be processed because too many other streams are already open.
ALREADY_OPEN	Indicates that a stream is already open on the connection for the requested data.
SOURCE_UNKNOWN	Indicates that the requested service is not known, though the service might be available at a later point in time.
NOT_OPEN	Indicates that the stream was not opened. Additional information should be available in the <code>text</code> .
NON_UPDATING_ITEM	Indicates that a streaming request was made for non-updating data.
UNSUPPORTED_VIEW_TYPE	Indicates that the domain on which a request is made does not support the requested <code>viewType</code> . Section 13.8 discusses views in more detail.
INVALID_VIEW	Indicates that the requested view is invalid, possibly due to bad formatting. Additional information should be available in the <code>text</code> . Section 13.8 discusses views in more detail.
FULL_VIEW_PROVIDED	Indicates that the full view (e.g., all available fields) is being provided, even though only a specific view was requested. Section 13.8 discusses views in more detail.

Table 77: `StateCodes` Values(Continued)

STATE CODE	DESCRIPTION
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. Section 13.7 discusses batch requesting in more detail.
NO_BATCH_VIEW_SUPPORT_IN_REQ	Indicates that the provider does not support batch and/or view functionality.
EXCEEDED_MAX_MOUNTS_PER_USER	Indicates that the login was rejected because the user exceeded their maximum number of allowed mounts.
ERROR	Indicates an internal error from the sender.
DACS_DOWN	Indicates that the connection to DACS is down and users are not allowed to connect.
USER_UNKNOWN_TO_PERM_SYS	Indicates that the user is unknown to the permissioning system and is not allowed to connect.
DACS_MAX_LOGINS_REACHED	Indicates that the maximum number of logins has been reached.
DACS_USER_ACCESS_TO_APP_DENIED	Indicates that the application is denied access to the system.
GAP_FILL	Indicates that the received content is meant to fill a recognized gap.
APP_AUTHORIZATION_FAILED	Indicates that application authorization using the secure token has failed.

Table 77: **StateCodes** Values(Continued)

11.2.6.7 StateCodes Methods

METHOD	DESCRIPTION
toString	Returns a Java String representation for a StateCodes value (e.g. “NON_UPDATING_ITEM” for StateCodes.NON_UPDATING_ITEM).
info	Returns a Java String representation of any information associated with a StateCodes value (e.g. “Non-updating item” for StateCodes.NON_UPDATING_ITEM).

Table 78: **StateCodes** Methods

11.2.7 Array

The **Array** is a uniform primitive type that can contain multiple simple primitive entries. A **Array** can contain zero to N primitive type entries³, where zero entries indicates an empty **Array**.

Each **ArrayEntry** can house only simple primitive types such as **Int**, **Real**, or **Date**. An **ArrayEntry** cannot house any container types or other **Array** types. This is a uniform type, where the **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.

A **ArrayEntry** 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 a **Buffer** (containing the pre-encoded primitive).

When decoding, the encoded content of the **ArrayEntry** is available as a **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.

11.2.7.1 Array Methods

METHOD	DESCRIPTION
primitiveType	Using a DataTypes value, primitiveType describes the base primitive type of each entry. Array can only contain simple primitive types and cannot house container types or other Arrays .
itemLength ^a	Sets the expected length of all array entries. <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> 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 TransportBuffer 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 Array.primitiveType and Array.itemLength methods have been properly populated. The EncodeIterator specifies the TransportBuffer into which it encodes data. Entries can be encoded after this method returns.

Table 79: **Array** Structure Members

3. An **Array** 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 an **Array** entry is bound by the maximum encoded length of the primitive types being contained. These limitations can change in subsequent releases.

METHOD	DESCRIPTION
encodeComplete	<p>Completes encoding of an Array. This method expects the same EncodeIterator used with encodeInit and ArrayEntry.encode methods.</p> <p>Set the boolean parameter to:</p> <ul style="list-style-type: none"> True if the array encoded successfully and to finish encoding. False if encoding of any entry failed and to roll back the encoding process to the last successfully-encoded point in the contents. <p>All entries should be encoded before calling encodeComplete.</p>
decode	Begins decoding an Array . This method decodes from the TransportBuffer specified to the DecodeIterator .
isBlank	Returns true if State is blank, otherwise false .
clear	<p>Clears this object, so that you can reuse it. Sets all members to an initial value.</p> <p> Tip: When decoding, the Array object can be reused without using clear.</p>

Table 79: Array Structure Members (Continued)

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.RMTE_STRING](#) support any legal length value; see those types for allowable lengths.

11.2.7.2 ArrayEntry Methods

METHOD	DESCRIPTION
encodedData	<ul style="list-style-type: none"> When encoding, this method specifies pre-encoded data for an <code>ArrayEntry</code>. Populate a <code>Buffer</code> with pre-encoded data, then call this method with the <code>Buffer</code>. When decoding, the decode method will populate a <code>Buffer</code> with the encoded primitive type (if any). Call this method without a parameter to return the <code>Buffer</code> containing the encoded primitive type.
encodeBlank	Encodes a blank entry.
encode	<p>Encodes an <code>ArrayEntry</code>. This method expects the same <code>EncodeIterator</code> used with <code>Array.encodeInit</code>.</p> <ul style="list-style-type: none"> If encoding from pre-encoded data, specify the <code>Buffer</code> populated with pre-encoded data. If encoding from a primitive type, specify the primitive type. (e.g. <code>UInt</code>). <p>This method should be called for each entry being encoded. The specified type must match the <code>Array.primitiveType</code>.</p>
decode	Decodes an <code>ArrayEntry</code> and populates an internal <code>Buffer</code> (available via <code>encodedData</code> method) with encoded entry contents. This method expects the same <code>DecodeIterator</code> used with <code>Array.decode</code> . Any contained primitive type's decode method can be called based on <code>Array.primitiveType</code> (e.g. <code>Uint.decode</code>) (refer to Section 11.2). Calling <code>ArrayEntry.decode</code> again will decode and provide the next entry in the <code>Array</code> until no more entries are available.
clear	<p>Clears the object so that you can reuse it. Sets all members to an initial value.</p> <p> Tip: When decoding, you can reuse the <code>Array</code> object without using <code>clear</code>.</p>

Table 80: `ArrayEntry` Methods

11.2.7.3 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 a **Buffer** containing a pre-encoded **UInt** type. The 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 = CodecFactory.createArray();
ArrayEntry arrayEntry = CodecFactory.createArrayEntry();
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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Array.encodeInit. Error Text:
                      %s\n", error.errorId(), error.sysError(), error.text());
}
else
{
    UInt uInt = CodecFactory.createUInt();
    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 20: Array Encoding Example #1

11.2.7.4 Encoding: Example 2

This example demonstrates encoding an **Array** containing ASCII string values. The example includes error handling for the initial encode method 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 = CodecFactory.createBuffer();
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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Array.encodeInit. Error Text: %s\n",
                      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 21: Array Encoding Example #2

11.2.7.5 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 a [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 method. 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)) >= CodecReturnCodes.SUCCESS)
{
    /* decode each array entry */
    while ((retCode = arrayEntry.decode(decIter)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with ArrayEntry.decode. Error Text:
                               %s\n", 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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Array.decode. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

```

Code Example 22: Array Decoding Example

11.2.8 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, RMES, or UTF8 strings.
- Contain encoded data on both container and message header structures.

No validation or enforcement checks are performed on the contents of a **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 **Buffers** are typically backed by Java **ByteBuffers**, they can also be backed by Java **Strings**.

- 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. Thomson Reuters recommends that users pool their **ByteBuffers** for reuse, otherwise it will be garbage collected whenever the reference is lost.

Note: If data is backed by a Java **String** and the **Buffer.data** method is called, garbage is created to return a **ByteBuffer**.

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**.

11.2.8.1 Methods

Buffer contains the following methods:

STRUCTURE MEMBER	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 Java ByteBuffer that contains some type of content, where the specific type description of the content is provided outside of the Transport API via an external source (domain model definition, field dictionary, etc.). <ul style="list-style-type: none"> • Do not use the position and limit from the ByteBuffer. • Use position and length from the Buffer.
	Note: If data is backed by a Java String , garbage is created to return a ByteBuffer .

Table 81: **Buffer** Methods

STRUCTURE MEMBER	DESCRIPTION
data(ByteBuffer)	Sets the Buffer data to the ByteBuffer . Position and length are derived from the ByteBuffer's position and limit.
data(ByteBuffer, position, limit)	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.
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 , a ByteBuffer , or a byte[] (with or without a destination offset). The destination must have adequate space.
encode	Encodes a Buffer .
decode	Decodes a Buffer .
toString	Converts the underlying buffer into a Java 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 81: **Buffer** Methods (Continued)

11.2.8.2 Example

For performance purposes contents are not copied while decoding **Buffer**. 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.

```
/* display only the specified length of Buffer contents */
System.out.println(buffer.toString());
```

Code Example 23: Displaying Contents of an **Buffer**

11.2.9 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 *Reuters 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.

11.2.9.1 RmtesCacheBuffer: Structure

The `RmtesCacheBuffer` is a simple structure 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:

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

Table 82: `RmtesCacheBuffer` Methods

11.2.9.2 RmtesBuffer: Structure Members

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:

STRUCTURE MEMBER	DESCRIPTION
allocatedLength()	Returns the length (in bytes) allocated when creating <code>data</code> . This is typically larger than <code>length</code> to allow for the growth of <code>data</code> when applying future partial updates.
allocatedLength(Integer)	Sets the <code>allocatedLength</code> of the data (in bytes).
byteData(ByteBuffer)	Sets the <code>ByteBuffer</code> data to that of the input. The length must be set separately for the <code>RmtesBuffer</code> using <code>length(Integer)</code> .
data()	Returns the RMTEs content as a <code>ByteBuffer</code> . The length member should be set to the number of bytes of data in the buffer.
length()	Returns the length integer of the content pointed to by <code>data</code> ; it represents the number of bytes of RMTEs content.
length(Integer)	Sets the length (in bytes) of the actual data.
toString()	Converts the underlying buffer into a Java <code>String</code> . This should only be called when the <code>RmtesBuffer</code> is known to contain UTF-16 data.
	 Warning! Unless the underlying buffer is a <code>String</code> , this method creates garbage.

Table 83: `RmtesBuffer` Methods

11.2.9.3 RmtesDecoder

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 <code>RmtesCacheBuffer</code> .
	Note: The <code>RmtesCacheBuffer</code> must be large enough to handle the additional data.
hasPartialRMTESUpdate(Buffer)	Returns a boolean for whether the buffer contains a partial update command: <ul style="list-style-type: none"> • <code>true</code>: RMTEs content in the buffer contains a partial update command. • <code>false</code>: RMTEs content in the buffer does not contain a partial update command.
RMTESToUCS2(RmtesBuffer, RmtesCacheBuffer)	Converts the given <code>RmtesCacheBuffer</code> into UCS2 Unicode and stores the data into the <code>RmtesBuffer</code> .

Table 84: RmtesDecoder Decode Functions

11.2.9.4 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 = CodecFactory.createRmtesCacheBuffer(100);
/* create RmtesBuffer to convert into */
RmtesBuffer rmtesBuffer =CodecFactory.createRmtesBuffer(100);
/* create RmtesDecoder used for the decoding process */
RmtesDecoder decoder = CodecFactory.createRmtesDecoder();
/*Our Buffer of data we are converting */
Buffer data = CodecFactory.createBuffer();

/* apply RMTES content to cache, if successful convert to UCS-2 */
if ((retVal = decoder.RMTESApplyToCache(data, rmtesCache)) < CodecReturnCodes.SUCCESS)
{
    /* error while applying to cache */
    System.out.println("Error encountered while applying buffer to RMTES cache. Error code: "
        + CodecReturnCodes.toString(retVal));
}
else if ((retVal = decoder.RMTESToUCS2(rmtesBuffer, rmtesCache)) < CodecReturnCodes.SUCCESS)
{
    /* error when converting */
    System.out.println("Error encountered while converting from RMTES to UCS-2. Error code: "
        + CodecReturnCodes.toString(retVal));
}
else
{
    /* SUCCESS: Conversion was successful - application can now use converted content stored in
       rmtesBuffer */
}

```

Code Example 24: 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 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. Transport API messages and container types can house other 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	CONTAINER TYPE	DESCRIPTION	ENTRY TYPE INFORMATION
FIELD_LIST	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.	<p>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.</p> <ul style="list-style-type: none"> • 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.

Table 85: Transport API Container Types

ENUM TYPE NAME	CONTAINER TYPE	DESCRIPTION	ENTRY TYPE INFORMATION
ELEMENT_LIST	<code>ElementList</code>	A self-describing, non-uniform type, with each entry containing <code>name</code> , <code>dataType</code> , and a value. This type is equivalent to <code>FieldList</code> , but without the optimizations provided through <code>fieldID</code> 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.	<p>Entry type is <code>ElementEntry</code>, which can house any <code>Rss1DataType</code>, including set-defined data (Section 11.6), base primitive types (Section 11.2), and container types.</p> <ul style="list-style-type: none"> 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.
MAP	<code>Map</code>	<p>A container of key-value paired entries. <code>Map</code> is a uniform type, where the base primitive type of each entry's key and the <code>containerType</code> of each entry's payload are specified on the <code>Map</code>.</p> <ul style="list-style-type: none"> For more information on base primitive types, refer to Section 11.2. For more details on this container, refer to Section 11.3.3. 	<p>Entry type is <code>MapEntry</code>, which can include only container types, as specified on the <code>Map</code>. Each entry's key is a base primitive type, as specified on the <code>Map</code>. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.</p>
SERIES	<code>Series</code>	<p>A uniform type, where the <code>containerType</code> of each entry is specified on the <code>Series</code>. 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.</p>	<p>Entry type is <code>SeriesEntry</code>, which can include only container types, as specified on the <code>series</code>. <code>SeriesEntry</code> types do not contain explicit actions; though as entries are received, the user should append them to any previously received entries.</p>
VECTOR	<code>Vector</code>	<p>A container of position index-value paired entries. This container is a uniform type, where the <code>containerType</code> of each entry's payload is specified on the <code>Vector</code>. Each entry's <code>index</code> is represented by an unsigned integer. For more details on this container, refer to Section 11.3.5.</p>	<p>Entry type is <code>VectorEntry</code>, which can house only container types, as specified on the <code>vector</code>. Each entry's <code>index</code> is an unsigned integer. Each entry has an associated action, which informs the user on how to apply the information stored in the entry.</p>

Table 85: Transport API Container Types (Continued)

ENUM TYPE NAME	CONTAINER TYPE	DESCRIPTION	ENTRY TYPE INFORMATION
FILTER_LIST	FilterList	<p>A non-uniform container of <code>filterId</code>-value paired entries. A <code>filterId</code> corresponds to one of 32 possible bit-value identifiers, typically defined by a domain model specification. <code>FilterId</code>'s can be used to indicate interest or presence of specific entries through the inclusion of the <code>filterId</code> in the message key's <code>filter</code> member.</p> <ul style="list-style-type: none"> 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 <code>FilterEntry</code> , which can house only container types. Though the <code>FilterList</code> can specify a <code>containerType</code> , 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.
MSG	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 Chapter 12, Message Package Detailed View.	None
NO_DATA	None	<p>Indicates there are no contents.</p> <ul style="list-style-type: none"> When <code>DataTypes.NO_DATA</code> is housed in a message, the message has no payload. If <code>DataTypes.NO_DATA</code> is housed in a container type, each container entry has no payload.^a 	None
ANSI_PAGE	None	Indicates that contents are ANSI Page format. Though the Transport API does not natively support encoding and decoding for the ANSI Page format, the Transport API supports the use of a separate ANSI Page encoder/decoder. For further details, refer to the <i>Transport API ANSI Library Manual</i> . For more details on housing non-RWF types inside of container types, refer to Section 11.3.7.	None
XML	None	Indicates that contents are XML-formatted data. Though the Transport API does not natively support encoding and decoding XML, the Transport API supports the use of a separate XML encoder/decoder. For more details on housing non-RWF types inside of container types, refer to Section 11.3.7.	None

Table 85: Transport API Container Types (Continued)

ENUM TYPE NAME	CONTAINER TYPE	DESCRIPTION	ENTRY TYPE INFORMATION
OPAQUE	None	Indicates that the contents are opaque and additional details are not provided through the 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-RWF types inside of container types, refer to Section 11.3.7.	None

Table 85: Transport API Container Types (Continued)

a. A **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 Structure Members

FieldList includes the following methods:

METHOD	DESCRIPTION
flags	<p>Sets or gets a combination of bit values (flags) that indicate the presence of optional field list content. For more information about FieldListFlags values, refer to Section 11.3.1.2.</p> <ul style="list-style-type: none"> 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	<p>Sets or gets a two-byte, signed integer (dictionaryId) that refer 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.</p> <p>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.</p> <p>A dictionaryId can be changed in two ways.</p> <ul style="list-style-type: none"> 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 a 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 containerTypes 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. <p>dictionaryId values have an allowed range of -16,384 to 16,383.</p>
fieldListNum	<p>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.</p> <p>fieldListNum values have an allowed range of -32,768 to 32,767.</p>

Table 86: **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
setId	Sets or gets a two-byte, unsigned integer (<code>setId</code>) corresponding to the set definition used for encoding or decoding the set-defined data in this <code>FieldList</code> . <ul style="list-style-type: none"> 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 <code>setId</code> value is not present on a message containing set-defined data, a <code>setId</code> of 0 is implied. <code>setId</code> 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.
encodedSetData	Sets or gets <code>encodedSetData</code> , which is a <code>Buffer</code> (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 <code>setId</code> 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 <code>encodedEntries</code> , which is a <code>Buffer</code> (with position and length) that contains the encoded <code>fieldId</code> -value pair encoded data, if any, contained in the message. This would refer to encoded <code>FieldList</code> payload and length information.
encodeInit	Begins encoding a <code>FieldList</code> . The Transport API will encode all content into the <code>Buffer</code> to which the passed in <code>EncodeIterator</code> refers. Entries can be encoded after this method returns. <ul style="list-style-type: none"> If you are encoding set-defined data, pass in the set definition database to this method. The 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 <code>FieldList</code>). 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 <code>FieldList</code> . This method expects the same <code>EncodeIterator</code> that was used with <code>encodeInit</code> and all entries. <ul style="list-style-type: none"> If encoding succeeds, a <code>boolean success</code> parameter setting of <code>true</code> finishes the encoding. If encoding any entry fails, a <code>boolean success</code> parameter setting of <code>false</code> rolls back encoding to the last successfully encoded point in the contents. Encode all field entries prior to this call.
decode	Begins decoding a <code>FieldList</code> from the <code>Buffer</code> referenced in the <code>DecodeIterator</code> . This method allows the user to pass in local set definitions. If the <code>FieldList</code> contains set-defined data (e.g., if the <code>FieldListFlags.HAS_SET_DATA</code> 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 <code>FieldList</code> without needing to call <code>clear</code> .

