

# REFINITIV ROBUST FOUNDATION API V8.2.2.L1

DEVELOPERS GUIDE  
JAVA EDITION

© Refinitiv 2007 - 2017, 2019, 2020 - 2021. All Rights Reserved.

Refinitiv, 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. Refinitiv, 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 Refinitiv and may not be reproduced, disclosed, or used in whole or part without the express written permission of Refinitiv.

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 Refinitiv as set out in the contract existing between us.

# Contents

<b>Chapter 1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Purpose .....	1
1.2	Product Description .....	1
1.3	What's New in this Release .....	1
1.4	Scope.....	2
1.5	Audience.....	2
1.6	Use This Document .....	2
1.7	Acronyms.....	2
1.8	Documentation Feedback.....	3
1.9	References .....	3
1.10	Conventions.....	4
1.10.1	<i>Typographic</i> .....	4
1.10.2	<i>Packages</i> .....	4
1.10.3	<i>Diagrams</i> .....	4
1.10.4	<i>Programming Examples</i> .....	5
<b>Chapter 2</b>	<b>RFA Concepts .....</b>	<b>6</b>
2.1	Refinitiv APIs .....	6
2.1.1	<i>The Refinitiv Robust Foundation API</i> .....	6
2.1.2	<i>The Transport API</i> .....	7
2.1.3	<i>Contrasting RFA and the Transport API</i> .....	7
2.1.4	<i>Performance</i> .....	8
2.1.5	<i>Functionality</i> .....	8
2.2	Features of RFA .....	10
<b>Chapter 3</b>	<b>Concepts: Consumer / Provider .....</b>	<b>15</b>
3.1	Consumer / Provider Overview.....	15
3.2	Consumers .....	15
3.2.1	<i>Subscription: Request/Response</i> .....	16
3.2.2	<i>Watchlist</i> .....	17
3.2.3	<i>Batch</i> .....	17
3.2.4	<i>Views</i> .....	17
3.2.5	<i>Symbol List</i> .....	18
3.2.6	<i>Enhanced Symbol List</i> .....	19
3.2.7	<i>Pause/Resume</i> .....	19

3.2.8	<i>Posting</i> .....	20
3.2.9	<i>Connection Recovery</i> .....	22
3.2.10	<i>Item Recovery and SingleOpen</i> .....	22
3.2.11	<i>Warm Standby</i> .....	22
3.2.12	<i>Private Streams</i> .....	22
3.2.13	<i>Load Balancing</i> .....	23
3.3	<i>Providers</i> .....	23
3.3.1	<i>Interactive Providers</i> .....	24
3.3.2	<i>Non Interactive Providers</i> .....	25
<b>Chapter 4</b>	<b>System View .....</b>	<b>27</b>
4.1	Overview of Network System Architecture .....	27
4.2	RFA High Level Architecture Overview .....	28
4.3	Refinitiv Real-Time Advanced Distribution Server .....	28
4.4	Refinitiv Real-Time Advanced Distribution Hub .....	29
4.5	Refinitiv Real-Time .....	30
4.6	Refinitiv Data Feed Direct.....	31
4.7	Internet Connectivity via HTTP and HTTPS.....	31
4.8	Direct-Connect.....	32
<b>Chapter 5</b>	<b>Model Overview .....</b>	<b>33</b>
5.1	Open Message Model .....	33
5.2	Domain Message Model.....	33
5.3	Refinitiv Domain Model.....	33
5.4	User-Defined Domain Model .....	33
5.5	Refinitiv Wire Format .....	33
5.6	Refinitiv Domain Models Overview.....	33
5.7	OMM Data Constructs .....	35
5.7.1	<i>Data Types</i> .....	35
5.7.2	<i>Primitive Types</i> .....	35
5.7.3	<i>Container Types</i> .....	36
5.7.4	<i>Summary Data</i> .....	37
5.7.5	<i>Defined Data</i> .....	38
5.7.6	<i>Iterators</i> .....	38
5.8	OMM Messages .....	38
5.8.1	<i>Request Message</i> .....	38
5.8.2	<i>Response Message</i> .....	38

5.8.3	<i>Generic Message</i> .....	38
5.9	Services, Concrete Services, and Service Groups .....	39
5.10	Packages in RFA Java .....	39
5.10.1	<i>OMM Package</i> .....	39
5.10.2	<i>Common Package</i> .....	39
5.10.3	<i>Session Package</i> .....	40
5.10.4	<i>Configuration Package</i> .....	40
5.10.5	<i>Logging</i> .....	40
5.10.6	<i>Other RFA Packages</i> .....	41
<b>Chapter 6</b>	<b>OMM Package</b> .....	<b>42</b>
6.1	OMM Package Overview .....	42
6.2	OMM Message Interface: OMMMsg .....	42
6.2.1	<i>OMM Message Elements</i> .....	42
6.2.2	<i>OMMMsg Utility Functions</i> .....	45
6.2.3	<i>Message Types</i> .....	46
6.2.4	<i>Attribute Information</i> .....	47
6.2.5	<i>Indication Flags</i> .....	49
6.2.6	<i>Hint Flags</i> .....	51
6.2.7	<i>User Rights</i> .....	52
6.2.8	<i>Item Group</i> .....	52
6.3	OMM Message Type: Request .....	52
6.3.1	<i>Streaming Request</i> .....	53
6.3.2	<i>Changeable Event Stream Attributes</i> .....	53
6.3.3	<i>Nonstreaming Request</i> .....	54
6.4	OMM Message Type: Close Request .....	54
6.5	OMM Message Type: Refresh Response .....	54
6.6	OMM Message Type: Update Response .....	54
6.7	OMM Message Type: Status Response .....	55
6.8	OMM Message Type: Generic .....	55
6.9	OMM Message Type: Post .....	56
6.9.1	<i>Publisher Data</i> .....	56
6.9.2	<i>On-stream and Off-stream Posting</i> .....	56
6.10	OMM Message Type: Acknowledge Response .....	56
6.11	OMM Encoder .....	57
6.11.1	<i>Encoding Sequence</i> .....	57