Table 86: FieldList Methods (Continued)

11.3.1.2 FieldListFlag Values

FIELD LIST FLAG	MEANING
NONE	Indicates that optional flags are not set.
HAS_FIELD_LIST_INFO	Indicates that <code>dictionaryId</code> and <code>fieldListNum</code> 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 <code>CLEAR_CACHE</code> command.
HAS_STANDARD_DATA	Indicates that the <code>FieldList</code> contains standard <code>fieldId</code> -value pair encoded data. This value can be set in addition to <code>HAS_SET_DATA</code> if both standard and set-defined data are present in this <code>FieldList</code> . If no entries are present in the <code>FieldList</code> , this flag value should not be set.
HAS_SET_DATA	Indicates that the <code>FieldList</code> contains set-defined data. This value can be set in addition to <code>HAS_STANDARD_DATA</code> if both standard and set-defined data are present in this <code>FieldList</code> . If no entries are present in the <code>FieldList</code> , this flag value should not be set. For more information, refer to Section 11.6.
HAS_SET_ID	Indicates the presence of a <code>setId</code> , used to determine the set definition used for encoding or decoding the set data on this <code>FieldList</code> . For more information, refer to Section 11.6.

Table 87: FieldListFlag Values

11.3.1.3 FieldEntry Methods

Each FieldEntry can house any **DataTypes**. This includes primitive types (as described in Section 11.2), set-defined types (as described in Section 11.6), 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 Thomson Reuters-defined values. fieldId has an allowable range of -32,768 to 32,767 where Thomson Reuters 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 DataTypes of this FieldEntry 's contents. <ul style="list-style-type: none"> 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 DataTypes 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 . <ul style="list-style-type: none"> 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 88: **FieldEntry** Methods

METHOD	DESCRIPTION
encodeInit	<p>Encodes a <code>FieldEntry</code> from a complex type, such as a container type or an array. This method expects the same <code>EncodeIterator</code> used with <code>FieldList.encodeInit</code>. You must properly populate <code>FieldEntry.fieldId</code> and <code>FieldEntry.dataType</code>.</p> <p>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 <code>0</code> which allows the maximum content length.</p> <p>Typical use (e.g. encode an element list as a field entry):</p> <ol style="list-style-type: none"> 1. Call <code>FieldEntry.encodeInit</code>. 2. Call one or more encoding methods for the complex type using the same buffer. 3. Call <code>FieldEntry.encodeComplete</code>.
encodeComplete	<p>Completes encoding of a <code>FieldEntry</code> for a complex type, such as a container type or an array. This method expects the same <code>EncodeIterator</code> used with <code>FieldList.encodeInit</code>, <code>FieldEntry.encodeInit</code>, and all other entry encoding.</p> <ul style="list-style-type: none"> • If encoding succeeds, set the <code>boolean success</code> to <code>true</code> to finish entry encoding. • If encoding the entry fails, set the <code>boolean success</code> parameter to <code>false</code> to roll back the encoding of this particular <code>FieldEntry</code>.
decode	<p>Decodes a <code>FieldEntry</code>, expecting the same <code>DecodeIterator</code> used with <code>FieldList.decode</code> and populates <code>encodedData</code> with the entry's encoded contents.</p> <ul style="list-style-type: none"> • If decoding set-defined entries, the <code>FieldEntry.dataType</code> populates with the type from the set definition. • If decoding standard <code>fieldId</code>-value data, <code>FieldEntry.dataType</code> is set to <code>DataTypes.UNKNOWN</code>, indicating that the user must determine the type from a field dictionary. <p>After determining the type, the specific decode method can be called if needed. Calling <code>FieldEntry.decode</code> again will begin decoding the next entry in the <code>FieldList</code> until no more entries are available.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, <code>FieldEntry</code> can be reused without using <code>clear</code>.</p>

Table 88: `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 dictionary⁵). 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 Transport API does not perform entry rippling.

5. In the RDM Field Dictionary, the 'RIPPLES TO' column defines the `fieldId` information to use when rippling.

11.3.1.5 Encoding Example

The following example illustrates how to encode a **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 a **FieldEntry**.

This example demonstrates error handling for the initial encode method. 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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with FieldList.encodeInit. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}
else
{
    /* fieldListInit encoding was successful */
    /* create a single FieldEntry and reuse for each entry */
    FieldEntry fieldEntry = CodecFactory.createFieldEntry();
    /* stack allocate a date and populate {day, month, year} */
    com.thomsonreuters.upa.codec.Date date = CodecFactory.createDate();
    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 encData
   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)) < CodecReturnCodes.SUCCESS)

        /*----- Continue encoding array entries. See example in Section 11.2.7 ----- */

        /* 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 25: FieldList Encoding Example

11.3.1.6 Decoding Example

The following example demonstrates how to decode a `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 method. 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)) >= CodecReturnCodes.SUCCESS)
{
    /* decode each field entry */
    while ((retCode = fieldEntry.decode(decIter)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with FieldEntry.decode. Error Text:
                               %s\n", error.errorId(), error.sysError(), error.text());
        }
        else
        {
            /* look up type in field dictionary and call correct primitive decode method */
            DictionaryEntry 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 */
    System.out.printf("Error (%d) (errno: %d) encountered with FieldList.decode. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

```

Code Example 26: 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 Structure Members

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. <ul style="list-style-type: none"> You can use the following convenient methods to set specific ElementListFlags: applyHasInfo, applyHasSetData, applyHasSetId, applyHasStandardData. 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 . <ul style="list-style-type: none"> 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. <ul style="list-style-type: none"> 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 89: 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	<p>Starts encoding an <code>ElementList</code>.</p> <p>The Transport API encodes data into the <code>TransportBuffer</code> referred to by the <code>EncodeIterator</code>. Entries can be encoded after this method returns.</p> <ul style="list-style-type: none"> 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 <code>ElementList</code>). If the approximate length of encoded set data is not known, you can pass in a value of 0. <p>For more details on local set definitions, refer to Section 11.6.</p>
encodeComplete	<p>Completes the encoding of an <code>ElementList</code>.</p> <p>This method expects the same <code>EncodeIterator</code> used with <code>ElementList.encodeInit</code> and all entries.</p> <ul style="list-style-type: none"> If all entries were encoded successfully, a <code>boolean success</code> parameter setting of true finishes encoding. If encoding of any entry failed, a <code>boolean success</code> parameter setting of false rolls back the encoding process to the last successfully encoded point in the contents. <p>Encode any element entries prior to this call.</p>
decode	<p>Starts decoding an <code>ElementList</code>.</p> <p>This method will decode from the <code>Buffer</code> referred to by the passed-in <code>DecodeIterator</code> and allows the user to pass in local set definitions. If the <code>ElementList</code> contains set-defined data (e.g., <code>ElementListFlags.HAS_SET_DATA</code> 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.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>ElementList</code> without needing to call <code>clear</code>.</p>

Table 89: `ElementList` Methods (Continued)

11.3.2.2 ElementListFlags Values

ELEMENTLISTFLAG VALUES	MEANING
NONE	Indicates that optional flags are not set.
HAS_ELEMENT_LIST_INFO	Indicates the presence of the <code>elementListNum</code> member. This member is provided as part of the initial refresh message on a stream or on the first refresh message after a <code>CLEAR_CACHE</code> command.
HAS_STANDARD_DATA	Indicates that the <code>ElementList</code> contains standard element <code>name</code> , <code>dataType</code> , value-encoded data. You can set this value in addition to <code>ElementListFlags.HAS_SET_DATA</code> if both standard and set-defined data are present in this <code>ElementList</code> . If the <code>ElementList</code> does not have entries, do not set this flag value.
HAS_SET_DATA	<p>Indicates that <code>ElementList</code> contains set-defined data.</p> <ul style="list-style-type: none"> If both standard and set-defined data are present in this <code>ElementList</code>, this value can be set in addition to <code>ElementListFlags.HAS_STANDARD_DATA</code>. If the <code>ElementList</code> does not have entries, do not set this flag value. <p>For more information, refer to Section 11.6.</p>
HAS_SET_ID	Indicates the presence of a <code>setId</code> and determines the set definition to use when encoding or decoding set data on this <code>ElementList</code> . For more information, refer to Section 11.6.

Table 90: ElementListFlags Values

11.3.2.3 ElementEntry Methods

Each `ElementList` can contain multiple `ElementEntries` and each `ElementEntry` can house any `DataTypes`, including primitive types (refer to Section 11.2), set-defined types (refer to Section 11.6), 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	<p>Sets or gets a <code>Buffer</code> containing the <code>name</code> associated with this <code>ElementEntry</code>. Element names are defined outside of the Transport API, typically as part of a domain model specification or dictionary. A <code>name</code> can be empty; however this provides no identifying information for the element.</p> <p>The <code>name</code> buffer allows for content length ranging from 0 bytes to 32,767 bytes.</p>
dataType	<p>Sets or gets the <code>dataType</code>, which defines the <code>DataTypes</code> of this <code>ElementEntry</code>'s contents.</p> <ul style="list-style-type: none"> While encoding, set this to the enumerated value of the target type. While decoding, <code>dataType</code> describes the type of contained data so that the correct decoder can be used. <p>If set-defined data is used, <code>dataType</code> will indicate any specific <code>DataTypes</code> information as defined in the set definition.</p>
encodedData	<p>Sets or gets the encoded content (<code>encodedData</code>) of this <code>ElementEntry</code>. If populated on encode methods, this indicates that data is pre-encoded and <code>encodedData</code> copies while encoding. While decoding, this refers to the encoded <code>ElementEntry</code>'s payload and length data.</p>

Table 91: ElementEntry Methods

METHOD	DESCRIPTION
encode(w/primitiveType)	<p>Encodes an <code>ElementEntry</code> with a primitive data type (e.g. <code>UInt</code>). This method expects the same <code>EncodeIterator</code> used with <code>ElementList.encodeInit</code>. <code>ElementEntry.name</code> and <code>ElementEntry.dataType</code> must be properly populated.</p> <p>Call this method for each <code>primitiveType</code> entry that you want to encode.</p>
encode	<p>Encodes an <code>ElementEntry</code> with pre-encoded data. This method expects the same <code>EncodeIterator</code> used with <code>ElementList.encodeInit</code>. You must properly populate <code>ElementEntry.name</code> and <code>ElementEntry.dataType</code> and also set <code>encodedData</code> with pre-encoded data before calling this method.</p> <p>Call this method for each pre-encoded entry that you want to encode.</p>
encodeBlank	<p>Encodes a blank <code>ElementEntry</code>. This method expects the same <code>EncodeIterator</code> used with <code>ElementList.encodeInit</code>. You must properly populate <code>ElementEntry.name</code> and <code>ElementEntry.dataType</code>.</p> <p>Call this method for each blank entry that you want to encode.</p>
encodeInit	<p>Encodes an <code>ElementEntry</code> from a complex type, such as a container type or an array. This method expects the same <code>EncodeIterator</code> used with <code>ElementList.encodeInit</code>. You must properly populate <code>ElementEntry.name</code> and <code>ElementEntry.dataType</code>.</p> <p>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.</p> <p>Typical use (e.g. encode an element list as a field entry):</p> <ol style="list-style-type: none"> 1. Call <code>ElementEntry.encodeInit</code>. 2. Call one or more encoding methods for the complex type using the same buffer. 3. Call <code>ElementEntry.encodeComplete</code>.
encodeComplete	<p>Completes the encoding of an <code>ElementEntry</code>. This method expects the same <code>EncodeIterator</code> used with <code>ElementList.encodeInit</code>, <code>ElementEntry.encodeInit</code>, and all other entry encoding.</p> <ul style="list-style-type: none"> • If this specific entry is encoded successfully, a <code>boolean success</code> parameter setting of true finishes entry encoding. • If this specific entry fails to encode, a <code>boolean success</code> parameter setting of false rolls back the encoding of only this <code>ElementEntry</code>.
decode	<p>Decodes an <code>ElementEntry</code>. This method expects the same <code>DecodeIterator</code> used with <code>ElementList.decode</code> and populates <code>encodedData</code> with encoded entry contents.</p> <p>After this method returns, you can use the <code>ElementEntry.dataType</code> to invoke the correct contained type's decode methods. Calling <code>ElementEntry.decode</code> again starts decoding the next entry in the <code>ElementList</code> until no more entries are available.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>ElementEntry</code> without using <code>clear</code>.</p>

Table 91: `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 a `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 method. 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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with ElementList.encodeInit. Error Text:
                      %s\n", error.errorId(), error.sysError(), error.text());
}
else
{
    /* elementListInit encoding was successful */
    /* create a single ElementEntry and reuse for each entry */
    ElementEntry elemEntry = CodecFactory.createElementEntry();
    /* stack allocate a time and populate {hour, minute, second, millisecond} */
    Time time = CodecFactory.createTime();
    time.hour(10);
    time.minute(21);
    time.second(16);
    time.millisecond(777);
    Buffer elementEntryName = CodecFactory.createBuffer();

    /* 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)) < CodecReturnCodes.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 27: ElementList Encoding Example

11.3.2.5 ElementList Decoding Examples

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 method. 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.decode(decIter, localSetDefs)) >= CodecReturnCodes.SUCCESS)
{
    /* decode each element entry */
    while ((retCode = elemEntry.decode(decIter)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with ElementEntry.decode. Error
                               Text: %s\n", 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 */
    System.out.printf("Error (%d) (errno: %d) encountered with ElementList.decode. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

```

Code Example 28: 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 (Section 11.2) and value. A **Map** can contain zero to N^7 entries, where zero entries indicate an empty **Map**.

11.3.3.1 Map Methods

A **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. <ul style="list-style-type: none"> You can use the following convenient methods to set specific MapFlags: <code>applyHasKeyFieldId</code>, <code>applyHasPerEntryPermData</code>, <code>applyHasSetDefs</code>, <code>applyHasSummaryData</code>, <code>applyHasTotalCountHint</code>. You can use the following convenient methods to check whether specific MapFlags are set: <code>checkHasKeyFieldId</code>, <code>checkHasPerEntryPermData</code>, <code>checkHasSetDefs</code>, <code>checkHasSummaryData</code>, <code>checkHasTotalCountHint</code>.
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 Thomson Reuters-defined and negative values are user-defined.
containerType	Sets or gets the value (DataTypes) 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 92: **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 <code>encodedSummaryData</code> , which is a <code>TransportBuffer</code> (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 <code>Map</code> (e.g., currency type). The container type of summary data should match the <code>containerType</code> specified on the <code>Map</code> . If <code>encodedSummaryData</code> 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 <code>encodedSetDefs</code> , which is a <code>Buffer</code> (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 <code>Map</code> '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 <code>encodedEntries</code> , which is a <code>Buffer</code> (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 <code>Map</code> payload and length information.
encodeInit	Begins encoding a <code>Map</code> which can include summary data (Section 11.5) and Local Set Definitions (Section 11.6). <ul style="list-style-type: none"> If summary data and set definitions are pre-encoded, you can populate them on the <code>encodedSummaryData</code> and <code>encodedSetDefs</code> prior to calling <code>Map.encodeInit</code>. Additional work is not needed to complete encoding this content. If summary data and set definitions are not pre-encoded, <code>Map.encodeInit</code> performs the <code>Init</code> for these values. You must call the corresponding <code>Complete</code> 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 <code>0</code>. This is required only when content is not pre-encoded.
encodeComplete	Completes the encoding of a <code>Map</code> . This method expects the same <code>EncodeIterator</code> that was used with <code>Map.encodeInit</code> , any summary data, set data, and all entries. <ul style="list-style-type: none"> If encoding was successful, the <code>boolean success</code> parameter should be set to true to finish encoding. If any component failed to encode, the <code>boolean success</code> 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.

Table 92: `Map` Methods (Continued)

METHOD	DESCRIPTION
encodeSummaryDataComplete	<p>Completes encoding of any non-pre-encoded <code>Map</code> summary data.</p> <p>If <code>MapFlags.HAS_SUMMARY_DATA</code> is set and <code>encSodedsummaryData</code> is not populated, summary data is expected after <code>Map.encodeInit</code> or <code>Map.encodeSetDefsComplete</code> returns. This method expects the same <code>EncodeIterator</code> used with previous map encoding methods.</p> <ul style="list-style-type: none"> • If encoding succeeds, the <code>boolean success</code> parameter should be true to finish encoding. • If encoding fails, the <code>boolean success</code> parameter should be set to false to roll back to the last previously successful encoded point in the contents. <p>If both <code>MapFlags.HAS_SUMMARY_DATA</code> and <code>MapFlags.HAS_SET_DEFS</code> are present, then set definitions are expected first, and summary data is encoded after the call to <code>Map.encodeSetDefsComplete</code>.</p>
encodeSetDefsComplete	<p>Completes encoding of any non pre-encoded local set definition data.</p> <p>If <code>MapFlags.HAS_SET_DEFS</code> is set and <code>encSetDefs</code> is not populated, local set definition data is expected after <code>Map.encodeInit</code> returns. This method expects the same <code>EncodeIterator</code> used with <code>Map.encodeInit</code>.</p> <ul style="list-style-type: none"> • If encoding succeeds, the <code>boolean success</code> parameter should be true to finish encoding. • If encoding fails, the <code>boolean success</code> parameter should be set to false to roll back to the last previously successful encoded point in the contents. <p>If both <code>MapFlags.HAS_SUMMARY_DATA</code> and <code>MapFlags.HAS_SET_DEFS</code> are present, set definitions are expected first, while any summary data is encoded after the call to <code>Map.encodeSetDefsComplete</code>.</p>
decode	Begins decoding a <code>Map</code> . This method will decode from the <code>Buffer</code> to which the passed-in <code>DecodeIterator</code> refers.
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>Map</code> without using <code>clear</code>.</p>

Table 92: `Map` Methods (Continued)

11.3.3.2 MapFlags Values

FLAG ENUMERATION	MEANING
HAS_KEY_FIELD_ID	Indicates the presence of the <code>keyFieldId</code> member. <code>keyFieldId</code> should be provided if the key information is also a field that would be contained in the entry payload. This optimization allows <code>keyFieldId</code> to be included once instead of in every entry's payload.
HAS_TOTAL_COUNT_HINT	Indicates the presence of the <code>totalCountHint</code> 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.
HAS_PER_ENTRY_PERM_DATA	Indicates that permission information is included with some map entries. The <code>Map</code> encoding functionality sets this flag value on the user's behalf if any entry is encoded with its own <code>permData</code> . A decoding application can check this flag to determine if any contained entry has <code>permData</code> , often useful for fan out devices (if an entry does not have <code>permData</code> , 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 <code>MapEntryFlags.HAS_PERM_DATA</code> .
HAS_SUMMARY_DATA	Indicates that the <code>Map</code> contains summary data. If this flag is set while encoding, summary data must be provided by encoding or populating <code>encodedSummaryData</code> with pre-encoded information. If this flag is set while decoding, summary data is contained as part of the <code>Map</code> and the user can choose whether to decode it.
HAS_SET_DEFS	Indicates that the <code>Map</code> contains local set definition information. Local set definitions correspond to data contained within this <code>Map</code> 's entries and are used for encoding or decoding their contents. For more information, refer to Section 11.6.

Table 93: 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. <ul style="list-style-type: none"> You can use the following convenient methods to set specific MapEntryFlags: <code>applyHasPermData</code>. You can use the following convenient methods to check whether specific MapEntryFlags are set: <code>checkHasPermData</code>.
action	Sets or gets the entry action 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 a 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, MapEntryFlags.HAS_PERM_DATA should be set. permData has a maximum allowed length of 32,767 bytes. <ul style="list-style-type: none"> 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.
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 94: **MapEntry** Methods

METHOD	DESCRIPTION
encodeInit(w/ keyPrimitiveType)	<p>Encodes a <code>MapEntry</code> from a container type.</p> <p>This method expects the same <code>EncodeIterator</code> used with <code>Map.encodeInit</code>. After this call, you can use housed-type encode methods to encode contained types.</p> <p>The <code>keyPrimitiveType</code> accepts primitive <code>DataTypes</code> (values between 1 and 63), cannot be specified as blank and cannot be the <code>DataTypes.ARRAY</code> or <code>DataTypes.UNKNOWN</code> primitive types. For more information about base primitive types, refer to Section 11.2.</p> <p>You call this method after <code>Map.encodeInit</code> 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.</p>
encodeInit	<p>Encodes a <code>MapEntry</code> with pre-encoded primitive key.</p> <p>This method expects the same <code>EncodeIterator</code> used with <code>Map.encodeInit</code>. After this call, you can use housed-type encode methods to encode contained types.</p> <p>Call this method after <code>Map.encodeInit</code> 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.</p> <p>Set <code>Map.encodedKey</code> with pre-encoded data before calling this method.</p>
encodeComplete	<p>Completes the encoding of a <code>MapEntry</code>.</p> <p>This method expects the same <code>EncodeIterator</code> used with <code>Map.encodeInit</code>, <code>MapEntry.encodeInit</code>, and all other encoding for this container.</p> <ul style="list-style-type: none"> If this specific map entry is encoded successfully, the <code>boolean success</code> parameter should be set to <code>true</code> to finish entry encoding. If this specific entry fails to encode, the <code>boolean success</code> parameter should be set to <code>false</code> to roll back the encoding of only this <code>MapEntry</code>.
decode(keyData)	<p>Decodes a <code>MapEntry</code> and can optionally decode the <code>MapEntry.encodedKey</code>.</p> <p>This method expects the same <code>DecodeIterator</code> used with <code>Map.decode</code>. This populates <code>encodedData</code> with encoded entry contents and <code>encodedKey</code> with the encoded entry key.</p> <p>After this method returns, you can use the <code>Map.containerType</code> to invoke the correct contained-type's decode methods. Calling <code>MapEntry.decode</code> again continues the decoding of the next entry in the <code>Map</code> until no more entries are available.</p> <p><code>keyData</code> can be any valid <code>keyPrimitiveType</code> primitive (e.g. <code>UInt</code>). If <code>keyData</code> is non NULL, the entry key will also be decoded into the specified <code>keyData</code>.</p> <p>As entries are received, the action dictates how to apply contents.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>MapEntry</code> without using <code>clear</code>.</p>

Table 94: `MapEntry` Methods (Continued)

11.3.3.4 MapEntry Flag Enumeration Value

FLAG ENUMERATION	MEANING
HAS_PERM_DATA	Indicates that the container entry includes a <code>permData</code> member and also specifies any authorization information for this entry. For more information, refer to Section 11.4.

Table 95: `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 96: `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 method. 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 )) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Map.encodeInit. Error Text: %s\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 = CodecFactory.createUInt();

    /* 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)) < CodecReturnCodes.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)) < CodecReturnCodes.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 encData 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 29: **MapEntry** 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 a `FieldList` and that the `keyPrimitiveType` is `DataTypes.INT`. This sample also uses the `MapEntry.decode` method 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 method. 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)) >= CodecReturnCodes.SUCCESS)
{
    /* create primitive value to have key decoded into and a single map entry to reuse */
    Int tempInt = CodecFactory.createInt();

    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry)
       indicates to UPA 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)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with MapEntry.decode. Error Text:
                               %s\n", 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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Map.decode. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

```

Code Example 30: 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. <ul style="list-style-type: none"> 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 DataTypes 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 TransportBuffer (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 TransportBuffer (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.

Table 97: **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
encodedEntries	Returns <code>encodedEntries</code> , which is a <code>Buffer</code> (with position and length) that contains all encoded key-value pair encoded data, if any, contained in the message. This refers to encoded <code>Series</code> payload and length data.
encodeInit	<p>Starts encoding a <code>Series</code> 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).</p> <p>You can encode additional summary data, set definitions, or entries after this method returns.</p> <ul style="list-style-type: none"> If summary data or set definitions are pre-encoded, you populate them on the <code>encodedSummaryData</code> and <code>encodedSetDefs</code> prior to calling <code>encodeInit</code>. No additional work is needed to complete the encoding of this content. If summary data or set definitions are not pre-encoded, <code>encodeInit</code> will perform the <code>Init</code> for these components. After this content is encoded, you must call the corresponding <code>Complete</code> methods. <p>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.</p>
encodeComplete	<p>Completes the encoding of a <code>Series</code>. This method expects the same <code>EncodeIterator</code> used with <code>Series.encodeInit</code>, any summary data, set data, and all entries.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish encoding. If encoding fails, the <code>boolean success</code> parameter should be <code>false</code> to roll back the encoding to the last previously successful encoded point in the contents. <p>Encode all series content prior to this call.</p>
encodeSummaryDataComplete	<p>Completes the encoding of any non-pre-encoded <code>Series</code> summary data. If the <code>SeriesFlags.HAS_SUMMARY_DATA</code> flag is set and <code>encodedSummaryData</code> is not populated, summary data is expected after <code>Series.encodeInit</code> or <code>Series.encodeSetDefsComplete</code> returns. This method expects the same <code>EncodeIterator</code> used with previous series encoding methods.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish encoding. If encoding fails, the <code>boolean success</code> parameter should be <code>false</code> to roll back the encoding prior to summary data. <p>If both <code>SeriesFlags.HAS_SUMMARY_DATA</code> and <code>SeriesFlags.HAS_SET_DEFS</code> are present, set definitions are expected first, while any summary data is encoded after the call to <code>encodeSetDefsComplete</code>.</p>

Table 97: `Series` Methods (Continued)

METHOD	DESCRIPTION
encodeSetDefsComplete	<p>Completes the encoding of any non pre-encoded local set definition data. If the <code>SeriesFlags.HAS_SET_DEFS</code> flag is set and <code>encodedSetDefs</code> is not populated, local set definition data is expected after <code>Series.encodeInit</code> returns. This method expects the same <code>EncodeIterator</code> used with <code>Series.encodeInit</code>.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish encoding. If encoding fails, the <code>boolean success</code> parameter should be <code>false</code> to roll back the encoding prior to the set definition data. <p>If both <code>SeriesFlags.HAS_SUMMARY_DATA</code> and <code>SeriesFlags.HAS_SET_DEFS</code> are present, set definitions are expected first, while any summary data is encoded after the call to <code>Series.encodeSetDefsComplete</code>.</p>
decode	Begins decoding a <code>Series</code> from the <code>TransportBuffer</code> specified by <code>DecodeIterator</code> .
clear	<p>Clears the object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>Series</code> without using <code>clear</code>.</p>

Table 97: `Seri es` Methods (Continued)

11.3.4.2 SeriesFlags Values

SERIES FLAG	MEANING
NONE	Indicates that optional flags are not set.
HAS_TOTAL_COUNT_HINT	Indicates the presence of the <code>totalCountHint</code> 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	<p>Indicates that the <code>Series</code> contains summary data.</p> <ul style="list-style-type: none"> If set while encoding, summary data must be provided by encoding or populating <code>encodedSummaryData</code> with pre-encoded information. If set while decoding, summary data is contained as part of <code>Series</code> and the user can choose to decode it.
HAS_SET_DEFS	<p>Indicates that the <code>Series</code> contains local set definition information. Local set definitions correspond to data contained in this <code>Series</code>'s entries and encode or decode their contents.</p> <p>For more information, refer to Section 11.6.</p>

Table 98: `Seri esFl ags` 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 <code>seriesEntry</code> . <ul style="list-style-type: none"> If populated on encode methods, this indicates that data is pre-encoded and <code>encodedData</code> will be copied while encoding. If populated while decoding, this refers to this encoded <code>SeriesEntry</code>'s payload and length data.
encode	Encodes a <code>SeriesEntry</code> from pre-encoded data. This method expects the same <code>EncodeIterator</code> used with <code>Series.encodeInit</code> . You can pass in the pre-encoded series entry payload via <code>SeriesEntry.encodedData</code> . <code>SeriesEntry.encode</code> is called after <code>Series.encodeInit</code> and any summary data and local set definition data is encoded.
encodeInit	Encodes a <code>SeriesEntry</code> from a container type. <code>SeriesEntry.encodeInit</code> expects the same <code>EncodeIterator</code> used with <code>Series.encodeInit</code> . 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 <code>Series.encodeInit</code> 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 <code>SeriesEntry</code> . This method expects the same <code>EncodeIterator</code> used with <code>Series.encodeInit</code> , <code>SeriesEntry.encodeInit</code> , and all other encoding for this container. <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish entry encoding. If encoding of this specific entry fails, the <code>boolean success</code> parameter should be <code>false</code> to roll back the encoding of only this <code>seriesEntry</code>.
decode	Decodes a <code>SeriesEntry</code> . This method expects the same <code>DecodeIterator</code> used with <code>Series.decode</code> and populates <code>encodedData</code> with encoded entry. After <code>SeriesEntry.decode</code> returns, you can use <code>Series.containerType</code> to invoke the correct contained type's decode methods. Calling <code>SeriesEntry.decode</code> again decodes the next entry in the <code>Series</code> 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. <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  Tip: When decoding, you can reuse <code>SeriesEntry</code> without using <code>clear</code>. </div>

Table 99: `SeriesEntry` Methods

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 method. 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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Series.encodeInit. Error Text: %s\n",
                      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 = CodecFactory.createSeriesEntry();
    ElementList elementList = CodecFactory.createElementList();

    /* 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)) < CodecReturnCodes.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)) < CodecReturnCodes.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 31: Series Encoding Example

11.3.4.5 Series Decoding Example

The following sample illustrates how to decode a `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 method. 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)) >= CodecReturnCodes.SUCCESS)
{
    /* create single series entry and reuse while decoding each entry */
    SeriesEntry seriesEntry = CodecFactory.createSeriesEntry();
    /* 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 = CodecFactory.createElementList();
        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)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with SeriesEntry.decode.
                Error Text: %s\n", error.errorId(), error.sysError(), error.text());
        }
        else
        {
            ElementList elementList = CodecFactory.createElementList();
            retCode = elementList.decode(decIter, null);
            /* Continue decoding element entries. See example in Section 11.3.2 */
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    System.out.printf("Error (%d) (errno: %d) encountered with Series.decode. Error Text: %s\n",
        error.errorId(), error.sysError(), error.text());
}

```

Code Example 32: 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 Structure Members

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. <ul style="list-style-type: none"> You can use the following convenient methods to set specific VectorFlags: <code>applyHasPerEntryPermData</code>, <code>applyHasSetDefs</code>, <code>applyHasSummaryData</code>, <code>applyHasTotalCountHint</code>, <code>applySupportsSorting</code>. You can use the following convenient methods to check whether specific VectorFlags are set: <code>checkHasPerEntryPermData</code>, <code>checkHasSetDefs</code>, <code>checkHasSummaryData</code>, <code>checkHasTotalCountHint</code>, <code>checkSupportsSorting</code>.
containerType	Sets or gets the container type (containerType ; a DataTypes 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.

Table 100: **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
encodedEntries	Returns the <code>encodedEntries</code> , which is a <code>Buffer</code> (with position and length) containing the encoded <code>index</code> -value pair encoded data contained in the message. This would refer to encoded <code>Vector</code> payload and length information.
encodeInit	Begins encoding a <code>Vector</code> . 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. <ul style="list-style-type: none"> If summary data and set definitions are pre-encoded, they can be populated on the <code>encodedSummaryData</code> and <code>encodedSetDefs</code> prior to calling <code>Vector.encodeInit</code>. No additional work is needed to complete the encoding of this content. If summary data and set definitions are not pre-encoded, <code>Vector.encodeInit</code> will perform the <code>Init</code> for these components. After encoding this content, the corresponding <code>Complete</code> 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 <code>0</code> can be passed in. This is only needed when the content is not pre-encoded.
encodeComplete	Completes the encoding of a <code>Vector</code> . <p>This method expects the same <code>EncodeIterator</code> used with <code>Vector.encodeInit</code>, any summary data, set data, and all entries.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish encoding. If any component fails to encode, the <code>boolean success</code> parameter should be <code>false</code> to roll back encoding to the last successfully-encoded point in the contents. <p>Vector content should be encoded prior to this call.</p>
encodeSummaryDataComplete	Completes the encoding of <code>vector</code> summary data. <p>If <code>VectorFlags.HAS_SUMMARY_DATA</code> is set and <code>encodedSummaryData</code> is not populated, summary data is expected after <code>Vector.encodeInit</code> or <code>Vector.encodeSetDefsComplete</code> returns. This method expects the same <code>EncodeIterator</code> used with previous vector encoding methods.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish encoding. If any data fail to encode, the <code>boolean success</code> parameter should be <code>false</code> to roll back to the last successfully-encoded point prior to summary data. If both <code>VectorFlags.HAS_SUMMARY_DATA</code> and <code>VectorFlags.HAS_SET_DEFS</code> are present, set definitions are expected first, while summary data is encoded after the call to <code>Vector.encodeSetDefsComplete</code>.
encodeSetDefsComplete	Completes the encoding of local set definition data. If <code>VectorFlags.HAS_SET_DEFS</code> is set and <code>encodedSetDefs</code> is not populated, local set definition data is expected after <code>Vector.encodeInit</code> returns. This method expects the same <code>EncodeIterator</code> used with <code>Vector.encodeInit</code> . <ul style="list-style-type: none"> If set definition data encodes successfully, the <code>boolean success</code> parameter should be <code>true</code> to finish encoding. If set definition data fails to encode, the <code>boolean success</code> parameter should be <code>false</code> to roll back to the last successfully-encoded point prior to set definition data. If both <code>VectorFlags.HAS_SUMMARY_DATA</code> and <code>VectorFlags.HAS_SET_DEFS</code> are present, set definitions are expected first, and then any summary data is encoded after the call to <code>Vector.encodeSetDefsComplete</code>.

Table 100: `Vector` Methods (Continued)

METHOD	DESCRIPTION
decode	Begins decoding a <code>Vector</code> . This method decodes from the <code>TransportBuffer</code> to which the passed-in <code>DecodeIterator</code> refers.
clear	Clears this object, so that you can reuse it.  Tip: When decoding, you can reuse <code>Vector</code> without using <code>clear</code> .

Table 100: `Vector` Methods (Continued)

11.3.5.2 Vector Flag Enumeration Values

VECTOR FLAG	MEANING
NONE	Indicates that optional flags are not set.
HAS_TOTAL_COUNT_HINT	Indicates that the <code>totalCountHint</code> member is present. <code>totalCountHint</code> 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.
HAS_PER_ENTRY_PERM_DATA	Indicates that permission information is included with some vector entries. The <code>Vector</code> encoding functionality sets this flag value on the user's behalf if an entry is encoded with its own <code>permData</code> . A decoding application can check this flag to determine whether a contained entry has <code>permData</code> and is often useful for fan out devices (if an entry does not have <code>permData</code> , 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 <code>VectorEntryFlags.HAS_PERM_DATA</code> . Refer to Section 11.3.5.4.
HAS_SUMMARY_DATA	Indicates that the <code>Vector</code> contains summary data. <ul style="list-style-type: none"> If this flag is set while encoding, summary data must be provided by encoding or populating <code>encodedSummaryData</code> with pre-encoded data. If this flag is set while decoding, summary data is contained as part of <code>Vector</code> and the user can choose whether to decode it.
HAS_SET_DEFS	Indicates that the <code>Vector</code> contains local set definition information. Local set definitions correspond to data contained in this <code>Vector</code> 's entries and are used for encoding or decoding their contents. <p>For more information, refer to Section 11.6.</p>
SUPPORTS_SORTING	Indicates that the <code>Vector</code> may leverage sortable action types. If an <code>Vector</code> 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 <code>Vector</code> 's contents. <p>For more information on proper handling, refer to Section 11.3.5.5.</p>

Table 101: `VectorFlags` Values

11.3.5.3 VectorEntry Structure Members

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 (<code>flags</code>) that indicate whether optional <code>VectorEntry</code> content is present. For more information about <code>VectorEntryFlags</code> values, refer to Section 11.3.5.4. <ul style="list-style-type: none"> • You can use the convenient method <code>applyHasPermData</code> to set specific <code>VectorEntryFlags</code>. • You can use the convenient method <code>checkHasPermData</code> to check whether specific <code>VectorEntryFlags</code> are set.
action	Sets or gets <code>action</code> , which helps to manage change processing rules and informs the consumer of how to apply the entry's data. For specific information about possible <code>action</code> 's associated with an <code>VectorEntry</code> , refer to Section 11.3.5.5.
index	Sets or gets the entry's position (<code>index</code>) in the <code>vector</code> . This value can change over time based on other <code>VectorEntryActions</code> . <code>index</code> has an allowable range of 0 to 1,073,741,823.
permData	(Optional) Sets or gets <code>permData</code> , which is a Buffer (with position and length) that specifies authorization information for this specific entry. If present, the <code>VectorEntry.HAS_PERM_DATA</code> flag should be set. <ul style="list-style-type: none"> • For more information, refer to Section 11.4. • For more information about <code>VectorEntryFlags</code>, refer to Section 11.3.5.4. <code>permData</code> has a maximum allowed length of 32,767 bytes.
encodedData	Sets or gets <code>encodedData</code> , which is a <code>Buffer</code> (with position and length) that contains this <code>VectorEntry</code> 's encoded content. <ul style="list-style-type: none"> • If populated using encode methods, this indicates that data is pre-encoded and <code>encodedData</code> is copied while encoding. • If populated while decoding, this refers to this encoded <code>VectorEntry</code>'s payload and length information.
encode	Encodes a <code>VectorEntry</code> from pre-encoded data. This method expects the same <code>EncodeIterator</code> used with <code>Vector.encodeInit</code> . The pre-encoded vector entry payload can be passed in via <code>VectorEntry.encodedData</code> . This method is called after <code>Vector.encodeInit</code> and after encoding any summary data and local set definition data.