6.11.2	<i>Encoder Utility Methods</i> .....	57
6.12	Versioning Support .....	57
6.13	Pooling.....	59
6.13.1	<i>Managing the Pool</i> .....	59
6.13.2	<i>Managed Objects</i> .....	59
6.14	Creating the Message .....	61
6.14.1	<i>Allocate Memory</i> .....	61
6.14.2	<i>Set Header</i> .....	61
6.14.3	<i>Encoding Payloads</i> .....	63
6.14.4	<i>Message Encoding Examples</i> .....	63
6.15	Decoding Messages .....	65
6.15.1	<i>Decoding the Message Header</i> .....	65
6.15.2	<i>Message Header Decoding Example</i> .....	66
6.15.3	<i>Decoding the Attribute Information</i> .....	66
6.15.4	<i>Decoding Publisher Principal Identity Information</i> .....	67
6.15.5	<i>Decoding the Payload</i> .....	68
6.15.6	<i>Decoding Custom OMMData</i> .....	68
<b>Chapter 7</b>	<b>Common Package</b> .....	<b>69</b>
7.1	Common Package Overview .....	69
7.2	Context .....	69
7.2.1	<i>Context Scenarios</i> .....	69
7.2.2	<i>Context Usage Guidelines</i> .....	71
7.3	Event Distribution .....	71
7.3.1	<i>Overview</i> .....	71
7.3.2	<i>Event Distribution Interfaces</i> .....	75
7.4	Notification Client.....	79
7.4.1	<i>Dispatchable Interface Methods</i> .....	80
7.4.2	<i>Notification Client Example</i> .....	80
7.5	Status.....	81
7.6	RFA Exceptions .....	81
7.7	Refinitiv Wire Format Versioning .....	81
7.8	Quality of Service .....	82
7.8.1	<i>Overview</i> .....	82
7.8.2	<i>QualityOfService</i> .....	82
7.8.3	<i>QualityOfServiceRequest</i> .....	83

7.9	Common Package Usage Guidelines .....	84
7.9.1	Event Distribution Usage Guidelines .....	84
7.9.2	Using Event Distribution Model in a Single Thread Context .....	85
7.9.3	Using Event Distribution Model in Multiple Thread Contexts .....	85
7.9.4	Thread Safety .....	86
<b>Chapter 8</b>	<b>Session Package .....</b>	<b>87</b>
8.1	Overview .....	87
8.1.1	Session and Connection .....	87
8.1.2	Configuring a Session .....	88
8.2	Session Interface .....	88
8.2.1	Session Interface Methods .....	88
8.2.2	Session Interface Examples .....	89
8.3	OMM Event Sources .....	90
8.3.1	OMMConsumer .....	90
8.3.2	OMMProvider .....	92
8.4	OMM Command .....	94
8.4.1	OMMHandleItemCmd .....	94
8.4.2	OMMInactiveClientSessionCmd .....	95
8.4.3	OMMItemCmd .....	95
8.5	Interest Specifications .....	96
8.6	Event Interfaces and Statuses .....	97
8.6.1	Connection Event .....	98
8.6.2	OMM Command Error Event .....	99
8.6.3	OMM Active Client Session Event .....	99
8.6.4	OMM Inactive Client Session Event .....	99
8.6.5	OMM Item Event .....	99
8.6.6	OMM Listener Event .....	99
8.6.7	OMM Solicited Item Event .....	99
8.6.8	OMM Connection Event .....	100
8.6.9	OMM Connection Statistic Event .....	100
<b>Chapter 9</b>	<b>Config Package .....</b>	<b>101</b>
9.1	Config Package Overview .....	101
9.2	Configuration Database .....	101
9.2.1	Configuration Concepts .....	101
9.2.2	Naming Scheme .....	102

9.2.3	<i>Configuration Sharing</i> .....	103
9.3	Config Package .....	105
9.4	Configuration Using Java Preferences API .....	106
9.4.1	<i>Populate Config Database</i> .....	106
9.4.2	<i>Remove RFA Configuration Nodes</i> .....	107
9.4.3	<i>Query RFA Configuration Information: Retrieving and Iterating RFA Configuration Nodes</i> .....	107
9.5	Configuration Tools .....	108
<b>Chapter 10</b>	<b>OMM Data</b> .....	<b>109</b>
10.1	Overview .....	109
10.2	OMM Data Interfaces .....	109
10.2.1	<i>Interfaces Hierarchy</i> .....	110
10.2.2	<i>Data Dictionary Utilities</i> .....	111
10.3	Primitive Data .....	111
10.3.1	<i>Standard and Defined Data Types</i> .....	112
10.3.2	<i>OMMNumeric</i> .....	112
10.3.3	<i>OMMDataBuffer</i> .....	120
10.3.4	<i>OMMDateTime</i> .....	123
10.3.5	<i>OMMQos</i> .....	127
10.3.6	<i>OMMState</i> .....	129
10.3.7	<i>OMMEnum</i> .....	133
10.3.8	<i>OMMArray</i> .....	134
10.4	Dataformats .....	135
10.4.1	<i>Encoding Nested Dataformats</i> .....	137
10.4.2	<i>Decoding Nested Dataformats</i> .....	138
10.4.3	<i>ElementList</i> .....	139
10.4.4	<i>FieldList</i> .....	142
10.4.5	<i>Vector</i> .....	145
10.4.6	<i>Map</i> .....	151
10.4.7	<i>Series</i> .....	156
10.4.8	<i>FilterList</i> .....	159
10.5	Permission Data .....	164
10.5.1	<i>Encoding Permission Data</i> .....	164
10.5.2	<i>Decoding Permission Data</i> .....	164
10.6	Summary Data.....	165
10.6.1	<i>Encoding Summary Data</i> .....	165

10.6.2	<i>Decoding Summary Data</i> .....	165
10.7	Actions .....	166
10.7.1	<i>Encoding Actions</i> .....	166
10.7.2	<i>Decoding Actions</i> .....	166
10.8	Data Definition .....	166
10.8.1	<i>Data Definition Dictionary</i> .....	169
10.8.2	<i>Encoding Data Definitions</i> .....	169
10.8.3	<i>Decoding Data Definitions</i> .....	171
10.9	Complex Data .....	171
10.9.1	<i>OMMItemGroup</i> .....	171
10.9.2	<i>OMMPriority</i> .....	172
10.9.3	<i>OMMQosReq</i> .....	172
<b>Chapter 11</b>	<b>Logging</b> .....	<b>173</b>
11.1	Logging Overview .....	173
11.2	Logger Concepts .....	173
11.3	Logging Levels .....	173
11.4	Logger Names and Hierarchy .....	173
11.5	Logger Monitor .....	174
11.6	Resource Files .....	174
11.6.1	<i>Resource File Naming Convention</i> .....	174
11.6.2	<i>Resource File Location</i> .....	174
11.7	Namespaces of Loggers Used by the RFA Session Layer .....	174
11.8	Resource Files (For Localization) .....	175
<b>Chapter 12</b>	<b>Basic Applications</b> .....	<b>176</b>
12.1	Application Design .....	176
12.1.1	<i>Initialization</i> .....	176
12.1.2	<i>An Overview of Application Type-Specific Functionality</i> .....	176
12.1.3	<i>Cleanup</i> .....	177
12.1.4	<i>Design Considerations</i> .....	177
12.2	OMM Consumer Applications .....	178
12.2.1	<i>Consumer Application Tasks</i> .....	179
12.2.2	<i>Startup</i> .....	181
12.2.3	<i>Sending Requests</i> .....	181
12.2.4	<i>Modifying Event Stream</i> .....	185
12.2.5	<i>Sending Generic and Post Messages</i> .....	186

12.2.6	<i>Handling Inbound Events</i> .....	186
12.2.7	<i>Registering Interest In Submission Error Events</i> .....	193
12.2.8	<i>Handling Submission Error Events</i> .....	193
12.2.9	<i>Registering Interest In OMM Connection Events</i> .....	193
12.2.10	<i>Handling OMM Connection Events</i> .....	193
12.2.11	<i>Registering Interest in OMM Connection Stats Events</i> .....	194
12.2.12	<i>Handling OMM Connection Stats Events</i> .....	194
12.2.13	<i>Shutting Down an Application</i> .....	195
12.3	<i>OMM Interactive Provider Application</i> .....	197
12.3.1	<i>Interactive Provider Achitecture</i> .....	197
12.3.2	<i>Interactive Provider Task Overview</i> .....	198
12.3.3	<i>Client Session Handles and Request Tokens</i> .....	200
12.3.4	<i>Startup</i> .....	200
12.3.5	<i>Listening Port Setup</i> .....	201
12.3.6	<i>Accepting/Rejecting Client Sessions</i> .....	201
12.3.7	<i>Processing Messages from Clients</i> .....	202
12.3.8	<i>Sending Messages</i> .....	205
12.3.9	<i>Error Notifications</i> .....	207
12.3.10	<i>Client Session Disconnections</i> .....	207
12.3.11	<i>Interactive Provider Application Shut Down</i> .....	207
12.4	<i>OMM Non-Interactive Provider</i> .....	210
12.4.1	<i>Non-interactive Provider Architecture</i> .....	210
12.4.2	<i>Non-interactive Provider Task Overview</i> .....	211
12.4.3	<i>Startup</i> .....	212
12.4.4	<i>Login</i> .....	212
12.4.5	<i>Process Event</i> .....	213
12.4.6	<i>Sending Messages</i> .....	213
12.4.7	<i>Closing a Non-Interactive Provider Application</i> .....	214
12.5	<i>Hybrid</i> .....	215
12.5.1	<i>Hybrid Architecture</i> .....	215
12.5.2	<i>Hybrid Task Overview</i> .....	216
12.5.3	<i>Startup</i> .....	218
12.5.4	<i>Listening Port Setup</i> .....	218
12.5.5	<i>Accepting/Rejecting Client Sessions</i> .....	218
12.5.6	<i>Receiving Messages From Consumers</i> .....	219