Table 102: `VectorEntry` Methods

METHOD	DESCRIPTION
encodeInit	<p>Encodes a <code>VectorEntry</code> from a container type.</p> <p>This method expects the same <code>EncodeIterator</code> used with <code>Vector.encodeInit</code>. After this call, housed-type encode methods can encode the contained type. This method is called after <code>Vector.encodeInit</code> and after encoding any summary and local set definition data.</p> <p>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.</p>
encodeComplete	<p>Completes the encoding of a <code>VectorEntry</code>.</p> <p>This method expects the same <code>EncodeIterator</code> used with <code>Vector.encodeInit</code>, <code>VectorEntry.encodeInit</code> and all other encoding for this container.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be <code>true</code> to finish entry encoding. If encoding fails, the <code>boolean success</code> parameter should be <code>false</code> to roll back the encoding of this <code>VectorEntry</code> only.
decode	<p>Decodes a <code>VectorEntry</code>. This method expects the same <code>DecodeIterator</code> used with <code>Vector.decode</code> and populates <code>encodedData</code> with an encoded entry. After this method returns, you can use the <code>Vector.containerType</code> to invoke the correct contained type's decode methods.</p> <p>Calling <code>VectorEntry.decode</code> again will continue to decode subsequent entries in <code>Vector</code> until no more entries are available. As entries are received, the action will indicate how to apply their contents.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>VectorEntry</code> without using <code>clear</code>.</p>

Table 102: `VectorEntry` Methods (Continued)

11.3.5.4 VectorEntry Flag Enumeration Value

VECTOR ENTRY FLAG	MEANING
NONE	Indicates that optional flags are not set.
HAS_PERM_DATA	<p>Indicates the presence of the <code>permData</code> member in this container entry and indicates authorization information for this entry.</p> <p>For more information, refer to Section 11.4.</p>

Table 103: `VectorEntryFlags` Values

11.3.5.5 VectorEntryActions Values

ACTION	MEANING
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. <code>VectorEntryActions.SET_ENTRY</code> can apply to both sortable and non-sortable vectors.
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. <code>VectorEntryActions.UPDATE_ENTRY</code> can apply to both sortable and non-sortable vectors.
CLEAR	Indicates that the consumer should remove any stored or displayed information associated with this entry's index position. <code>VectorEntryActions.CLEAR_ENTRY</code> can apply to both sortable and non-sortable vectors. No entry payload is included when the action is a 'clear.'
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).
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 104: `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 method. 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 )) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Vector.encodeInit. Error Text: %s\n",
                      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 = CodecFactory.createVectorEntry();
    Series series = CodecFactory.createSeries();

    /* 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)) < CodecReturnCodes.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)) < CodecReturnCodes.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 33: **Vector** Encoding Example

11.3.5.7 Vector Decoding Example

The following sample illustrates how to decode a `vector` and is structured to decode each entry to the contained value. This sample code assumes the housed container type is a `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)) >= CodecReturnCodes.SUCCESS)
{
    /* create single vector entry and reuse while decoding each entry */
    VectorEntry vectorEntry = CodecFactory.createVectorEntry();
    /* 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)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with VectorEntry.decode. Error
                Text: %s\n", 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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Vector.decode. Error Text: %s\n",
        error.errorId(), error.sysError(), error.text());
}

```

Code Example 34: 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. <ul style="list-style-type: none"> 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 DataTypes 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 . <ul style="list-style-type: none"> If encoding succeeds, the boolean success parameter should be set to true to finish encoding. If any entry fails to encode, the boolean 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 .

Table 105: **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.

METHOD	DESCRIPTION
clear	Clears this object, so that you can reuse it.
	 Tip: When decoding, you can reuse <code>FilterList</code> without using <code>clear</code> .

Table 105: `FilterList` Methods (Continued)

11.3.6.2 FilterList Flag Enumeration Values

FILTER LIST FLAG	MEANING
NONE	Indicates that optional flags are not set.
HAS_TOTAL_COUNT_HINT	Indicates the presence of the <code>totalCountHint</code> member. <code>totalCountHint</code> 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.
HAS_PER_ENTRY_PERM_DATA	Indicates some filter entries include permission information. The <code>FilterList</code> encoding functionality sets this flag value on the user's behalf if any entry is encoded with its own <code>permData</code> . A decoding application can check this flag to determine whether any contained entry has <code>permData</code> , often useful for fan out devices (if entries do not have <code>permData</code> , 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 <code>FilterEntryFlags.HAS_PERM_DATA</code> flag. Refer to Section 11.3.6.4.

Table 106: `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 (<code>flags</code>) that indicate the presence of optional <code>FilterEntry</code> content. For more information about <code>FilterEntryFlags</code> values, refer to Section 11.3.6.4. <ul style="list-style-type: none"> You can use the following convenient methods to set specific <code>FilterEntryFlags</code>: <code>applyHasContainerType</code>, <code>applyHasPermData</code>. You can use the following convenient methods to check whether specific <code>FilterEntryFlags</code> are set: <code>checkHasContainerType</code>, <code>checkHasPermData</code>.
action	Sets or gets <code>action</code> , which helps manage change processing rules and informs the consumer how to apply the information contained in the entry. For specific information about possible <code>action</code> 's associated with an <code>FilterEntry</code> , refer to Section 11.3.6.5.
id	Sets or gets the ID (<code>id</code>) associated with the entry. Each possible <code>id</code> corresponds to a bit-value that can be used with the message key's <code>filter</code> member. This bit-value can be specified on the <code>filter</code> to indicate interest in the <code>id</code> when present in an <code>RequestMsg</code> or to indicate presence of the <code>id</code> when present in other messages. For additional information about the filter, refer to Section 12.1.2. <code>id</code> 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 <code>DataTypes</code> value describing the type of this <code>FilterEntry</code> . If present, the <code>FilterEntryFlags.HAS_CONTAINER_TYPE</code> flag should be set by the user. For more information about <code>FilterEntry</code> flag values, refer to Section 11.3.6.4.
permData	(Optional) Sets or gets <code>permData</code> , which is a <code>Buffer</code> (with position and length) that specifies authorization information for this entry. If <code>permData</code> is present, the user should set the <code>Flags.HAS_CONTAINER_TYPE</code> flag (<code>RSSL_FTEF_HAS_PERM_DATA</code>). <code>permData</code> has a maximum allowed length of 32,767 bytes. <ul style="list-style-type: none"> For more information about <code>FilterEntry</code> flag values, refer to Section 11.3.6.4. For more information, refer to Section 11.4.
encodedData	Sets or gets <code>encodedData</code> , which is a <code>Buffer</code> (with position and length) containing the <code>FilterEntry</code> 's encoded content. <ul style="list-style-type: none"> If populated on encode functions, <code>encodedData</code> indicates that data is pre-encoded, and <code>encodedData</code> will be copied while encoding. If populated while decoding, this refers to this encoded <code>FilterEntry</code>'s payload and length information.

Table 107: `FilterEntry` Methods

METHOD	DESCRIPTION
encode	<p>Encodes a <code>FilterEntry</code> from pre-encoded data. <code>encode</code> expects the same <code>EncodeIterator</code> used with <code>FilterList.encodeInit</code>. The pre-encoded filter entry payload can be passed in via <code>FilterEntry.encodeData</code>. This method can be called after <code>FilterList.encodeInit</code> completes.</p> <p>If this filter entry houses a type other than what is specified in <code>FilterList.containerType</code>, the entry's <code>containerType</code> should be populated to indicate the difference.</p>
encodeInit	<p>Encodes a <code>FilterEntry</code> from a container type. <code>encodeInit</code> expects the same <code>EncodeIterator</code> used with <code>FilterList.encodeInit</code>. After this call, the housed type encode Method can begin to encode the contained type. This method can be called after <code>FilterList.encodeInit</code> is completed.</p> <ul style="list-style-type: none"> To reserve space for encoding, pass in a maximum length hint value (associated with the expected maximum encoded length of this entry) to <code>FilterEntry.encodeInit</code>. 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 <code>FilterList.containerType</code>, the entry's <code>containerType</code> value must indicate the difference.
encodeComplete	<p>Completes the encoding of a <code>FilterEntry</code>. <code>encodeComplete</code> expects the same <code>EncodeIterator</code> used with <code>FilterList.encodeInit</code>, <code>FilterEntry.encodeInit</code> and all other encoding for this container.</p> <ul style="list-style-type: none"> If encoding succeeds, the <code>boolean success</code> parameter should be set to <code>true</code> to finish entry encoding. If encoding fails, the <code>boolean success</code> parameter should be set to <code>false</code> to roll back the encoding of this <code>FilterEntry</code>.
decode	<p>Decodes a <code>FilterEntry</code>. <code>decode</code> expects the same <code>DecodeIterator</code> used with <code>FilterList.decode</code>. This populates <code>encodedData</code> with an encoded entry. As an entry is received, its action indicates how to apply contents.</p> <p>After this method returns, the <code>FilterList.containerType</code> (or <code>FilterEntry.containerType</code> if present) can invoke the correct contained type's decode methods.</p> <p>Calling <code>FilterEntry.decode</code> again decodes the remaining entries in the <code>FilterList</code>.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse <code>FilterEntry</code> without using <code>clear</code>.</p>

Table 107: `FilterEntry` Methods (Continued)

11.3.6.4 FilterEntry Flag Enumeration Values

FILTER ENTRY FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
HAS_PERM_DATA	Indicates the presence of <code>permData</code> in this container entry and indicates authorization information for this entry. For more information, refer to Section 11.4.
HAS_CONTAINER_TYPE	Indicates the presence of <code>containerType</code> in this entry. This flag is used when the entry's <code>containerType</code> differs from the specified <code>FilterList.containerType</code> .

Table 108: FilterEntryFlags Values

11.3.6.5 FilterEntryActions Values

Each entry has an associated `action` which informs the user of how to apply the entry's contents.

ACTION ENUMERATION	MEANING
SET	Indicates that the consumer should set the entry corresponding to this <code>id</code> . A set action typically occurs when an entry is initially provided. Multiple set actions can occur for the same entry <code>id</code> , in which case, any previously received data associated with the entry <code>id</code> 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 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 <code>id</code> , in which case, the update action should be ignored.
CLEAR	Indicates that the consumer should remove any stored or displayed information associated with this entry's <code>id</code> . No entry payload is included when the action is a clear.

Table 109: FilterEntryActions Values

11.3.6.6 FilterEntry 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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with FilterList.encodeInit. Error Text:
                      %s\n", error.errorId(), error.sysError(), error.text());
}
else
{
    /* filterList init encoding was successful */
    /* create a single FilterEntry and reuse for each entry */
    FilterEntry filterEntry = CodecFactory.createFilterEntry();

    /* 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 = CodecFactory.createElementList();
        elementList.applyHasStandardData();
        /* now encode nested container using its own specific encode methods */
        if ((retCode = elementList.encodeInit(encIter, null, 0)) < CodecReturnCodes.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)) < CodecReturnCodes.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 35: 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)) >= CodecReturnCodes.SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    FilterEntry filterEntry = CodecFactory.createFilterEntry();

    /* decode each filter entry until there are no more left */
    while ((retCode = filterEntry.decode(decIter)) != CodecReturnCodes.END_OF_CONTAINER)
    {
        if (retCode < CodecReturnCodes.SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            System.out.printf("Error (%d) (errno: %d) encountered with FilterEntry.decode. Error
                Text: %s\n", 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 */  
    System.out.printf("Error (%d) (errno: %d) encountered with FilterList.decode. Error Text: %s\n",  
        error.errorId(), error.sysError(), error.text());  
}
```

Code Example 36: **FilterList** Decoding Example

11.3.7 Non-RWF Container Types

Transport API messages and container entries allow non-RWF content. Non-RWF content can be:

- A specific type of formatted data such as ANSI Page or XML, where a **DataTypes** value aids in identifying the type.
- A type of customized, user-defined information. You can use **DataTypes**'s range of **225 - 255** to define custom types.

11.3.7.1 Non-RWF Encode Functions

The Transport API provides utility methods to help encode non-RWF 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-RWF type being encoded.

METHOD	DESCRIPTION
EncoderIterator.encodeNonRWFIInit	Uses the EncoderIterator to populate a Buffer with encoding information for the user. Buffer.data contains the backing ByteBuffer . Buffer.position contains the position where encoding begins and Buffer.length contains the number of available bytes for encoding. After this method returns successfully, you can populate this buffer using non-RWF encode methods.
EncoderIterator.encodeNonRWFComplete	Integrates content encoded into Buffer with other pre-encoded information. Buffer.data.position should be set to the position of the last byte encoded prior to this method being called.

Table 110: Non-RWF Type Encode Methods

11.3.7.2 Non-RWF Encoding Example

Note: Do not change the value of **Buffer.data** between calls to **EncoderIterator.encodeNonRWFIInit** and **EncoderIterator.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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Series.encodeInit. Error Text: %s\n",
                      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 = CodecFactory.createSeriesEntry();
/* create an empty buffer for information to be populated into */
Buffer nonRWFBuffer = CodecFactory.createBuffer();

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 37: 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 DACS usage and permission data creation, refer to the *Transport API DACS LOCK Library Reference Manual*.

Permission data can be specified in some Transport API messages. When permission data is included in a **RefreshMsg** or a **StatusMsg**, 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 **StatusMsg** or **RefreshMsg** which contains the new permission data. When permission data is included in an **UpdateMsg**, this generally defines authorization information that applies only to that specific **UpdateMsg**.

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 TREP, all content is restricted (or filtered) based on user permissions.

When content is contributed, permission data in a **PostMsg** is used to permission the user who posts the information. If the payload of the **PostMsg** is another message type with permission data (i.e., **RefreshMsg**), 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 **PostMsg**, the nested message does not need permission data.

11.5 Summary Data

Some Transport API container 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 Section 11.2) 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. **DataTypes** 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., a **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 (Section 11.2), set-defined primitive type enumerations, and container type enumerations (Section 11.3). 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 a [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.

SET-DEFINED PRIMITIVE DATATYPE	BASE PRIMITIVE DATATYPE	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 111: Set-Defined Primitive Types

SET-DEFINED PRIMITIVE DATATYPE	BASE PRIMITIVE DATATYPE	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 <code>float</code> 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 <code>double</code> type. Follows the IEEE 754 specification. This type cannot be represented as blank.
DataTypes.REAL_4RB	DataTypes.REAL	Real	Real.decode	An optimized RWF representation of a decimal or fractional value which typically requires less bytes on the wire than <code>float</code> or <code>double</code> 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 11.2.1.
DataTypes.8RB	DataTypes.REAL	Real	Real.decode	An optimized RWF 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 11.2.1.

Table 111: Set-Defined Primitive Types (Continued)

SET-DEFINED PRIMITIVE DATATYPE	BASE PRIMITIVE DATATYPE	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
DataTypes.DATE_4	DataTypes.DATE	Date	Date.decode	<p>Representation of a date containing month, day, and year values.</p> <p>This value can be represented as blank.</p> <p>For more details on this type, refer to Section 11.2.2.</p>
DataTypes.TIME_3	DataTypes.TIME	Time	Time.decode	<p>Representation of a time containing hour, minute, and second values.</p> <p>This value can be represented as blank.</p> <p>For more details on this type, refer to Section 11.2.3.</p>
DataTypes.TIME_5	DataTypes.TIME	Time	Time.decode	<p>Representation of a time containing hour, minute, second, and millisecond values.</p> <p>This value can be represented as blank.</p> <p>For more details on this type, refer to Section 11.2.3.</p>
DataTypes.DATETIME_7	DataTypes.DATETIME	DateTime	DateTime.decode	<p>Combined representation of date and time. Contains all members of DataTypes.DATE and hour, minute, and second from DataTypes.TIME.</p> <p>This value can be represented as blank.</p> <p>For more details on this type, refer to Section 11.2.4.</p>
DataTypes.DATETIME_9	DataTypes.DATETIME	DateTime	DateTime.decode	<p>Combined representation of date and time. Contains all members of DataTypes.DATERSSL_DT_DATE and all members of DataTypes.TIMERSSL_DT_TIME.</p> <p>This value can be represented as blank.</p> <p>For more details on this type, refer to Section 11.2.4.</p>

Table 111: Set-Defined Primitive Types (Continued)

11.6.2 Set Definition Use

In the 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 FieldSetDefEntries 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 FieldSetDefEntries 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 112: **FieldSetDef** Method

11.6.2.2 FieldSetDefEntry Structure Members

METHOD	DESCRIPTION
fieldId	<p>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 Thomson Reuters-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.</p> <p>fieldId has an allowable range of -32,768 to 32,767 where positive values are Thomson Reuters-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.</p>
dataType	<p>Set or get the dataType, which defines the DataTypes 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.</p> <ul style="list-style-type: none"> 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 DataTypes information as indicated by the Set Definition, where any set-defined primitive type is converted to the corresponding base primitive type. <p>For a map of set-defined primitive types and their corresponding base primitive types, refer to Section 11.6.1.</p>
clear	<p>Clears this object, so that you can reuse it.</p> <p> Tip: When decoding, you can reuse FieldSetDefEntry without using clear.</p>

Table 113: **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	<p>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.</p> <p>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.</p> <p>For more information about how an ElementList indicates use of a set definition, refer to Section 11.3.2.</p>
count	<p>Sets or gets the count, which is the number of ElementSetDefEntries contained in this definition. Each entry defines how to encode or decode an ElementEntry. A set definition is limited to 255 entries.</p> <p>For more information, refer to Section 11.6.2.4.</p>

Table 114: **ElementSetDef** Methods

METHOD	DESCRIPTION
entries	Set or get entries, which is an array of <code>ElementSetDefEntry</code> s. Each entry defines how to encode or decode an <code>ElementEntry</code> . For more information, refer to Section 11.6.2.4.
clear	Clears this object, so that you can reuse it.
	 Tip: When decoding, you can reuse <code>ElementSetDef</code> without using <code>clear</code> .

Table 114: `ElementSetDef` Methods (Continued)

11.6.2.4 ElementSetDefEntry Methods

METHOD	DESCRIPTION
name	Sets or gets the <code>name</code> , which is a <code>Buffer</code> (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 <code>ElementEntry.name</code> with the <code>name</code> expected in the set definition. If <code>name</code> is not used, validation checking is not provided and information might be encoded that does not properly correspond to the definition. When decoding, <code>ElementEntry.name</code> is populated with the information indicated in the set definition. The <code>name</code> buffer allows content length ranging from 0 bytes to 32,767 bytes.
dataType	Sets or gets <code>dataType</code> . When encoding or decoding an entry using this set definition, <code>dataType</code> defines the entry's <code>DataTypes</code> . This can be a base primitive type, a set-defined primitive type, or a container type. <ul style="list-style-type: none"> While encoding, populate <code>ElementEntry.dataType</code> with the base primitive type or container type value that corresponds to the type contained in this definition. While decoding, populate <code>ElementEntry.dataType</code> with the specific <code>DataTypes</code> 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.
	Clears this object, so that you can reuse it.

Table 115: `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, an **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 FieldSetDefs . 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 FieldSetDefEntry that helps manage memory associated with set definition entries for each FieldSetDef . This memory is created by the Transport API and should not be overwritten or a GC will occur. <ul style="list-style-type: none"> 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 116: Local Field SetDefDb Methods

11.6.3.2 LocalElementSetDefDb Methods

`LocalElementSetDefDb` (which represents multiple local element set definitions) has the following methods:

METHOD	DESCRIPTION
definitions	<p>An array containing up to fifteen <code>ElementSetDefs</code>. Each contained element set definition defines a unique <code>setId</code> for use within the container on which this is present.</p> <p>Refer to the encoding example in Section 11.6.3.7 for suggested use.</p> <p> Warning! This memory is created by the Transport API. Do not overwrite this memory or a GC will occur.</p>
entries	<p>An <code>ElementSetDefEntry</code> that helps manage memory associated with set definition entries for each <code>ElementSetDef</code>.</p> <ul style="list-style-type: none"> When decoding, the 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. <p> Warning! This memory is created by the Transport API. Do not overwrite this memory or a GC will occur.</p>
clear	Clears this object, so that you can reuse it.