12.5.7	<i>Receiving Events From Providers</i> .....	220
12.5.8	<i>Closing Hybrid Applications</i> .....	220
<b>Chapter 13</b>	<b>RFA Feature Details</b> .....	<b>221</b>
13.1	Feature Detail Overview .....	221
13.2	Batch .....	222
13.2.1	<i>Overview</i> .....	222
13.2.2	<i>Configuration</i> .....	222
13.2.3	<i>OMM Interface Support</i> .....	222
13.2.4	<i>Session Package Support</i> .....	222
13.2.5	<i>Details</i> .....	222
13.2.6	<i>Application Examples</i> .....	226
13.3	Views .....	226
13.3.1	<i>OMM and RDM Interface Support</i> .....	226
13.3.2	<i>Details</i> .....	227
13.4	Pause and Resume .....	228
13.4.1	<i>OMM Interface Support</i> .....	228
13.4.2	<i>Details</i> .....	228
13.4.3	<i>Application Examples</i> .....	229
13.5	Generic Messages .....	232
13.5.1	<i>Overview</i> .....	232
13.5.2	<i>OMM Interfaces Support</i> .....	232
13.5.3	<i>Details</i> .....	233
13.5.4	<i>Application Examples</i> .....	234
13.6	Posting .....	234
13.6.1	<i>Overview</i> .....	234
13.6.2	<i>Configuration</i> .....	234
13.6.3	<i>OMM Interfaces Support</i> .....	234
13.6.4	<i>Application Examples</i> .....	242
13.7	Private Streams .....	243
13.7.1	<i>Overview</i> .....	243
13.7.2	<i>OMM Interfaces Support</i> .....	243
13.7.3	<i>Details</i> .....	243
13.7.4	<i>Application Example</i> .....	244
13.8	Visible Publisher Identifier .....	244
13.8.1	<i>OMM Interfaces Support</i> .....	244

13.8.2	<i>Details</i> .....	244
13.8.3	<i>Application Example</i> .....	244
13.9	Enhanced Symbol List.....	244
13.9.1	<i>OMM and RDM Interface Support</i> .....	245
13.9.2	<i>Details</i> .....	245
13.10	Service Groups .....	247
13.10.1	<i>Configuration</i> .....	248
13.10.2	<i>Details</i> .....	249
13.11	Recovery.....	251
13.11.1	<i>Configuration</i> .....	251
13.11.2	<i>OMM Interfaces Support</i> .....	251
13.11.3	<i>Details</i> .....	252
13.12	Quality of Service .....	252
13.12.1	<i>Overview</i> .....	252
13.12.2	<i>OMM Interface</i> .....	253
13.12.3	<i>Details</i> .....	253
13.13	Item Group Management.....	254
13.13.1	<i>Overview</i> .....	254
13.13.2	<i>OMM Interfaces Support</i> .....	254
13.13.3	<i>Details</i> .....	254
13.14	Warm Standby .....	255
13.14.1	<i>Configuration</i> .....	256
13.14.2	<i>OMM and RDM Interface Support</i> .....	256
13.14.3	<i>Details</i> .....	256
13.15	Connection Redirect .....	257
13.15.1	<i>Configuration</i> .....	258
13.15.2	<i>Details</i> .....	258
13.16	Tunneling .....	258
13.16.1	<i>Configuration</i> .....	259
13.16.2	<i>Details</i> .....	260
13.16.3	<i>Proxy Authentication</i> .....	260
13.17	Non-Interactive Provider to Reliable Multicast .....	274
13.17.1	<i>Configuration</i> .....	275
13.18	Provider Dictionary Download from an ADH .....	275
13.18.1	<i>OMM Interfaces Support</i> .....	275

13.18.2	<i>Details</i> .....	275
13.18.3	<i>Application Example</i> .....	276
13.19	Application Signing .....	276
<b>Chapter 14</b>	<b>Application Performance Tuning and Considerations</b> .....	<b>278</b>
14.1	Performance Tuning Overview .....	278
14.2	Threads .....	278
14.3	RFA Consumer Queues .....	278
14.4	Threading Models .....	279
14.4.1	<i>Callback Model</i> .....	279
14.4.2	<i>Client Model</i> .....	279
14.4.3	<i>Notification Client Model</i> .....	279
14.5	Configuring the RFA Consumer for Performance .....	280
14.5.1	<i>Low Latency with Callback Model</i> .....	280
14.5.2	<i>Low Latency with Client Model</i> .....	280
14.5.3	<i>Full Throughput with Client Model</i> .....	281
14.5.4	<i>Throughput with Callback Model</i> .....	282
14.5.5	<i>Multiple Event Queues (Client Model)</i> .....	282
14.5.6	<i>Horizontal Scaling</i> .....	283
14.5.7	<i>Connection Sharing</i> .....	284
14.5.8	<i>Throttling</i> .....	284
14.6	Configuring the Provider for Performance .....	285
14.7	Other Tuning Configuration Parameters .....	285
14.7.1	<i>tcp_nodelay</i> .....	285
14.7.2	<i>initialWatchlistSize</i> .....	285
14.7.3	<i>enableOMMEventAge</i> .....	285
<b>Appendix A</b>	<b>Quick Reference</b> .....	<b>286</b>
<b>Appendix B</b>	<b>Deprecated Functionality</b> .....	<b>292</b>
B.1	Deprecated Classes .....	292
B.2	Deprecated Fields .....	292
B.3	Deprecated Methods .....	293

# Chapter 1 Introduction

## 1.1 Purpose

This manual describes application development using the *RFA Java Edition* and provides basic concepts, typical activities (including configuration and logging), and examples of how to access market information.

## 1.2 Product Description

RFA is an API suite designed for performance and flexibility to access, use, create, and provide data. It provides a robust framework for component or object-oriented applications. RFA can connect to many different components (eg., Refinitiv Real-Time, Enterprise Platform, Refinitiv Real-Time Advanced Transformation Server, Refinitiv Data Feed Direct, etc.) and is an event-driven API at the session layer of the OSI model.

Use RFA for applications that require high throughput, low latency, or both. RFA can support thousands of requests per second and hundred of thousands of open items at a time:

- In high-throughput mode, applications can support hundreds of thousands of updates per second, or more, with very low latency.
- In low latency mode, applications can support hundreds of thousands of updates per second with sub-millisecond latency.
- To support high throughput and low latency, RFA has a flexible thread model that is both thread safe and thread aware.

RFA can connect to many different types of servers and data feeds. RFA can normalize market data events through the use of two different session layer interfaces. RFA also provides several encoders and decoders for creating and parsing different types of data. By separating data presentation and session semantics, RFA supports flexible application design and provides performance trade-offs when manipulating data.

RFA is composed of several conceptual packages, each corresponding to one or more Java packages. Each package is also contained in one or more libraries. RFA includes the OMM Message and Data, ANSI Page, and Refinitiv Data Access Control System Lock packages.

- **Common Package:** provides base classes for other packages. It also includes a queue-based Event Distribution mechanism for multi-threaded applications to safely dispatch events.
- The **Configuration and Logger Packages:** provide default and customizable utilities for configuration and logging by applications and RFA.
- The **Session Layer Package:** provides subscription, publication, and network connection encapsulation. It supports several connection types and several different wire formats. A Session corresponds to a configuration context, one or more connections, and a thread context. The Session Layer Package has OMM interface supports many different types of existing market data infrastructure components and data formats using a single interface.
- **OMM Message and Data, ANSI Page, and DACS Packages:** encode and decode various data formats that can be sent and received through the Session Layer Package. OMM interfaces support the OMM provided by Refinitiv Data Feed Direct and Refinitiv Real-Time Distribution System.

Additionally, the RFA product includes:

- The Value Added interface, a programmatic interface that simplifies some development tasks by eliminating boilerplate code. The use of this interface is most appropriate when your application does not need the broader range of options afforded by the RFA API. For details, refer to the *RFA Value Added API Developers Guide*.
- Example programs, developer guides, and HTML-based references manuals.

## 1.3 What's New in this Release

The 8.0 release of RFA Java Edition includes the following new features and enhancements:

- **Message in message:** an application can now encode and decode any message type inside of any message. In this release any message can be set as a payload for other messages.
- **Connection statistics:** an application can register to receive events for the number of bytes written and read on the connection.
- **Expanded OMM DateTime:** OMM DateTime supports nanosecond and microsecond precision.
- **Expanded OMMNumeric:** OMMNumeric supports +/- Infinity and Not A Number (NaN).
- The MarketData interface is removed from the Session Layer Package. Only the OMM interface is available.

All deprecated functionality is listed in Appendix B.

## 1.4 Scope

The document provides functional descriptions of the RFA and RFA API applications, including their design, architecture, and usage. Usage focuses on the following packages: *Session Layer*, *Open Message Model*, *Configuration*, *Logger* and *Common*.

This document occasionally provides UML Activity Diagrams and code snippets depicting typical activities for developing applications using RFA.

## 1.5 Audience

The developers guide is intended for Java software programmers developing RFA-based applications for the financial marketplace.

To best understand the manual's contents, developers should have a basic understanding of trading room infrastructure and concepts of object-oriented analysis and design.

## 1.6 Use This Document

- To gain a broad understanding of RFA.
- To learn how RFA Java programs are architected.
- To become familiar with RFA's programatic interfaces.
- As a reference to be used in conjunction with the Example programs.

## 1.7 Acronyms

ACK	Acknowledge
ADH	Refinitiv Real-Time Advanced Distribution Hub
ADS	Refinitiv Real-Time Advanced Distribution Server
ANSI	American National Standard Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
DACS	Refinitiv Data Access Control System
EED	Refinitiv Real-Time Edge Device
FID	Field IDentifier
GC	Garbage Collection
GUI	Graphical User Interface
RRT	Refinitiv Real-Time
IP	Internet Protocol
JAR	Java Archive
JDK	Java Development Kit
JIT	Just in Time
JNI	Java Native Interface
JRE	Java Runtime Environment
LAN	Local Area Network
MD	Market Data
NAK	Negative Acknowledge
NYSE	New York Stock Exchange
OMM	Open Message Model
QoS	Quality of Service

RBCS	Refinitiv Basic Character Set
RDF	Refinitiv Data Feeds
RDF Direct	Refinitiv Data Feed Direct
RDM	Refinitiv Domain Models
RFA	Refinitiv Robust Foundation API
RIC	Reuters Instrument Code
RSSL	Refinitiv Source Sink Library
RWF	Refinitiv Wire Format
SFC	System Foundation Classes
RTDS	Refinitiv Real-Time Distribution System
UML	Unified Modeling Language
URL	Uniform Resource Locator
UTF-8	8-bit Unicode Transformation Format
VM	Virtual Machine
XML	Extensible Markup Language

## 1.8 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@refinitiv.com](mailto:apidocumentation@refinitiv.com).
- Mark up the PDF using the Comment feature in Adobe Reader. After adding your comments, you can submit the entire PDF to Refinitiv by clicking **Send File** in the **File** menu. Use the [apidocumentation@refinitiv.com](mailto:apidocumentation@refinitiv.com) address.

## 1.9 References

You can obtain all Refinitiv documentation on the Developer Community Portal at the URL: <https://developers.refinitiv.com/home>. You will need an account to access the site (Refinitiv will set up an account for you for free).

If RFA API interacts with other components, their configuration can affect how data is handled. Be aware of how these components are configured, so that you can write your applications more effectively. To realize best-of-class RFA API application design, you might need to refer to the appropriate component documentation. To obtain third-party documentation, visit the vendor's website and contact them directly.

1. *RFA, RDM Usage Guide*  
Describes the Refinitiv Data Models.
2. *RFA Java Reference Manual*  
Set of HTML pages that describe all public interfaces in RFA API Java.
3. *RFA - Java Edition - Configuration and Logging Guide*  
Describes the configuration and logging information.
4. *Java 2 Standard Edition Version 1.8 API Specification*  
<http://docs.oracle.com/javase/8/docs/api/>.
5. *AnsiPage API JAVA Edition 5.0 Release- Developers Guide*  
Describes the concepts and usage of page-based encoding and decoding.
6. *DACS LOCK API JAVA Edition Developers Guide*  
Describes the DACSLOCK API concepts and usage.
7. *DACS LOCK API Reference Manual*  
Set of HTML pages that describe interfaces for managing authorization lock information.