Table 117: Local ElementSetDefDb Methods

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 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
<code>LocalFieldSetDefDb.encode</code>	Encodes a non-pre-encoded local field set definition database into its own buffer for use with <code>encodedSetDefs</code> or directly into a <code>Map</code> , <code>Vector</code> , or <code>Series</code> . After the container's <code>EncodeInit</code> method, local set definition encoding is expected prior to any summary data or container entries.
<code>LocalElementSetDefDb.encode</code>	Encodes a non-pre-encoded local element set definition database into its own buffer for use with <code>encodedSetDefs</code> or directly into a <code>Map</code> , <code>Vector</code> , or <code>Series</code> . After the containers <code>EncodeInit</code> method, local set definition encoding is expected prior to any summary data or container entries.
<code>Map.encodeSetDefsComplete</code>	<p>Completes encoding non-pre-encoded element or field set definition database content.</p> <p>This applies to local set definition database content on a <code>Map</code>, refer to Section 11.3.3.</p>

Table 118: Local Set Definition Database Encode Methods

ENCODE INTERFACE	DESCRIPTION
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 118: Local Set Definition Database Encode Methods (Continued)

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, 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 119: 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 = CodecFactory.createLocalFieldSetDefDb();

/* create entries arrays */
FieldSetDefEntry[] fieldSetDefEntries = new FieldSetDefEntry[3];

/* Contains BID as Real */
fieldSetDefEntries[0] = CodecFactory.createFieldSetDefEntry();
fieldSetDefEntries[0].dataType(DataTypes.REAL);
fieldSetDefEntries[0].fieldId(22);

/* Contains ASK as an optimized Real */
fieldSetDefEntries[1] = CodecFactory.createFieldSetDefEntry();
fieldSetDefEntries[1].dataType(DataTypes.REAL_8RB);
fieldSetDefEntries[1].fieldId(25);

/* Contains TRADE TIME as an optimized Time */

```

```

fieldSetDefEntries[2] = CodecFactory.createFieldSetDefEntry();
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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Map.encodeInit. Error Text: %s\n",
                      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 38: 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 an [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)) != CodecReturnCodes.END_OF_CONTAINER)
{
    if (retCode < CodecReturnCodes.SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        System.out.printf("Error (%d) (errno: %d) encountered with MapEntry.decode. Error Text: %s\n",
                           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 39: 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 a **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 = CodecFactory.createLocalElementSetDefDb();
/* create entries arrays */
ElementSetDefEntry elementSetDefEntries[] = new ElementSetDefEntry[3];

/* Contains BID as a Real */
elementSetDefEntries[0] = CodecFactory.createElementSetDefEntry();
elementSetDefEntries[0].dataType(DataTypes.REAL);
Buffer bidBuffer = CodecFactory.createBuffer();
bidBuffer.data("BID");
elementSetDefEntries[0].name(bidBuffer);

/* Contains ASK as an optimized Real */
elementSetDefEntries[1] = CodecFactory.createElementSetDefEntry();
elementSetDefEntries[1].dataType(DataTypes.REAL_8RB);
Buffer askBuffer = CodecFactory.createBuffer();
askBuffer.data("ASK");
elementSetDefEntries[1].name(askBuffer);

/* Contains TRADE TIME as an optimized Time */
elementSetDefEntries[2] = CodecFactory.createElementSetDefEntry();
elementSetDefEntries[2].dataType(DataTypes.TIME_3);
Buffer tradeTimeBuffer = CodecFactory.createBuffer();
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)) < CodecReturnCodes.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 */
System.out.printf("Error (%d) (errno: %d) encountered with Series.encodeInit. Error Text: %s\n",
                  error.errorId(), error.sysError(), error.text());
}
else
{
    /* series init encoding was successful */
    SeriesEntry seriesEntry = CodecFactory.createSeriesEntry();
    ElementList elementList = CodecFactory.createElementList();
    ElementEntry elementEntry = CodecFactory.createElementEntry();

    /* 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 = CodecFactory.createBuffer();
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 40: 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 a [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)) != CodecReturnCodes.END_OF_CONTAINER)
{
    if (retCode < CodecReturnCodes.SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        System.out.printf("Error (%d) (errno: %d) encountered with Series.decode. Error Text: %s\n",
                           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 41: Element Set Definition Database Decoding Example

Chapter 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 **RequestMsg** followed by the provider responding with an **RefreshMsg** or **StatusMsg**. 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 Codec Package offers a suite of message 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 Messages in the Transport API Codec.

12.1.1 Common Message Interface

Each Transport API message consists of both unique members and common message methods. The common methods form the **Msg** portion of the message structure, which all other Message interfaces extend.

12.1.1.1 Msg Methods

STRUCTURE MEMBER	DESCRIPTION
msgClass	<p>Required on all messages.</p> <p>Sets or gets the msgClass, which identifies the specific type of a message (e.g. UpdateMsg, RequestMsg). msgClass allows a range from 0 to 31, with all values reserved for use by Thomson Reuters.</p> <p>For more details about the various message classes, refer to Section 12.1.1.2.</p>
domainType	<p>Required on all messages.</p> <p>Sets or gets the domainType, which identifies the specific domain message model type. domainType allows a range from 0 to 255, where Thomson Reuters-defined values are between 0 and 127 and user-defined values are between 128 and 255.</p> <p>The domain model definition is decoupled from the API and domain models are typically defined in a specification document. Domain models defined by Thomson Reuters are specified in the <i>Transport API RDM Usage Guide</i>, and Domain types are defined in com.thomsonreuters.upa.rdm.DomainTypes.</p>
containerType	<p>Required on all messages.</p> <p>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-RWF container type (225 - 255), or no message payload (128).</p> <p>For more details about container type definitions and use, refer to Section 11.3.</p>

Table 120: **Msg Methods**

STRUCTURE MEMBER	DESCRIPTION
msgKey	<p>Required on an RequestMsg and optional on RefreshMsg, StatusMsg, UpdateMsg, GenericMsg, PostMsg, and AckMsg.</p> <p>Returns the <code>msgKey</code>, which houses various attributes that help identify contents flowing within a stream. The <code>msgKey</code>, in conjunction with QoS and <code>domainType</code>, uniquely identifies the stream. The key typically includes naming and service-related information.</p> <p>For more information about the message key and stream identification, refer to Section 12.1.2 and Section 12.1.3.</p>
streamId	<p>Required on all messages.</p> <p>Sets or gets <code>streamId</code>, which specifies a unique, signed-integer identifier associated with all messages flowing within a stream. <code>streamId</code> allows a range from -2,147,483,648 to 2,147,483,647, where:</p> <ul style="list-style-type: none"> Positive values indicate a consumer-instantiated stream (typically via RequestMsg). Negative values indicate a provider-instantiated stream (often associated with NIPs). <p>For more information about stream identification and <code>streamId</code> use, refer to Section 12.1.3.</p>
encodedDataBody	<p>Set or get the <code>encodedDataBody</code>, which is a Buffer (with position and length) containing any encoded data contained in the message. If populated, the content type is described by <code>containerType</code>. <code>encodedDataBody</code> would contain only encoded message payload and length information.</p> <p><code>encDataBody</code> can represent up to 4,294,967,295 bytes of payload. This payload length is typically limited by the contained type's specification.</p> <ul style="list-style-type: none"> When encoding, <code>encodedDataBody</code> refers to any pre-encoded message payload. When decoding, <code>encodedDataBody</code> refers to any encoded message payload.
encodedMsgBuffer	<p>Returns the <code>encodedMsgBuffer</code>, which is a Buffer (with position and length) containing the entire encoding of the message. <code>encodedMsgBuffer</code> would contain both encoded message header and encoded message payload.</p> <p><code>encodedMsgBuffer</code> is typically populated only while decoding, and refers to the entire encoded message header and payload.</p>
extendedHeader	<p>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.</p>
validateMsg	<p>Performs a basic validation on the populated <code>Msg</code> structure (useful when encoding), ensuring that optional members indicated as present are correctly populated (e.g., that length and data are both populated).</p>
isFinalMsg	<p>Returns <code>true</code> if the message is the last message received on a stream, such as:</p> <ul style="list-style-type: none"> The final response to non-streaming requests. A message with a <code>streamState</code> indicating a closed stream (refer to Section 11.2.6). A message that explicitly closes the stream (e.g. closed with a CloseMsg). <p>Returns <code>false</code> if data is to continue streaming.</p>

Table 120: `Msg` Methods (Continued)

STRUCTURE MEMBER	DESCRIPTION
copy	<p>Performs a deep copy of a <code>Msg</code> structure.</p> <p>Expects all memory to be owned and managed by the user.</p> <ul style="list-style-type: none"> • If the memory for the <code>Buffers</code> (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 120: `Msg` Methods (Continued)

12.1.1.2 MsgClasses Values

MSG CLASS VALUE	MESSAGE INTERFACE NAME	DESCRIPTION
REQUEST	RequestMsg	<p>Consumers use RequestMsg to express interest in a new stream or modify some parameters on an existing stream; typically results in the delivery of an RefreshMsg or StatusMsg.</p> <p>For more information, refer to Section 12.2.1.</p>
REFRESH	RefreshMsg	<p>The Interactive Provider can use this class to respond to a consumer's request for information (solicited) or provide a data resynchronization point (unsolicited).</p> <p>The NIP 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.</p> <p>For more information, refer to Section 12.2.2.</p>
UPDATE	UpdateMsg	<p>Interactive or NIPs use the UpdateMsg to convey changes to information on a stream. Update messages typically flow on a stream after delivery of a refresh.</p> <p>For more information, refer to Section 12.2.3.</p>
STATUS	StatusMsg	<p>Indicates changes to the stream or data properties. A provider uses StatusMsg 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.</p> <p>This message can indicate changes:</p> <ul style="list-style-type: none"> • In streamState or dataState • In a stream's permissioning information • To the item group to which the stream belongs
CLOSE	CloseMsg	<p>A consumer uses CloseMsg to indicate no further interest in a stream. As a result, the stream should be closed.</p> <p>For more information, refer to Section 12.2.5.</p>
GENERIC	GenericMsg	<p>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.</p> <p>After a stream is established via a request-refresh/status interaction:</p> <ul style="list-style-type: none"> • A consumer can send this message to a provider. • A provider can send this message to a consumer. • NIPs can send this message to the ADH.

Table 121: MsgClasses Values

MSG CLASS VALUE	MESSAGE INTERFACE NAME	DESCRIPTION
POST	PostMsg	A consumer uses PostMsg to push content upstream. This information can be applied to an Enterprise Platform 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	AckMsg	A provider uses AckMsg to inform a consumer of success or failure for a specific PostMsg or CloseMsg . For more information, refer to Section 12.2.8.

Table 121: [MsgClasses](#) Values (Continued)

12.1.1.3 [MsgClasses](#) Methods

METHOD	DESCRIPTION
toString	Returns a Java String representation of a message interface.  Warning! This method creates garbage.

Table 122: [MsgClasses](#) Methods

12.1.2 [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](#), QoS data, and message key.

12.1.2.1 [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. <ul style="list-style-type: none"> 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.

Table 123: [msgKey](#) Methods

METHOD	DESCRIPTION
serviceId	<p>Sets or gets a service's two-byte, unsigned integer identifier (<code>serviceId</code>); a logical mechanism that provides or enables access to a set of capabilities. <code>serviceId</code> allows a range from 0 to 65,535, with 0 being reserved. This value should correspond to the service content being requested or provided.</p> <p>In the 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 QoS, 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.</p> <p>For details, refer to the <i>Transport API RDM Usage Guide</i>.</p>
nameType	<p>Sets or gets a numeric value (<code>nameType</code>), typically enumerated, that indicates the type of the <code>name</code> member. Examples are User Name or RIC (i.e., the Reuters Instrument Code). <code>nameTypes</code> are defined on a per-domain model basis.</p> <p><code>nameType</code> allows a range from 0 to 255. Name type values and rules are defined within domain message model specifications. Values associated with Thomson Reuters domain models can be found in <code>com.thomsonreuters.upa.rdm.InstrumentNameTypes</code>.</p>
name	<p>Sets or gets the <code>name</code>, which is a <code>Buffer</code> (with position and length) containing the name associated with the contents of the stream. Specific <code>name</code> type and contents should comply with the rules associated with the <code>nameType</code> member.</p> <p><code>name</code> is an <code>Buffer</code> type that allows for a name of up to 255 bytes.</p>
filter	<p>Sets or gets a filter; a combination of up to 32 unique <code>filterId</code> bit-values (where each <code>filterId</code> corresponds to a filter bit-value) that describe content for domain model types with an <code>FilterList</code> payload. Filter identifier values are defined by the corresponding domain model specification.</p> <ul style="list-style-type: none"> When specified in a <code>RequestMsg</code>, <code>filter</code> conveys information which entries to include in responses. When specified on a message housing an <code>FilterList</code> payload, <code>filter</code> conveys information about which filter entries are present. <p>For more information, refer to Section 11.3.6.</p>
addFilterId	<p>Converts a <code>filterId</code> value into the bit-value representation and adds bit-value to the <code>MsgKey.filter</code> member. Used with <code>FilterList</code> container types.</p> <p>For more information, refer to Section 11.3.6.</p>
checkFilterId	<p>Converts a <code>filterId</code> value into the bit-value representation and checks for the bit-value presence in the <code>MsgKey.filter</code> member. Used with <code>FilterList</code> container types.</p> <p>For more information, refer to Section 11.3.6.</p>
identifier	<p>User-specified numeric identifier defined on a per-domain model basis.</p> <p><code>identifier</code> allows a range from -2,147,483,648 to 2,147,483,647.</p> <p>Note: More information should be present as part of the specific domain model definition.</p>
attribContainerType	<p>Sets or gets the content type (<code>attribContainerType</code>) of the <code>msgKey.encodedAttrib</code> information. Can indicate the presence of a container type (value 129 - 224) or some type of customer-defined container type (225 - 255).</p> <p>For more details about container type definitions and use, refer to Section 11.3.</p>

Table 123: `msgKey` Methods (Continued)

METHOD	DESCRIPTION
encodedAttrib	Sets or gets <code>name</code> , which is a <code>Buffer</code> (with position and length) containing additional, encoded, message key attribute information. If populated, contents are described by the <code>attribContainerType</code> member. Additional attribute information typically allows for further uniqueness in the identification of a stream. <code>encodedAttrib</code> is a <code>Buffer</code> that can represent up to 32,767 bytes of information.
equals	Compares this <code>MsgKey</code> to another <code>MsgKey</code> , to determine whether they are the same. Returns <code>true</code> if the keys match; <code>false</code> otherwise.
copy	Performs a deep copy of a <code>MsgKey</code> . Expects all memory to be owned and managed by user. If the memory for the <code>Buffers</code> (i.e. <code>name</code> , <code>attrib</code>) are not provided, they will be created.
clear	Clears this object, so that you can reuse it.  Tip: When decoding, the <code>MsgKey</code> object can be reused without using <code>clear</code> .

Table 123: `msgKey` Methods (Continued)

12.1.2.2 Message Key Flag Enumeration Values

MSG KEY FLAG	MEANING
HAS_SERVICE_ID	Indicates the presence of the <code>serviceId</code> member.
HAS_NAME	Indicates the presence of the <code>name</code> member.
HAS_NAME_TYPE	Indicates the presence of the <code>nameType</code> member.
HAS_FILTER	Indicates the presence of the <code>filter</code> member.
HAS_IDENTIFIER	Indicates the presence of the <code>identifier</code> member.
HAS_ATTRIB	Indicates the presence of the <code>attribContainerType</code> and <code>encodedAttrib</code> members.

Table 124: `MsgKeyFlags` Values

12.1.3 Stream Identification

The 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 stream², 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 NIP application.

For a consumer application, a positive value `streamId` should be specified on any `RequestMsg`, along with the `domainType`, `msgKey` and additional key attributes, and desired QoS information. An interactive provider application should provide a response, typically a `RefreshMsg`, 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 11.2.5.

For an NIP, the initial `RefreshMsg` published for each item should contain `domainType`, message key information, and the QoS being provided for the stream. In addition, the NIP should specify a negative value `streamId` to be associated with the stream for the remainder of the run-time.

12.1.3.1 Stream Comparison

To most efficiently use a connection's bandwidth, Thomson Reuters 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 `RequestMsg` 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 `RefreshMsg` can satisfy the additional request, and any `statusMsg`, `UpdateMsg`, and `GenericMsg` 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 `StatusMsg`. In this case, the `StatusMsg` 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 11.2.6.

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 Section 13.9.

2. `domainType` is present on all messages and cannot be optimized out like quality of service and `msgKey` information.

12.1.3.2 Private Streams

The 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 a TREP 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 **GenericMsgs**).

For more information about private stream instantiation, refer to Section 13.10.

12.1.3.3 Changeable Stream Attributes

A select number of attributes may change during the life of a stream. A consumer can change attributes via a subsequent **RequestMsg** that uses the same **streamId** as previous requests. An Interactive or NIP can change attributes via a subsequent solicited or unsolicited **RefreshMsg**.

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 **RequestMsg** to indicate more or less interest in a stream. For more information, refer to Section 13.2.

If a QoS range is requested, the provided **RefreshMsg** includes only the concrete QoS, which may be different from the best and worst specified. If a **dynamic** QoS is supported, QoS may occasionally change over the life of the stream, however this should stay within the range requested in **RequestMsg**.

An item's identification might also change, which can result in changes to multiple **msgKey** members. Such a case can occur via a **redirect**, a **RefreshMsg** or **StatusMsg** with a **streamState** of **StreamState.REDIRECTED** (for more information on the redirected state value, refer to see Section 11.2.6.2). 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 **RequestMsg**, containing the redirected **msgKey**.

Some **RequestMsg.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 `RequestMsg` interface extends the `Msg` interface. An OMM consumer uses a `RequestMsg` 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 a `RequestMsg` 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 a `RefreshMsg` being delivered to the consumer, though a `StatusMsg` 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, a `StatusMsg` is typically delivered to provide more detailed information to the consumer.

Issuing a `RequestMsg` 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 RequestMsg Methods

METHOD	DESCRIPTION
flags	<p>Sets or gets a combination of bit values (<code>flags</code>) to indicate special behaviors and the presence of optional <code>RequestMsg</code> content.</p> <p>For more information about flag values, refer to Section 12.2.1.2.</p> <ul style="list-style-type: none"> • You can use the following convenient methods to set specific <code>RequestMsgFlags</code>: <code>applyConfInfoInUpdates</code>, <code>applyHasBatch</code>, <code>applyHasExtendedHdr</code>, <code>applyHasPriority</code>, <code>applyHasQos</code>, <code>applyHasView</code>, <code>applyHasWorstQos</code>, <code>applyMsgKeyInUpdates</code>, <code>applyNoRefresh</code>, <code>applyPause</code>, <code>applyPrivateStream</code>, <code>applyStreaming</code>. • You can use the following convenient methods to check whether specific <code>RequestMsgFlags</code> are set: <code>checkConfInfoInUpdates</code>, <code>checkHasBatch</code>, <code>checkHasExtendedHdr</code>, <code>checkHasPriority</code>, <code>checkHasQos</code>, <code>checkHasView</code>, <code>checkHasWorstQos</code>, <code>checkMsgKeyInUpdates</code>, <code>checkNoRefresh</code>, <code>checkPause</code>, <code>checkPrivateStream</code>, <code>checkStreaming</code>.
priority	<p>Returns a <code>Priority</code> object which you can use to set or get the <code>priorityClass</code> and <code>priorityCount</code>.</p> <ul style="list-style-type: none"> • <code>Priority.priorityClass</code> can contain values ranging from 0 to 255. • <code>Priority.priorityCount</code> can contain values ranging from 0 to 65,535. <p>For more information about <code>Priority</code> and its uses, refer to Section 13.2.</p>

Table 125: `RequestMsg` Methods

METHOD	DESCRIPTION
qos	<p>Returns a QoS object which you can use to set or get the allowable QoS for the requested stream.</p> <ul style="list-style-type: none"> When specified without a <code>worstQos</code> 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 <code>worstQos</code>, 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 <code>qos</code> nor <code>worstQos</code> are present on the request, this indicates that any available QoS will satisfy the request. <p>Some components may require <code>qos</code> on initial request and reissue messages. See specific component documentation for details.</p> <ul style="list-style-type: none"> For more information, refer to Section 11.2.5. For specific handling information, refer to Section 13.3.
worstQos	<p>Returns a <code>QoS</code> object which you can use to set or get the least acceptable QoS for the requested stream. When specified with a <code>qos</code> 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.</p> <ul style="list-style-type: none"> For more information, refer to Section 11.2.5. For specific handling information, refer to Section 13.3.