8. *RFAJ Quick Start Guide*  
Provides introduction to the RFA API Java
9. *RFA Value Added API Developers Guide*  
Describes the Refinitiv Value Added user interface.
10. *The Rational Unified Process – an Introduction, Philippe Kruchten, Addison-Wesley, 1999.*

## 1.10 Conventions

### 1.10.1 Typographic

- Java keywords, inline code snippets, and methods are shown in the `Blue Lucida Console` font. Method names omit their arguments unless the arguments are needed for clarity.
- Parameters, filenames, and directories are formatted in **bold**.
- Sample code is shown within an orange-shaded block.
- Variables are *italicized*, as well as key terms (when introduced for the first time).

### 1.10.2 Packages

There are two categories of packages discussed in this document:

- **RFA packages:** which are groups of related programmatic interfaces. For example, the **Common** package contains classes that are used by the other packages. It contains classes such as Events, Timers, and other helpful classes. The **OMM** package contains classes that define the data and its format.
- **Java packages:** which are mechanisms for organizing [Java classes](#) into [namespaces](#). This document will try to use the full name when talking about packages to distinguish between the two.

### 1.10.3 Diagrams

This document depicts many diagrams using Unified Modeling Language (UML). Diagrams are primarily class and interaction diagrams. All diagrams target a high-level overview of RFA architecture as opposed to detailed designs.

**Class diagrams** have the following conventions:

- Only public methods are shown.
- In most cases, default constructors, copy constructors, overloaded assignment operators and destructors are not shown.
- Exception specifications on interface methods are not shown.

**Interaction diagrams** have the following conventions:

- Scenario diagrams in several sections describe the main success scenario (typically the most common and expected scenario).
- Instance names are commonly not specified but rather assumed to have the same name as the class name.
- In cases when the scenario contains multiple instances of the same class, instance names are specified.
- A broken rectangle on the same vertical line implies a possible thread context switch.

## 1.10.4 Programming Examples

Throughout this manual, examples aim for clarity and brevity over flexibility (e.g., minimal objectification, minimal optional parameters). For exception specifications defined on particular methods, refer to the *RFA Java Reference Manual*. For more complete examples, please review the examples in the RFA Java release as these example applications contain most of the code described in this guide.

## Chapter 2 RFA Concepts

### 2.1 Refinitiv APIs

#### 2.1.1 The Refinitiv Robust Foundation API

The RFA is an API suite designed for performance and flexibility that clients can use to access, create and provide both market information and custom data. RFA provides a robust framework for component or object-oriented applications developed in C++ or Java. Furthermore, RFA provides an application framework that allows for multi-threaded scalability, item recovery, connection resiliency, and access to different transport mechanisms. By using a scalable thread model that is both thread safe and thread aware, RFA can be used for applications that require high throughput and low latency data access.

In addition, RFA can seamlessly manage multiple connections on an application's behalf. It can handle the routing and recovery for content across these connections. It can save network bandwidth by combining similar requests into a single stream on the network and fanning out the responses to the application.

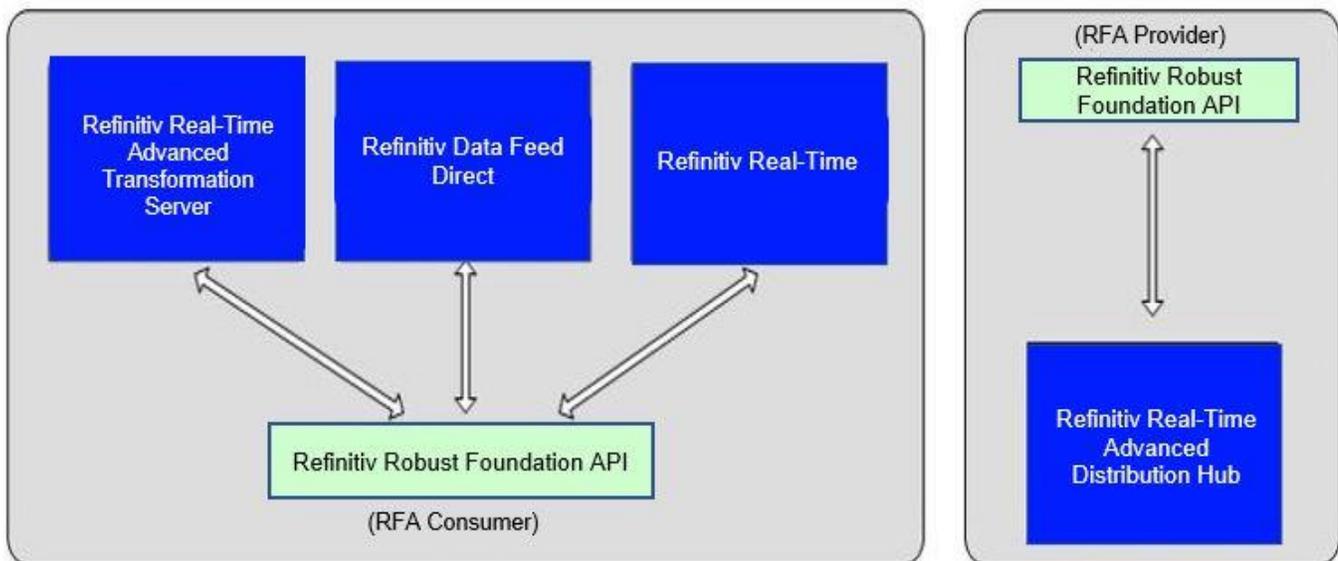
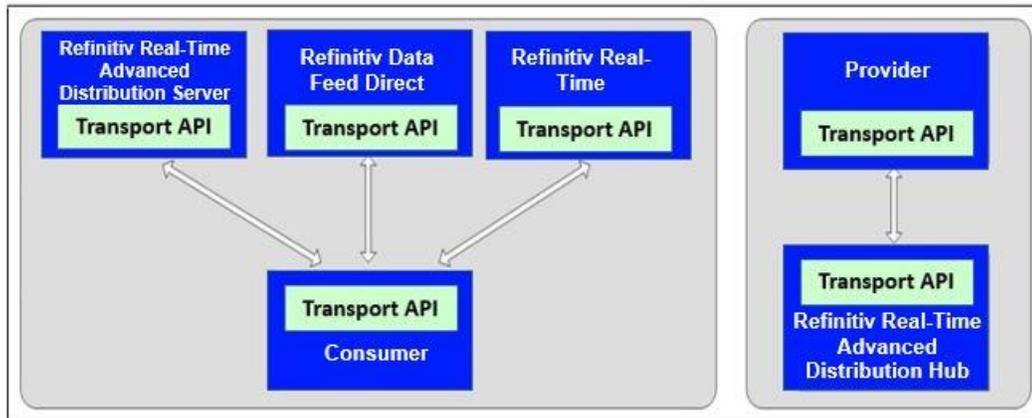