Table 125: RequestMsg Methods (Continued)

12.2.1.2 RequestMsgFlags Values

REQUEST MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
STREAMING	<p>Indicates whether the request is for streaming data.</p> <ul style="list-style-type: none"> If present, the OMM consumer wants to continue to receive changes to information after the initial refresh is complete. If absent, the OMM consumer wants to receive only the refresh, after which the OMM Provider should close the stream. Such a request is typically referred to as a <i>non-streaming</i> or <i>snapshot</i> data request. <p>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.</p> <p>For more information about multi-part message handling, refer to Section 13.1.</p>
NO_REFRESH	<p>Indicates that the consumer application does not require a refresh for this request.</p> <p>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).</p>

Table 126: RequestMsgFlags Values

REQUEST MSG FLAG	MEANING
PAUSE	<p>Indicates that the consumer would like to pause the stream, though this does not guarantee that the stream will pause.</p> <p>To resume data flow, the consumer must send a subsequent request message with the <code>RequestMsgFlags.STREAMING</code> flag set.</p> <p>For more information, refer to Section 13.6.</p>
HAS_PRIORITY	<p>Indicates the presence of the <code>priority</code> member, which contains <code>priorityClass</code> and <code>priorityCount</code> members.</p> <p>For more information about using priority, refer to Section 13.2.</p>
HAS_QOS	<p>Indicates the presence of the <code>qos</code> member.</p> <ul style="list-style-type: none"> For more information, refer to Section 12.2.1.1 and Section 11.2.5. For specific handling information, refer to Section 13.3.
HAS_WORST_QOS	<p>Indicates the presence of the <code>worstQos</code> member.</p> <ul style="list-style-type: none"> For more information, refer to Section 12.2.1.1 and Section 11.2.5. For specific handling information, refer to Section 13.3.
HAS_VIEW	<p>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.</p> <p>For more information about using dynamic views, refer to Section 13.8.</p>
HAS_BATCH	<p>Indicates that the request message payload contains a list of items of interest, all with matching <code>msgKey</code> information.</p> <p>For more information on using batch requests, refer to Section 13.7.</p>
HAS_EXTENDED_HEADER	<p>Indicates that the <code>extendedHeader</code> member is present. Information in the <code>extendedHeader</code> is defined outside of the scope of the Transport API.</p>
MSG_KEY_IN_UPDATES	<p>Indicates that the consumer wants to receive the full <code>msgKey</code> in update messages.</p> <p>This flag does not guarantee that the <code>msgKey</code> 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 <code>msgKey</code> in any <code>UpdateMsg</code>). When specified on a request to ADS, the ADS fulfills the request.</p>
CONF_INFO_IN_UPDATES	<p>Indicates that the consumer wants to receive conflation information in update messages delivered on this stream.</p> <p>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 <code>UpdateMsg</code>).</p> <p>For details about conflation information on update messages, refer to Section 12.2.3.</p>

Table 126: `RequestMsgFlags` Values (Continued)

REQUEST MSG FLAG	MEANING
PRIVATE_STREAM	Requests that the stream be opened as private. For details, refer to Section 13.10.

Table 126: RequestMsgFlags Values (Continued)

12.2.2 Refresh Message Interface

The `RefreshMsg` interface extends the `Msg` interface. `RefreshMsg` is often provided as an initial response or when an upstream source requires a data resynchronization point. A `RefreshMsg` contains payload information along with state, QoS, permissioning, and group information.

- If provided as a response to a `RequestMsg`, 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 `RefreshMsg` 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 a `RefreshMsg`, the `streamId` should match the `streamId` on the corresponding `RequestMsg`. The `msgKey` should be populated with the appropriate stream identifying information, and often matches the `msgKey` of the request. When an OMM NIP sends a `RefreshMsg`, 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 `RefreshMsg`, 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 RefreshMsg Methods

METHOD	DESCRIPTION
flags	<p>Set or get <code>flags</code>, which is a combination of bit values that indicate special behaviors and the presence of optional <code>RefreshMsg</code> content.</p> <p>For more information about flag values, refer to Section 12.2.2.2.</p> <ul style="list-style-type: none"> • You can use the following convenient methods to set specific <code>RefreshMsgFlags</code>: <code>applyClearCache</code>, <code>applyDoNotCache</code>, <code>applyHasExtendedHdr</code>, <code>applyHasMsgKey</code>, <code>applyHasPartNum</code>, <code>applyHasPermData</code>, <code>applyHasPostUserInfo</code>, <code>applyHasQos</code>, <code>applyHasSeqNum</code>, <code>applyPrivateStream</code>, <code>applyRefreshComplete</code>, <code>applySolicited</code>. • You can use the following convenient methods to check whether specific <code>RefreshMsgFlags</code> are set: <code>checkClearCache</code>, <code>checkDoNotCache</code>, <code>checkHasExtendedHdr</code>, <code>checkHasMsgKey</code>, <code>checkHasPartNum</code>, <code>checkHasPermData</code>, <code>checkHasPostUserInfo</code>, <code>checkHasQos</code>, <code>checkHasSeqNum</code>, <code>checkPrivateStream</code>, <code>checkRefreshComplete</code>, <code>checkSolicited</code>.

Table 127: RefreshMsg Methods

METHOD	DESCRIPTION
partNum	Sets or gets the part number (<code>partNum</code>) of this refresh. <code>partNum</code> can contain values ranging from 0 to 32,767 where a value of 0 indicates the initial part of a refresh. <ul style="list-style-type: none"> On multi-part refresh messages, <code>partNum</code> 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 <code>partNum</code> of 0 should be used.
seqNum	Sets or gets a user-defined sequence number (<code>seqNum</code>), which allows for values ranging from 0 to 4,294,967,295. <code>seqNum</code> 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 <code>seqNum</code> .
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. <ul style="list-style-type: none"> For details about state information, refer to Section 11.2.6. For a decision table that provides example behavior for various state combinations, refer to Appendix A.
qos	Returns a Qos object which you can use to set or get the concrete QoS of the stream. If a range was requested by the <code>RequestMsg</code> , the <code>qos</code> should fall somewhere in this range, otherwise <code>qos</code> should exactly match what was requested. <ul style="list-style-type: none"> For more details on QoS, refer to Section 11.2.5. For specific handling information, refer to Section 13.3.
permData	Optional. Sets or gets <code>permData</code> , which is a <code>Buffer</code> (with position and length) that specifies authorization information for this stream. <code>permData</code> has a maximum allowed length of 32,767 bytes. When <code>permData</code> is specified on an <code>RefreshMsg</code> , this indicates authorization information for all content on the stream, unless additional permission information is provided with specific content (e.g., <code>MapEntry.permData</code>). For more information, refer to Section 11.4.
groupId	Sets or gets <code>groupId</code> , which is a <code>Buffer</code> (with position and length) containing information about the item group to which this stream belongs. The <code>groupId Buffer</code> has a maximum allowed length of 255 bytes. You can change the associated <code>groupId</code> via a subsequent <code>StatusMsg</code> or <code>RefreshMsg</code> . 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 <code>PostUserInfo</code> object which can be used to set or get information that identifies the user posting this information. If present on an <code>RefreshMsg</code> , this implies that the refresh was posted to the system by the user described in <code>postUserInfo</code> . <ul style="list-style-type: none"> For more information about posting, refer to Section 13.9. For more information about the Visible Publisher Identifier (VPI), refer to Section .

Table 127: `RefreshMsg` Methods (Continued)

12.2.2.2 RefreshMsgFlags Values

REFRESH MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
REFRESH_COMPLETE	Indicates that the message is the final part of the <code>RefreshMsg</code> . This flag value should be set when: <ul style="list-style-type: none"> 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.
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.
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.
CLEAR_CACHE	Indicates that the stream's stored payload information should be cleared. This is typically set by providers when: <ul style="list-style-type: none"> Sending the initial solicited <code>RefreshMsg</code>. Sending the first part of a multi-part <code>RefreshMsg</code>. Some portion of data is known to be invalid.
HAS_MSG_KEY	Indicates that the <code>RefreshMsg</code> contains a populated <code>msgKey</code> . This can aid in associating a request with its corresponding refresh or identify an item sent from an NIP application.
HAS_QOS	Indicates the presence of the <code>qos</code> member. For specific handling information, refer to Section 13.3.
HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
HAS_PART_NUM	Indicates the presence of the <code>partNum</code> member.
HAS_PERM_DATA	Indicates the presence of the <code>permData</code> member.
HAS_POST_USER_INFO	Indicates that this message includes <code>postUserInfo</code> , implying that this <code>RefreshMsg</code> was posted by the user described in <code>postUserInfo</code> .
HAS_EXTENDED_HEADER	Indicates the presence of the <code>extendedHeader</code> member.
PRIVATE_STREAM	Acknowledges the initial establishment of a private stream or, when combined with a <code>streamState</code> value of <code>StreamStates.REDIRECTED</code> , indicates that a stream can only be opened as private. For details, refer to Section 13.10.

Table 128: RefreshMsgFlags Values

12.2.3 Update Message Interface

The [UpdateMsg](#) interface extends the [Msg](#) interface. Providers (both interactive and non-interactive) use [UpdateMsg](#) 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 11.2.5), 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 [UpdateMsg](#) via optional conflation information.

12.2.3.1 UpdateMsg Methods

METHOD	DESCRIPTION
flags	<p>Sets or gets a combination of bit values (flags) that indicate special behaviors and the presence of optional content.</p> <p>For more information about flag values, refer to Section 12.2.3.2.</p> <ul style="list-style-type: none"> 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	<p>Sets or gets the type of data (updateType) in the UpdateMsg, 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.</p> <ul style="list-style-type: none"> Domain message model specifications define available update types. For Thomson Reuters's provided domain models, com.thomsonreuters.upa.rdm.UpdateEventTypes defines available update types.
seqNum	<p>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.</p> <p>Details about sequence number use should be defined within the domain model specification or any documentation for products which require the use of seqNum.</p>
conflationCount	<p>Sets or gets the conflationCount. When conflating data, this value indicates the number of updates conflated or aggregated into this UpdateMsg.</p> <p>conflationCount allows for values ranging from 1 to 32,767.</p>
conflationTime	<p>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 UpdateMsg (typically in milliseconds; for further details, refer to specific component documentation).</p> <p>conflationTime allows for values ranging from 1 to 65,535.</p>

Table 129: [UpdateMsg](#) Methods

METHOD	DESCRIPTION
permData	<p>Optional. Sets or gets <code>permData</code>, which is a <code>Buffer</code> (with position and length) that specifies authorization information for this stream. When specified, <code>permData</code> indicates authorization information for only the content within this message, though this can be overridden for specific content within the message (e.g., <code>MapEntry.permData</code>).</p> <p><code>permData</code> has a maximum allowed length of 32,767 bytes.</p> <p>For more information, refer to Section 11.4.</p>
postUserInfo	<p>Optional. Returns a <code>PostUserInfo</code> object that you can use to set or get information that identifies</p> <ul style="list-style-type: none"> • For more information about posting, refer to Section 13.9. • For more information about the Visible Publisher Identifier, refer to Section .

Table 129: `UpdateMsg` Methods (Continued)

12.2.3.2 `UpdateMsgFlags` Values

UPDATE MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
DISCARDABLE	Indicates that this update can be discarded. Common for options with no open interest.
DO_NOT_CACHE	Indicates that payload information associated with this message should not be cached. <code>UpdateMsgFlags.DO_NOT_CACHE</code> applies only to the message on which it is present.
DO_NOT_CONFLATE	Indicates that this message should not be conflated. This flag value only applies to the message on which it is present.
DO_NOT_RIPPLE	Indicates that the contents of this message should not be rippled. Rippling is typically associated with a <code>FieldList</code> . For additional information, refer to Section 11.3.1.4.
HAS_MSG_KEY	Indicates that the <code>UpdateMsg</code> contains a populated <code>msgKey</code> . The additional key information can help associate a request with updates or identify an item being sent from an NIP application. This information is typically not necessary in an <code>UpdateMsg</code> as the <code>streamId</code> can be used to determine the same information with less bandwidth cost.
HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
HAS_CONF_INFO	Indicates the presence of <code>conflationTime</code> and <code>conflationCount</code> information.
HAS_PERM_DATA	Indicates the presence of the <code>permData</code> member.
HAS_POST_USER_INFO	Indicates that this message includes <code>postUserInfo</code> , implying that this <code>UpdateMsg</code> was posted by the user described in the <code>postUserInfo</code> .
HAS_EXTENDED_HEADER	Indicates the presence of the <code>extendedHeader</code> member.

Table 130: `UpdateMsgFlags` Values

12.2.4 Status Message Interface

The **StatusMsg** interface extends the **Msg** interface. A **StatusMsg** can convey changes in **streamState** or **dataState** (refer to Section 11.2.6), changes in a stream's permissioning information (refer to Section 10.4), or changes to the item group of which the stream is a part (refer to Section 13.4). A Provider application uses **StatusMsg** to close streams to a consumer, in conjunction with an initial request or later after the stream has been established. A **StatusMsg** 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 **GenericMsgs**).

12.2.4.1 StatusMsg Methods

METHOD	DESCRIPTION
flags	<p>Sets or gets a combination of bit values (flags) indicating special behaviors and the presence of optional content.</p> <p>For more information about flag values, refer to Section 12.2.4.2.</p> <ul style="list-style-type: none"> • You can use the following convenient methods to set specific statusMsgFlags: <code>applyClearCache</code>, <code>applyHasExtendedHdr</code>, <code>applyHasGroupId</code>, <code>applyHasMsgKey</code>, <code>applyHasPermData</code>, <code>applyHasPostUserInfo</code>, <code>applyHasState</code>, <code>applyPrivateStream</code>. • You can use the following convenient methods to check whether specific statusMsgFlags are set: <code>checkClearCache</code>, <code>checkHasExtendedHdr</code>, <code>checkHasGroupId</code>, <code>checkHasMsgKey</code>, <code>checkHasPermData</code>, <code>checkHasPostUserInfo</code>, <code>checkHasState</code>, <code>checkHasPrivateStream</code>.
state	<p>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.</p> <ul style="list-style-type: none"> • For details about state information, refer to Section 11.2.6. • For a decision table that provides example behavior for various state combinations, refer to Appendix A.
permData	<p>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., <code>MapEntry.permData</code>). permData allows a maximum length of 32,767 bytes.</p> <p>For more information, refer to Section 11.4.</p>
groupId	<p>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.</p> <p>A subsequent StatusMsg or RefreshMsg can change the item group's associated groupId, while group status notifications can change the state of an entire group of items.</p> <p>For more information about item groups, refer to Section 13.4.</p>
postUserInfo	<p>Optional. Returns a PostUserInfo object that you can use to set or get information that identifies the user who posted this information.</p> <ul style="list-style-type: none"> • For more information about posting, refer to Section 13.9. • For more information about Visible Publisher Identifier, refer to Section .

Table 131: **StatusMsg** Methods

12.2.4.2 StatusMsgFlags Values

STATUS MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
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.
HAS_MSG_KEY	Indicates that the <code>StatusMsg</code> contains a populated <code>msgKey</code> . The <code>msgKey</code> can be used to aid in associating a request to a status message or identify an item sent from an NIP application.
HAS_STATE	Indicates the presence of <code>state</code> information. If <code>state</code> information is not present, the message might be changing the stream's permission information or <code>groupId</code> .
HAS_PERM_DATA	Indicates the presence of <code>permData</code> . When present, the message might be changing the stream's permission information.
HAS_GROUP_ID	Indicates the presence of <code>groupId</code> . When present, the message might be changing the stream's <code>groupId</code> .
HAS_POST_USER_INFO	Indicates the presence of <code>postUserInfo</code> , which identifies the user who posted the <code>statusMsg</code> .
HAS_EXTENDED_HEADER	Indicates the presence of <code>extendedHeader</code> .
PRIVATE_STREAM	Acknowledges the establishment of a private stream, or when combined with a <code>streamState</code> value of <code>streamStates.REDIRECTED</code> , indicates that a stream can be opened only as private. For details, refer to Section 13.10.

Table 132: `StatusMsgFlags` Values

12.2.5 Close Message Interface

The `CloseMsg` interface extends the `Msg` interface. A consumer uses `closeMsg` to indicate no further interest in an item stream and to close the stream. The `streamId` indicates the item stream to which `CloseMsg` applies.

12.2.5.1 CloseMsg Methods

METHOD	DESCRIPTION
flags apply* check*	Sets or gets a combination of bit values (<code>flags</code>) that indicate special behaviors and the presence of optional content. For available flag values, refer to <code>CloseMsgFlags</code> in Section 12.2.5.2. <ul style="list-style-type: none"> You can use the following convenient methods to set specific <code>StatusMsgFlags</code>: <code>applyAck</code>, <code>applyHasExtendedHdr</code>. You can use the following convenient methods to check whether specific <code>StatusMsgFlags</code> are set: <code>checkAck</code>, <code>checkHasExtendedHdr</code>.

Table 133: `CloseMsg` Methods

12.2.5.2 CloseMsgFlags Values

CLOSE MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
ACK	If present, the consumer wants the provider to send an <code>AckMsg</code> to indicate that the <code>CloseMsg</code> 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.
HAS_EXTENDED_HEADER	Indicates the presence of <code>extendedHeader</code> .

Table 134: `CloseMsgFlags` Values

12.2.6 Generic Message Class

The `GenericMsg` interface extends the `Msg` interface. `GenericMsg` 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 `GenericMsgs` to one another, and NIP applications can leverage them. Generic messages are transient and typically not cached by Enterprise Platform components.