Figure 1: RFA Core Diagram

## 2.1.2 The Transport API

The Transport API is the customer release of Refinitiv's low-level internal API, currently used by the Refinitiv Real-Time Distribution System (and its dependent APIs) for the optimal distribution of OMM /RWF data.

The Transport API is a low-level C language API that provides the most flexible development environment to the application developer. It is the foundation on which all Refinitiv 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 provided by the Core Infrastructure.

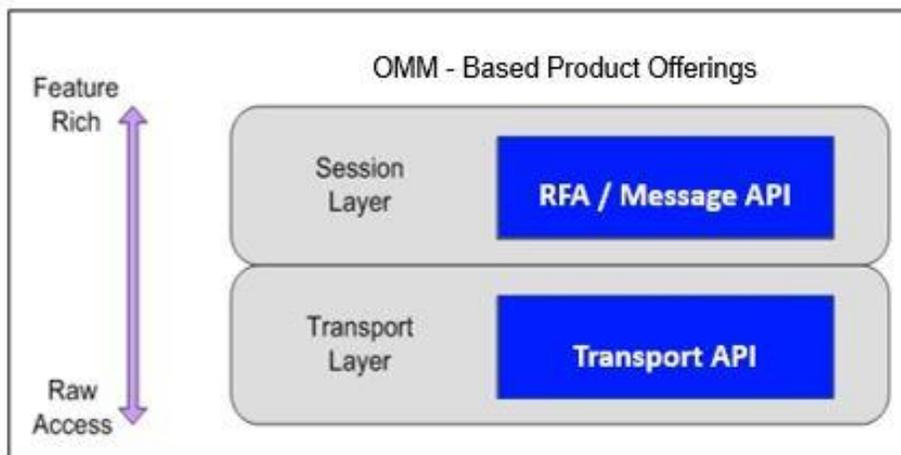


**Figure 2: Transport API Core Diagram**

The Transport API is currently deployed in products such as the Refinitiv Real-Time Advanced Distribution Server, ADH, RFA C++, Refinitiv Data Feed Direct, Refinitiv Real-Time, and Eikon. Due to its well-integrated and common usage across these products, The Transport API allows clients to write applications for use on the Refinitiv Real-Time Distribution System with the highest performance, highest throughput, and lowest latency. Transport API applications are compatible with RFA applications and can be used together in a system.

## 2.1.3 Contrasting RFA and the Transport API

RFA and the Transport API are custom APIs to support the usage of OMM data. The two APIs complement each other by allowing users to choose the type of functionality and layer (Session or Transport) at which they want to access the Refinitiv Real-Time Distribution System. Customers can choose between a feature loaded session level API and a high performance transport level API (Transport API). RFA uses the Transport API as its transport layer and builds its session layer on it.



**Figure 3: OMM - Based Product Offerings**

## 2.1.4 Performance

The following table lists a high-level comparison of existing API products and their performance. Key factors are latency, throughput, memory, and thread safety. Results may vary depending on whether the application uses watchlists and memory queues and can vary according to the hardware and operating system in use. 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
RFA (OMM)	Safe and aware	High	Low	Medium (Watchlist, optional queues)
RFA (SSL)	Safe and aware	Medium	High	Medium (Watchlist, optional queues)
System Foundation Classes C++	None	Medium	High	Medium/High (Watchlist, cache)

**Table 1: API Performance Comparison**

## 2.1.5 Functionality

To make an informed decision on which API to use, tradeoffs must be made between performance and capabilities. While Section 2.1.4 touches on performance aspects, this section focuses on differences in capability.