The `msgKey` of an `GenericMsg` 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	<p>Sets or gets a combination of bit values (<code>flags</code>) that indicate special behaviors and the presence of optional content.</p> <p>For more information about flag values, refer to Section 12.2.6.2.</p> <ul style="list-style-type: none"> You can use the following convenient methods to set specific <code>GenericMsgFlags</code>: <code>applyHasExtendedHdr</code>, <code>applyHasMsgKey</code>, <code>applyHasPartNum</code>, <code>applyHasPermData</code>, <code>applyHasSecondarySeqNum</code>, <code>applyHasSeqNum</code>, <code>applyMessageComplete</code>. You can use the following convenient methods to check whether specific <code>GenericMsgFlags</code> are set: <code>checkHasExtendedHdr</code>, <code>checkHasMsgKey</code>, <code>checkHasPartNum</code>, <code>checkHasPermData</code>, <code>checkHasSecondarySeqNum</code>, <code>checkHasSeqNum</code>, <code>checkMessageComplete</code>.
partNum	<p>Sets or gets the part number (<code>partNum</code>) of this generic message, typically used with multi-part generic messages. <code>partNum</code> can contain values ranging from 0 to 32,767, where a value of 0 indicates the initial part of a refresh.</p> <ul style="list-style-type: none"> If sent on a single-part post message, use a <code>partNum</code> of 0. On multi-part post messages, use a <code>partNum</code> of 0 on the initial part and increment <code>partNum</code> in each subsequent part by 1.
seqNum	<p>Sets or gets a user-defined sequence number (<code>seqNum</code>) ranging in value from 0 to 4,294,967,295. A <code>seqNum</code> typically corresponds to the sequencing of this message.</p> <p>To help with temporal ordering, <code>seqNum</code> should increase across messages, but can have gaps depending on the sequencing algorithm in use. Details about using <code>seqNum</code> should be defined in the domain model specification or the documentation for products that must use <code>seqNum</code>.</p>
secondarySeqNum	<p>Sets or gets an additional user-defined sequence number (<code>secondarySeqNum</code>) ranging in value from 0 to 4,294,967,295. When using <code>GenericMsg</code> on a stream in a bi-directional manner, <code>secondarySeqNum</code> is often used as an acknowledgment sequence number.</p> <p>For example, a consumer sends a generic message with <code>seqNum</code> populated to indicate the sequence of this message in the stream and <code>secondarySeqNum</code> set to the <code>seqNum</code> last received from the provider. This effectively acknowledges all messages received up to that point while still sending additional information.</p> <p>Sequence number use should be defined within the domain model specification or any documentation for products that use <code>secondarySeqNum</code>.</p>

Table 135: `GenericMsg` Methods

METHOD	DESCRIPTION
permData	<p>Optional. Sets or gets <code>permData</code>, which is a <code>Buffer</code> (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. <code>MapEntry.permData</code>).</p> <p><code>permData</code> allows a maximum length of 32,767 bytes.</p> <p>For more information, refer to Section 11.4.</p>

Table 135: GenericMsg Methods (Continued)

12.2.6.2 GenericMsgFlags Values

GENERIC MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
MESSAGE_COMPLETE	<p>When set, this flag indicates that the message is the final part of an <code>GenericMsg</code>. This flag should be set on:</p> <ul style="list-style-type: none"> 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.
HAS_MSG_KEY	<p>Indicates the presence of a populated <code>msgKey</code>.</p> <p>Use of a <code>msgKey</code> differentiates a generic message from the <code>msgKey</code> information specified for other messages within the stream. Contents and semantics associated with an <code>GenericMsg.msgKey</code> should be defined by the domain model specification that employs them.</p>
HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
HAS_SECONDARY_SEQ_NUM	Indicates the presence of the <code>secondarySeqNum</code> member.
HAS_PART_NUM	Indicates the presence of the <code>partNum</code> member.
HAS_PERM_DATA	Indicates the presence of the <code>permData</code> member.
HAS_EXTENDED_HEADER	Indicates presence of the <code>extendedHeader</code> member.

Table 136: GenericMsgFlags Values

12.2.7 Post Message Interface

The `PostMsg` interface extends the `Msg` interface. A consumer application uses `PostMsg` to push content to upstream components. Such content can be applied to a TREP 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. `PostMsg` 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 Post Msg Methods

METHOD	DESCRIPTION
flags	<p>Sets or gets a combination of bit values (<code>flags</code>) that indicate special behaviors and the presence of optional content.</p> <p>For more information about flag values, refer to Section 12.2.7.2.</p> <ul style="list-style-type: none"> You can use the following convenient methods to set specific <code>PostMsgFlags</code>: <code>applyAck</code>, <code>applyHasExtendedHdr</code>, <code>applyHasMsgKey</code>, <code>applyHasPartNum</code>, <code>applyHasPermData</code>, <code>applyHasPostId</code>, <code>applyHasPostUserRights</code>, <code>applyHasSeqNum</code>, <code>applyPostComplete</code>. You can use the following convenient methods to check whether specific <code>PostMsgFlags</code> are set: <code>checkAck</code>, <code>checkHasExtendedHdr</code>, <code>checkHasMsgKey</code>, <code>checkHasPartNum</code>, <code>checkHasPermData</code>, <code>checkHasPostId</code>, <code>checkHasPostUserRights</code>, <code>checkHasSeqNum</code>, <code>checkPostComplete</code>.
partNum	<p>Sets or gets the part number for this post message, typically used with multi-part post messages. <code>partNum</code> can contain values ranging from 0 to 32,767, where a value of 0 indicates the initial part of a refresh.</p> <ul style="list-style-type: none"> If sent on a single-part post message, use a <code>partNum</code> of 0. On multi-part post messages, use a <code>partNum</code> of 0 on the initial part and in each subsequent part, increment <code>partNum</code> part by 1.
postId	<p>Sets or gets the consumer-assigned identifier (<code>postId</code>), which can range in value from 0 to 4,294,967,295. <code>postId</code> distinguishes different post messages. In multi-part post messages, each part must use the same <code>postId</code> value.</p>
seqNum	<p>Sets or gets a user-defined sequence number (<code>seqNum</code>), typically corresponding to the sequencing of the message. <code>seqNum</code> allows for values ranging from 0 to 4,294,967,295. To help with temporal ordering, <code>seqNum</code> should increase, though gaps might exist depending on the sequencing algorithm in use. Details about <code>seqNum</code> use should be defined in the domain model specification or any documentation for products that use <code>seqNum</code>. When acknowledgments are requested, the <code>seqNum</code> will be provided back in the <code>AckMsg</code> to help identify the <code>PostMsg</code> being acknowledged.</p>
permData	<p>Optional. Sets or gets <code>permData</code>, which is a <code>Buffer</code> (with position and length) that specifies authorization information for content in this message only. <code>permData</code> can be overridden for specific content within the message (e.g. <code>MapEntry.permData</code>).</p> <p><code>permData</code> allows a maximum length of 32,767 bytes.</p> <p>For more information, refer to Section 11.4.</p>

Table 137: `PostMsg` Methods

METHOD	DESCRIPTION
postUserInfo	Returns a <code>PostUserInfo</code> object which can set or get information that identifies the posting user. <code>postUserInfo</code> can optionally be provided along with posted content via a <code>RefreshMsg</code> , <code>UpdateMsg</code> , and <code>StatusMsg</code> . <ul style="list-style-type: none"> For more information about posting, refer to Section 13.9. For more information about Visible Publisher Identifier, refer to Section .
postUserRights	Conveys the rights or abilities of the user posting this content, which can indicate whether the user is permissioned to: <ul style="list-style-type: none"> Create items in the cache of record, Delete items from the cache of record, or Modify the <code>permData</code> on items already present in the cache of record. For details about different rights, refer to Section 12.2.7.3.

Table 137: `PostMsg` Methods (Continued)

12.2.7.2 PostMsgFlags Values

POST MSG FLAG	MEANING
NONE	Indicates that none of the optional flags are set.
POST_COMPLETE	Indicates that this is the final part of the <code>PostMsg</code> . This flag should be set on: <ul style="list-style-type: none"> 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.
ACK	Specifies that the consumer wants the provider to send an <code>AckMsg</code> to indicate that the <code>PostMsg</code> was processed properly. When acknowledging a <code>PostMsg</code> , the provider must include the <code>postId</code> in the <code>ackId</code> and communicate any associated <code>seqNum</code> .
HAS_MSG_KEY	Indicates that the <code>PostMsg</code> contains a populated <code>msgKey</code> that identifies the stream on which the information is posted. A <code>msgKey</code> 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.
HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
HAS_POST_ID	Indicates the presence of the <code>postId</code> member.
HAS_POST_USER_RIGHTS	Indicates the presence of the <code>postUserRights</code> member.
HAS_PART_NUM	Indicates the presence of the <code>partNum</code> member.
HAS_PERM_DATA	Indicates the presence of the <code>permData</code> member.
HAS_EXTENDED_HEADER	Indicates the presence of the <code>extendedHeader</code> member.

Table 138: `PostMsgFlags` Values

12.2.7.3 PostUserRights Values

POST USER RIGHT	MEANING
NONE	The user has no additional posting abilities.
CREATE	The user is allowed to create items in the cache of record.
DELETE	The user is allowed to remove items from the cache of record.
MODIFY_PERM	The user is allowed to modify the <code>permData</code> associated with items already in the cache of record.

Table 139: `PostUserRights` Values

12.2.7.4 PostUserInfo Methods

METHOD	DESCRIPTION
userId	Sets or gets the <code>userId</code> , which identifies the specific user that posted this data.
userAddr	Sets or gets the IP Address (<code>userAddr</code>) of the user that posted this data. Though the address can be specified as either a <code>long</code> or <code>String</code> (e.g. "127.0.0.1"), if it is specified as a <code>String</code> , it will be converted to its <code>integer</code> equivalent.
userAddrToString	Converts an IP address in integer format to its string equivalent.
clear	Clears the object, so that it can be reused.

Table 140: `PostUserRights` Methods

12.2.8 Acknowledgment Message Interface

The `AckMsg` interface extends the `Msg` interface. A provider can send an `AckMsg` 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 `PostMsg` or `CloseMsg`.

12.2.8.1 AckMsg 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. <ul style="list-style-type: none"> You can use the following convenient methods to set specific <code>AckMsgFlags</code>: <code>applyHasExtendedHdr</code>, <code>applyHasMsgKey</code>, <code>applyHasNakCode</code>, <code>applyHasSeqNum</code>, <code>applyHasText</code>, <code>applyPrivateStream</code>. You can use the following convenient methods to check whether specific <code>AckMsgFlags</code> are set: <code>checkHasExtendedHdr</code>, <code>checkHasMsgKey</code>, <code>checkHasNakCode</code>, <code>checkHasSeqNum</code>, <code>checkHasText</code>, <code>checkPrivateStream</code>.
ackId	Sets or gets <code>ackId</code> , which associates the <code>AckMsg</code> with the message it acknowledges. <code>ackId</code> allows for values ranging from 0 to 4,294,967,295. When acknowledging a <code>PostMsg</code> , <code>ackId</code> typically matches the post message's <code>postId</code> .
seqNum	Sets or gets <code>seqNum</code> , which specifies a user-defined sequence number, ranging in value from 0 to 4,294,967,295. To help with temporal ordering, <code>seqNum</code> should increase, though gaps might exist depending on the sequencing algorithm in use. The acknowledgment message may populate this with the <code>seqNum</code> of the <code>PostMsg</code> being acknowledged. This helps correlate the message being acknowledged when the <code>postId</code> alone is not sufficient (e.g., multi-part post messages).
nakCode	Sets or gets <code>nakCode</code> . If present, this message indicates a NAK. The <code>nakCode</code> is an enumerated code value (ranging in value from 1 to 255) that provides additional information about the reason for the NAK. <code>nakCode</code> values are defined in Section 12.2.8.3
text	Optional. Sets or gets <code>text</code> , which is a <code>Buffer</code> (with position and length) that provides additional information about the acceptance or rejection of the message being acknowledged. <code>text</code> has a maximum allowed length of 65,535 bytes.

Table 141: `AckMsg` Methods

12.2.8.2 AckMsgFlags Values

ACK MSG VALUE	MEANING
NONE	Indicates that none of the optional flags are set.
HAS_MSG_KEY	Indicates the presence of a populated <code>msgKey</code> . When present, this is typically populated to match the information being acknowledged.
HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.

Table 142: `AckMsgFlags` Values

ACK MSG VALUE	MEANING
HAS_NAK_CODE	Indicates the presence of the <code>nakCode</code> member.
HAS_TEXT	Indicates the presence of the <code>text</code> member.
HAS_EXTENDED_HEADER	Indicates presence of the <code>extendedHeader</code> member.
PRIVATE_STREAM	Acknowledges the initial establishment of a private stream. For details, refer to Section 13.10.

Table 142: `AckMsgFlags` Values (Continued)

12.2.8.3 NakCodes Values

NAK CODES VALUE	DESCRIPTION
NONE	Indicates that none of the optional flags are set.
ACCESS_DENIED	The user is not permissioned to post on the item or service.
DENIED_BY_SRC	The source being posted to has denied accepting this post message.
SOURCE_DOWN	The source being posted to is down or unavailable.
SOURCE_UNKNOWN	The source being posted to is unknown and unreachable.
NO_RESOURCES	Some component along the path of the post message does not have appropriate resources available to continue processing the post.
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.
GATEWAY_DOWN	A gateway device for handling posted or contributed information is down or unavailable.
SYMBOL_UNKNOWN	The system does not recognize the item information provided with the post message. This may be an invalid item.
NOT_OPEN	The item being posted to does not have an available stream.
INVALID_CONTENT	The content of the post message is invalid (it does not match the expected formatting) and cannot be posted.

Table 143: `AckMsgNakCodes` Values

12.2.9 Msg Encoding and Decoding

All message interfaces (e.g. `RequestMsg`, `RefreshMsg`, etc.) extend the `Msg` interface.

12.2.9.1 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.

`Msg` can be encoded from pre-encoded data or by encoding individual pieces of data as they are provided.

Encode Interface	Description
encode	<p>Encodes a message where all message content is pre-encoded.</p> <ul style="list-style-type: none"> • <code>msgKey</code> attribute information should be encoded and populated on <code>msgKey.encodedAttrib</code> prior to this call. • <code>extendedHeader</code> information should be encoded and populated on the message's <code>extendedHeader</code> member prior to this call. • Message payload information should be encoded and populated on the <code>encodedDataBody</code> member prior to this call.
encodeInit	<p>Begins encoding of an <code>Msg</code>.</p> <p>All message header elements should be properly populated. The <code>containerType</code> member should be populated with the specific type of message payload.</p> <ul style="list-style-type: none"> • If encoding <code>msgKey</code> attribute information: pre-encoded <code>msgKey</code> attribute information should be populated in <code>msgKey.encodedAttrib</code>. Unencoded <code>msgKey</code> attribute information should be encoded after <code>encodeInit</code> returns, followed by <code>encodeKeyAttribComplete</code>. • If encoding <code>extendedHeader</code> information: pre-encoded <code>extendedHeader</code> information should be populated in the <code>extendedHeader</code> member of the message. Unencoded <code>extendedHeader</code> information should be encoded after the call to <code>encodeInit</code> and after <code>msgKey</code> attribute information is encoded. When <code>extendedHeader</code> encoding is completed, call <code>encodeExtendedHeaderComplete</code>.
encodeComplete	<p>Completes encoding of an <code>Msg</code>.</p> <p>All message content should be encoded prior to this call. This function expects the same <code>EncodeIterator</code> that was used with <code>encodeInit</code>.</p> <ul style="list-style-type: none"> • If the content (i.e., payload, <code>msgKey</code> attrib, and <code>extendedHeader</code>) encodes successfully, the <code>Boolean success</code> parameter should be set to <code>true</code> to finish encoding. • If any of the content fails to encode, the <code>Boolean success</code> parameter should be set to <code>false</code> to roll back the encoding of the message.
encodeKeyAttribComplete	<p>Completes encoding of any non-pre-encoded <code>msgKey</code> attribute information. Can be used only when message encoding leverages <code>encodeInit</code>. If the <code>MsgKeyFlags.HAS_ATTRIB</code> flag is set and <code>msgKey.encodedAttrib</code> is not populated, <code>msgKey</code> attribute information is expected after <code>encodeInit</code> returns, with the specific <code>attribContainerType</code> methods being used to encode it. This method expects the same <code>EncodeIterator</code> used with <code>encodeInit</code>.</p> <ul style="list-style-type: none"> • If encoding of the <code>msgKey</code> attribute information succeeds, the <code>Boolean success</code> parameter should be set to <code>true</code> to finish attribute encoding. • If encoding of attributes fails, the <code>Boolean success</code> parameter should be set to <code>false</code> to roll back encoding prior to <code>msgKey</code> attributes. <p>If both <code>msgKey</code> attributes and <code>extendedHeader</code> information are being encoded, <code>msgKey</code> attributes are expected first with <code>extendedHeader</code> being encoded after the call to <code>encodeKeyAttribComplete</code>.</p>

Table 144: `Msg` Encode Methods

ENCODE INTERFACE	DESCRIPTION
encodeExtendedHeaderComplete	<p>Completes encoding of any non-pre-encoded <code>extendedHeader</code> information. Can be used only when the message encoding leverages <code>encodeInit</code>. If the specific message's <code>HAS_EXTENDED_HEADER</code> flag is set and <code>extendedHeader</code> is not populated, this information is expected after <code>encodeInit</code> (and <code>encodeKeyAttribComplete</code> if encoding <code>msgKey</code> attributes) returns. This function expects the same <code>EncodeIterator</code> used with previous message encoding functions.</p> <ul style="list-style-type: none"> If encoding of <code>extendedHeader</code> succeeds, the <code>Boolean success</code> parameter should be set to <code>true</code> to finish encoding. If encoding of <code>extendedHeader</code> fails, the <code>Boolean success</code> parameter should be set to <code>false</code> to roll back to encoding prior to <code>extendedHeader</code>. <p>If both <code>msgKey</code> attributes and <code>extendedHeader</code> information are being encoded, <code>msgKey</code> attributes are expected first, while <code>extendedHeader</code> should be encoded after the call to <code>encodeKeyAttribComplete</code>.</p>

Table 144: `Msg` Encode Methods (Continued)

12.2.9.2 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 method, 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 */
RequestMsg reqMsg = (RequestMsg)CodecFactory.createMsg();
reqMsg.msgClass(MsgClasses.REQUEST); /* message is a request */
reqMsg.domainType(DomainTypes.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 name and length of 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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Msg.encodeInit. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}
else
{
    Buffer nonRWFBuffer = CodecFactory.createBuffer();
    /* retCode should be CodecReturnCodes.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)) < CodecReturnCodes.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 CodecReturnCodes.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 CodecReturnCodes.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)) < CodecReturnCodes.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 42: **Msg** Encoding Example #1, **encodeInit / encodeComplete** Use

12.2.9.3 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 */
RefreshMsg refreshMsg = (RefreshMsg)CodecFactory.createMsg();
refreshMsg.msgClass(MsgClasses.REFRESH); /* message is a refresh */
refreshMsg.domainType(DomainTypes.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 RefreshMsg */
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 encAttrib */
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)) < CodecReturnCodes.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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Msg.encode. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

```

Code Example 43: **Msg** Encoding Example #2, **encode** Use

12.2.9.4 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	<p>Decodes Msg header members.</p> <p>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.</p> <p>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.</p>
decodeKeyAttrib	<p>Prepares the DecodeIterator to decode Msg.msgKey.encodedAttrib information.</p> <p>This method expects the same DecodeIterator as was used with Msg.decode and the Msg.msgKey member that was populated by calling Msg.decode. 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.</p>

Table 145: **Msg** Decode Methods

12.2.9.5 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 following sample code, some error handling is omitted.

```

/* decode contents into the Msg structure */
if ((retCode = msg.decode(decIter)) >= CodecReturnCodes.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 */
    MsgKey key = msg.msgKey();
    /* If we have a key and it has attribute information, decode it */
    if (key != 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:
        System.out.println("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 */
    System.out.printf("Error (%d) (errno: %d) encountered with Msg.decode. Error Text: %s\n",
                      error.errorId(), error.sysError(), error.text());
}

```

Code Example 44: **Msg** Decoding Example

12.2.9.6 Encodelterator Utility Methods

The Transport API provides the following **Encodelterator** utility methods for use with the **Msg**.

METHOD	DESCRIPTION
replaceStreamId	Takes an encoded message and replaces the <code>streamId</code> without re-encoding the message. For more details on the <code>streamId</code> , refer to Section 12.1.3.
replaceSeqNum	Takes an encoded message and replaces the <code>seqNum</code> without re-encoding the message.
replaceGroupId	Takes an encoded message and replaces the <code>groupId</code> without re-encoding the message. For more information about group use, refer to Section 13.4.
replacePostId	Takes an encoded message and replaces the <code>postId</code> without re-encoding the message. For more information, refer to Section 13.9.
replaceStreamState	Takes an encoded message and replaces the <code>streamstate</code> without re-encoding the message. For more information about state values, refer to Section 11.2.6.
replaceDataState	Takes an encoded message and replaces the <code>datastate</code> without re-encoding the message. For more information about state values, refer to Section 11.2.6.
replaceStateCode	Takes an encoded message and replaces the <code>state.code</code> without re-encoding the message. For more information about state values, refer to Section 11.2.6.
setConfInfoInUpdatesFlag unsetConfInfoInUpdatesFlag	Sets or unsets the <code>RefreshMsgFlags.CONF_INFO_IN_UPDATES</code> flag on an encoded buffer.
setMsgKeyInUpdatesFlag unsetMsgKeyInUpdatesFlag	Sets or unsets the <code>RefreshMsgFlags.MSG_KEY_IN_UPDATES</code> flag on an encoded buffer.
setNoRefreshFlag unsetNoRefreshFlag	Sets or unsets the <code>RefreshMsgFlags.NO_REFRESH</code> flag on an encoded buffer.
setRefreshCompleteFlag unsetRefreshCompleteFlag	Sets or unsets the <code>RefreshMsgFlags.REFRESH_COMPLETE</code> flag on an encoded buffer.
setSolicitedFlag unsetSolicitedFlag	Sets or unsets the <code>RefreshMsgFlags.SOLICITED</code> flag on an encoded buffer.
setStreamingFlag unsetStreamingFlag	Sets or unsets the <code>RefreshMsgFlags.STREAMING</code> flag on an encoded buffer.

Table 146: Encodelterator Utility Methods

12.2.9.7 Decodelterator Utility Methods

The Transport API provides the following **Decodelterator** 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 <code>msgClass</code> information without fully decoding the message header.
extractDomainType	Takes an encoded message and returns the <code>domainType</code> information without fully decoding the message header.
extractStreamId	Takes an encoded message and returns the <code>streamId</code> information without fully decoding the message header. For more details on the <code>streamId</code> , refer to Section 12.1.3.
extractSeqNum	Takes an encoded message and returns the <code>seqNum</code> information without fully decoding the message header.
extractGroupId	Takes an encoded message and returns the <code>groupId</code> information without fully decoding the message header. For more information about group use, refer to Section 13.4.
extractPostId	Takes an encoded message and returns the <code>postId</code> information without fully decoding the message header. For more information, refer to Section 13.9.

Table 147: `DecoderIterator` Utility Methods

Chapter 13 Advanced Messaging Concepts

13.1 Multi-Part Message Handling

`RefreshMsg`, `PostMsg`, and `GenericMsg` 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 TREP 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 `RefreshMsg`, 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 `RefreshMsg`, 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 `RefreshMsg` 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 `RequestMsg` 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 ADH uses the combination of `priorityCount` and `priorityClass` to preempt items when the user's allowable cache list size is exceeded. ADH always preempts the item with the lowest `priorityCount` within the `priorityClass` and then provides an `StatusMsg` 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 **RequestMsg**:

- 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 **RequestMsg** 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 **RefreshMsg** 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 *Transport API Java Edition RDM Usage Guide*.

- An initial request containing only **RequestMsg.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 **RequestMsg.qos** and **RequestMsg.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 **RefreshMsg.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 **RequestMsg** 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 **RefreshMsg**, 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 **RefreshMsg**.

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 `RefreshMsg.groupId` (refer to Section 12.2.2) or `StatusMsg.groupId` (refer to Section 12.2.4) members. Once established, item group information can be modified via a subsequent `StatusMsg` or `RefreshMsg` 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 NIP applications, each publishing item streams belonging to specific services and item groups.

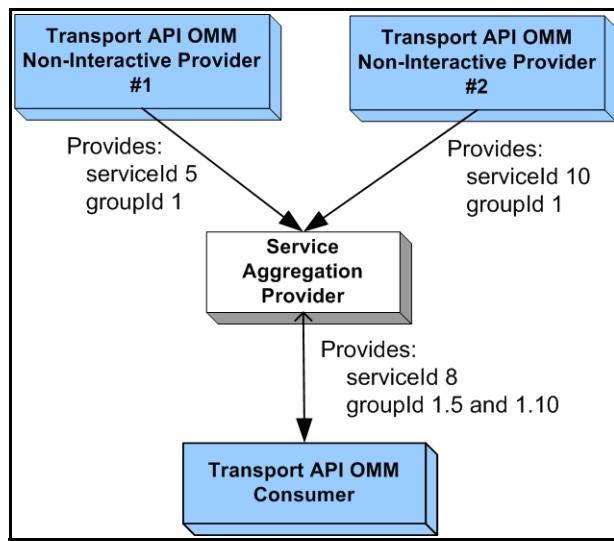


Figure 39. Item Group Example

Though the providers in this diagram use the same `groupId` for an item, using different `serviceIds` 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 NIP #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
CodecUtils.addGroupId	Appends a two-byte, unsigned integer to existing <code>groupId</code> content. Useful when modifying <code>groupId</code> buffers to ensure uniqueness.
groupId (from <code>RefreshMsg</code> and <code>StatusMsg</code>)	Takes a populated <code>Msg</code> structure, determines if <code>groupId</code> information is present and if available, returns it; NULL otherwise.
Decodelterator.extractGroupId	Takes an encoded message and returns the <code>groupId</code> without fully decoding the message header. Note: Multiple <code>DecodeIterator.extract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>Msg.decode</code> .
Encodelterator.replaceGroupId	Takes an encoded message and replaces the <code>groupId</code> without re-encoding the message.

Table 148: `groupId Buffer` Utility Methods

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 Java Edition RDM Usage Guide*.
- For a decision table providing example behavior for various state combinations, refer to Appendix A.

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.

An OMM interactive provider or NIP application is responsible for:

- Assigning and providing item group id values. This is accomplished by specifying the `RefreshMsg.groupId` or `StatusMsg.groupId` for all provided content¹.
- 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.

1. This does not include administrative domains such as Login, Source Directory, and Dictionary.

For more information about group status messages (including specific message content and formatting), refer to the *Transport API Java Edition RDM Usage Guide*.

- If items become available again, recovery should occur and items' states should be updated via a subsequent `RefreshMsg` or `StatusMsg` provided to any downstream components interested in the item.

An OMM 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 Java Edition RDM Usage Guide*.

- If group status changes are received, the state change should be propagated to all items associated with the indicated group, as noted by the `RefreshMsg.groupId` or `StatusMsg.groupId` provided with the item stream.
- Any recovery should follow `singleOpen` and `AllowSuspectData` rules, as described in the *Transport API Java Edition RDM 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 `RequestMsg` using the `singleOpen` and `AllowSuspectData msgKey` attributes. A providing application can acknowledge support for the behavior in the Login domain `RefreshMsg`, 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 `RefreshMsg`. For additional information, including on the `DomainTypes.LOGIN` domain definition, refer to the *Transport API Java Edition RDM 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 ALLOWSSUSPECTDATA = 1	CONVERSION WHEN: SINGLEOPEN = 1 ALLOWSSUSPECTDATA = 0	CONVERSION WHEN: SINGLEOPEN = 0 ALLOWSSUSPECTDATA = 1	CONVERSION WHEN: SINGLEOPEN = 0 ALLOWSSUSPECTDATA = 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 149: **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 **UpdateMsg** flow.
- Issuing a **resume** on a paused stream restarts the **UpdateMsg** 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 **RequestMsg** with the **RequestMsgFlags.PAUSE** flag set. This can occur on the initial **RequestMsg** or via a subsequent **RequestMsg** 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 **RefreshMsg** (this conveys initial state, permissioning, QoS, 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 **RequestMsg** 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 **RefreshMsg** 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 **RequestMsg** with the **RequestMsgFlags.STREAMING** flag set using the **streamId** associated with the applications login.

For more information about the **RequestMsg** and the **RSSL_RQMF_PAUSE****RequestMsgFlags.PAUSE** or **RSSL_RQMF_STREAMING****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 **RefreshMsg**. For more details on the Login **domainType** (**DomainTypes.LOGIN**), refer to the *Transport API Java Edition RDM Usage Guide*.

13.7 Batch Requesting

Applications can use the Transport API to send and / or receive batch requests.

- Consumers use a ***batch request*** to indicate interest in multiple like-item streams with a single `RequestMsg`.
- Providers should respond by providing a status on the batch request stream itself and with new individual streams for each item in the batch.

13.7.1 Batch Request Usage

Batch requesting can be leveraged across all non-administrative² 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 `StatusMsg` 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 a `RequestMsg` 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 Thomson Reuters-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`). Thomson Reuters reserves the empty name space (e.g., `:Element`). The `com.thomsonreuters.upa.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 a `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 a `StatusMsg` 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 `RequestMsg.msgKey` content should be associated with each item. If any request cannot be fulfilled, the provider should send a `StatusMsg` 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 `RequestMsg`. Because an OMM consumer requests the batch, positive `streamId` values should be assigned. For example, if the batch request uses `streamId 20` and requests ten items, the `StatusMsg` 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 `RequestMsg` 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 `RefreshMsg`.

- For an example of encoding a batch request, refer to Section 13.7.2.
- For more information about `RequestMsg` 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 *Transport API Java Edition RDM 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 RequestMsg Encoding Example

The following example demonstrates how to encode a batch `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:

- A `statusMsg` delivered on `streamId 10` which indicates that the batch is being processed and closes the stream.
- Three `RefreshMsgs` 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(DomainTypes.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 = CodecFactory.createArray();
    ArrayEntry nameEntry = CodecFactory.createArrayEntry();
    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 = CodecFactory.createBuffer();
        /* 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 45: Batch Request Encoding Example

13.8 Dynamic View Use

Applications can use the 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-administrative³ 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 **RequestMsg** 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 a **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 Thomson Reuters-defined portions, the 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**). Thomson Reuters reserves the empty name space (e.g., **:Element**). View-related enumerations and element name string constants are defined **com.thomsonreuters.upa.rdm.ElementNames**.

If a consumer application wishes to change a previously-specified view, the same process can be followed by issuing a subsequent **RequestMsg** 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 **RequestMsg** should continue to include

3. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. Other domains are considered non-administrative.

the `RequestMsgFlags.HAS_VIEW` flag, `:viewType` or `:ViewData` are not required. Sending a `RequestMsg` 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), a `RefreshMsg` 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 `RefreshMsg`.

- For details on `state.code` values, refer to Section 11.2.6.6.
- For details on the `RequestMsg` and `RequestMsgFlags.HAS_VIEW` flag values, refer to Section 12.2.1.
- For details on the `ElementList`, refer to Section 11.3.2.
- 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 RDM Usage Guide*.

13.8.1 RDM ViewTypes Names

The following table defines the `com.thomsonreuters.upa.rdm.ViewTypes`.

VIEW TYPE	DESCRIPTION
<code>FIELD_ID_LIST</code>	Indicates that <code>:ViewData</code> contains an array populated with <code>fieldId</code> values. The array should specify a <code>primitiveType</code> of <code>DataTypes.INT</code> and a fixed two-byte <code>itemLength</code> . For specific details about the <code>Array</code> , refer to Section 11.2.7.
<code>ELEMENT_NAME_LIST</code>	Indicates that <code>:ViewData</code> contains an array populated with element <code>name</code> values. The array should specify a <code>primitiveType</code> corresponding to the type used for the domain model's element names (e.g. <code>DataTypes.ASCII_STRING</code>). For specific details about the <code>Array</code> , refer to Section 11.2.7.

Table 150: RDM `Viewtypes` Values

13.8.2 Dynamic View RequestMsg Encoding Example

The following example demonstrates how to encode an `RequestMsg` 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(DomainTypes.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 = CodecFactory.createUInt();
    Array fidList = CodecFactory.createArray();
    ArrayEntry fidEntry = CodecFactory.createArrayEntry();
    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(":ViewType");
    elemEntry.dataType(DataTypes.ARRAY);
    retCode = elemEntry.encodeInit(encIter, 0);
    {
        Int fieldIdInt = CodecFactory.createInt();
        /* 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 46: View Request Encoding Example

13.9 Posting

The Transport API provides **posting** functionality: an easy way for OMM 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 Transport API can push content to all non-administrative⁴ domain model types, where specific usage and support should be indicated in the model definition. **PostMsg** payloads can include any container type; often this is a **Msg (DataTypes.MSG)**. When payload is a **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 a **PostMsg** 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 a **PostMsg** 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.

A **PostMsg** contains **Visible Publisher Identifier (VPI)** information (contained in **PostMsg.postUserInfo**), which identifies the user who posted it. **PostMsg.postUserInfo** must be populated and consists of:

- **postUserId**: which should be an ID associated with the user. For example, a DACS user ID or if unavailable, a process id)
- **postUserAddr**: which should contain the IP address⁵ of the application posting the content.

Optionally, such information can be carried along with republished **RefreshMsgs**, **UpdateMsgs**, or **StatusMsgs** so that receiving consumers can identify the posting user. For more information about VPI, refer to Section .

PostMsg.permData permissions the user who posts data. If the payload of the **PostMsg** is another nested message type (i.e., **RefreshMsg**) 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 **PostMsg**, the nested message does not need to include permission data. The permission data is used in conjunction with the **PostMsg.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 **AckMsg.ackId** should be populated using the **PostMsg.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.

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 9.14.

You can split content across multiple `PostMsg` messages. When sending a multi-part `PostMsg`, 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 `AckMsg.ackId` is populated using the `PostMsg.postId`, and each `AckMsg.seqNum` is populated using the `PostMsg.seqNum`. Each part of the `PostMsg` should specify a `partNum`, where the first part begins with `0`. The final part of a multi-part `PostMsg` should have the `PostMsgFlags.POST_COMPLETE` flag set to indicate that it is the final part.

A provider application can indicate support for posting and acknowledgment use by sending the `msgKey` attribute `supportOmmPost` in the Login domain `RefreshMsg`.

- For more information on the `PostMsg`, refer to Section 12.2.7.
- For more information on the `AckMsg`, refer to Section 12.2.8.
- For more information on managing multi-part `PostMsg`s, refer to Section 13.1.
- For more details on the Login `domainType` (`DomainTypes.LOGIN`), see the *Transport API RDM 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 */
PostMsg postMsg = (PostMsg)CodecFactory.createMsg();
postMsg.msgClass(MsgClasses.POST);
postMsg.streamId(1); /* Use streamId of the Login stream for off-stream posting */
postMsg.domainType(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.applyHasMsgKey();
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(Thread.currentThread().getId());
postMsg.postUserInfo().userAddr(InetAddress.getLocalHost().getHostAddress());

/* 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 */
    UpdateMsg updMsg = (UpdateMsg)CodecFactory.createMsg();
    updMsg.msgClass(MsgClasses.UPDATE);
    updMsg.streamId(1);
    updMsg.domainType(DomainTypes.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 47: 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 */
AckMsg ackMsg = (AckMsg)CodecFactory.createMsg();
ackMsg.msgClass(MsgClasses.ACK);
ackMsg.domainType(DomainTypes.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 48: Post Acknowledgement Encoding Example

13.10 Private Streams

The 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 a TREP 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 **GenericMsgs**).

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, an OMM consumer observes the following general process:

- The OMM consumer application issues a request for the item data it wants on a private stream. This **RequestMsg** 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:
 - **RefreshMsg** with the **RefreshMsgFlags.PRIVATE_STREAM** flag; typically sent when there is immediate content to provide in the response.
 - **StatusMsg** 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.
 - **AckMsg** with the **AckMsgFlags.PRIVATE_STREAM** flag; can be used as an alternative to the **StatusMsg**.
- 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 **StatusMsg** 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 `RequestMsg` 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 a:
 - `StatusMsg` including the `StatusMsgFlags.PRIVATE_STREAM` flag; typically sent when there is not any content to provide as part of the redirect.
 - `RefreshMsg` including the `RefreshMsgFlags.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 `RequestMsg` and use the `RequestMsgFlags.PRIVATE_STREAM` flag. This process follows standard stream redirect logic and the private stream establishment protocol described above.

Visible Publisher Identifier (VPI)

The Transport API offers the **Visible Publisher Identifier (VPI)** feature, which inserts originating publisher information into both RSSL and SSL message payloads. You can use VPI 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 ADH.

VPI is present on Post, Refresh, Update, and Status Messages and is carried in `PostMsg.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.

A `PostMsg` contains data (in `PostMsg.postUserInfo`) that identifies the user who posts content. For this reason, `PostMsg.postUserInfo` must be populated with a:

- `postUserId`: An ID associated with the posting user. The application should determine what information to put into this field (e.g., a DACS user ID).
- `postUserAddr` The address of the posting user's application that posted the contents. The application should decide what information to put into this field (e.g., an IP address).

Optionally, this data can be republished by the provider in a `RefreshMsgs`, `UpdateMsgs`, or `StatusMsgs` so that receiving consumers can identify the posting user.

The Transport API allows the VPI to be populated on Post messages submitted by an OMM Consumer application before the post is sent over the network.

Provider applications receive VPI in Post Messages. Additionally, OMM providers can optionally set VPI in their response messages. If the upstream provider is an intermediary device getting data from an upstream source, then the intermediary device will route the VPI as set in the `PostMsg` to the upstream source. The final publisher in the upward chain decides whether to set the VPI in its published responses.

VPI information can also be communicated using FIDs defined in the publisher component. For further details refer to the publishing component's documentation.

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 11.2.6.
- For general information about Item Groups, refer to Section 13.4.
- For information about group status delivery and formatting, refer to the *Transport API RDM 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.

Table 151: Item and Group State Decision Table

Status Type	Stream State	Data State	Description	Application Action
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.
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 151: Item and Group State Decision Table (Continued)

© 2015, 2016 Thomson Reuters. All rights reserved.

Republication or redistribution of Thomson Reuters content, including by framing or similar means, is prohibited without the prior written consent of Thomson Reuters. 'Thomson Reuters' and the Thomson Reuters logo are registered trademarks and trademarks of Thomson Reuters and its affiliated companies.

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

Document ID: ET AJ300UM.160
Date of issue: 05 February 2016