### 2.1.5.1 General Capability Comparison

CAPABILITY TYPE	CAPABILITY	RFA	TRANSPORT API
Transport	Compression via OMM	X	X
	HTTP via WinInet (Refinitiv Wire Format)	X	X
	RV Multicast	X	
	TCP/IP: Refinitiv Wire Format	X	X
	TCP/IP: Source Sink Library	X	
Application Type	Consumer	X	X
	Provider: Interactive	X	X
	Provider: Non-Interactive	X	X
General	Batch Support	X	X
	Generic Messages	X	X
	Pause/Resume	X	X
	Posting	X	X
	Snapshot Requests	X	X
	Streaming Requests	X	X
	Private Streams	X	X

CAPABILITY TYPE	CAPABILITY	RFA	TRANSPORT API
	Views	X	X
Domain Models	Custom Data Model Support	X	X
	RDM: Dictionary	X	X
	RDM: Login	X	X
	RDM: Market Price	X	X
	RDM: MarketByOrder	X	X
	RDM: MarketByPrice	X	X
	RDM: Market Maker	X	X
	RDM: Service Directory	X	X
	RDM: Symbol List	X	X
Encoders/Decoders	AnsiPage	X	X
	Refinitiv Data Access Control System Lock	X	X
	OMM	X	X
	RMTES	X	
	TS1 Parser	X	

**Table 2: Capabilities by API**

### 2.1.5.2 Layer-Specific Capability Comparison

The following table lists capabilities that are specific to the individual session-layer (Refinitiv Robust Foundation API) or transport-layer (Transport API).

RFA uses information provided from the Transport API and creates a specific implementation of a capability. Although these capabilities are not implemented by the Transport API, Transport API clients can use information provided by the Transport API to implement the same functionality (i.e., as provided by RFA).

CAPABILITY	RFA	TRANSPORT API
Config: file-based	X	*
Config: programmatic	X	X
Group fanout to items	X	*
Load balancing: API-based	X	*
Logging: file-based	X	*
Logging: programmatic	X	X
Quality of Service Management	X	*
Network Pings: automatic	X	*
Recovery: connection	X	*
Recover: items	X	*
Request routing	X	*

CAPABILITY	RFA	TRANSPORT API
Session management	X	*
Service Groups	X	*
Single Open: API-based	X	*
Warm Standby: API-based	X	*
Watchlist	X	*
Controlled fragmentation and assembly of large messages		X
Controlled locking / threading model		X
Controlled dynamic message buffers		X
Controlled message packing		X
Different priority levels per message		X
* Transport API customers can get the same functionality but must implement it themselves. RFA implements this feature.		

**Table 3: Layer-Specific Capabilities**

## 2.2 Features of RFA

The following list of features are features current to RFA.

FEATURE	DESCRIPTION
Application Signing	RFA applications can now exchange an authorization token between the components they are connected to via OMM Login. Customers can work with their account managers to obtain an application authorization token, used to indicate that the application and its specific behaviors have been fully inspected, approved, and 'signed' by Refinitiv. For more information, refer to Section 13.19.
Multicast to Refinitiv Real-Time Advanced Distribution Server Connectivity	A new connection type allows many-to-many connectivity to an ADS Server that supports multicast. This connection type provides gap detection and recovery, message sequencing, and fault tolerant grouping of multicast configured ADS
Cache Component (Value Add)	A Value Added Payload Cache component is added to provide a facility for storing OMM containers (the data payload of OMM messages) utilizing technology based upon the Refinitiv Real-Time Advanced Distribution Server/ADH cache. The cache is populated by applying refresh, update and status messages. Subsequently, applications can then retrieve the last known values from that cache.
Dictionary Download from an ADH	A publishing application can now request the dictionary from the ADH. This provides a central location for publishers to obtain a dictionary. This functionality requires ADH 2.6 or newer.
Batch Close/Reissue (on the wire)	Batch close and reissue requests are now optimized on the wire (if supported by the ADS). If the upstream device supports batch close and reissue on the wire, the batch request is maintained as a single request on the wire. If the ADS does not support batch close/reissue, the batch request will be broken into individual closes or reissues before being put on the wire.
Expanded DACS Login Return Codes	Return codes for login status are expanded to provide client applications with more specific reasons for login failures from the ADS.

FEATURE	DESCRIPTION
Enhanced Symbol List	<p>The Symbol List feature is enhanced to allow an application to request not only the symbol list, but also optionally subscribe to all items in that symbol list on the users behalf.</p> <p>For more details, refer to Section 13.9.</p>
Visible Publisher Identifier (VPI)	<p>Originating publisher information (the VPI) is available via the <a href="#">PublisherPrincipalIdentity</a>. This class provides both the publisher ID and publisher address. This <a href="#">PublisherPrincipalIdentity</a> is made available within the respMsg and postMsg.</p> <p>Consumer applications can specify VPI information on the <a href="#">postMsg</a> via <a href="#">setPrincipalIdentity()</a>. Previous to the 7.6.1 release, VPI information was automatically set on behalf of the user.</p> <p>For more details, refer to Section 13.8.</p>
Performance Enhancement for Outbound Messages at Low Update Rates	<p>The new connection-level configuration parameter <a href="#">flushTimeInterval</a> enables applications to control delays in message flushing which results in lower latency for outbound messages at low update rates. If configured, RFA will attempt to flush messages no later than the configured value. This feature applies to the RSSL, RSSL_PROV and RSSL_NIPROV connections.</p> <p>For more details, refer to the <i>RFA Configuration Guide</i>.</p>
More descriptive Error messages	<p>Select log messages have been updated to be more descriptive.</p>
Performance Tools	<p>A set of OMM Performance Tools are included with the distribution. These open source tools provide clients a means of performance testing their own environment. There are three tools (<a href="#">rfajConsPerf</a>, <a href="#">rfajProvPerf</a>, and <a href="#">rfajNIProvPerf</a>) which provide statistics on images, updates, posting, throughput, latency and more. Since they are open source, clients have the ability to customize them for their own testing needs.</p>
Component Version	<p>An OMMConnectionEvent class provides information about the connected component. OMMActiveClientSessionEvent and OMMInactiveClientSessionEvent provide component version information.</p>
Connection Information	<p>An OMMConnectionEvent class provides information about the connected component's hostname and port. The OMMActiveClientSessionEvent and OMMInactiveClientSessionEvent provide the hostname (no port) as well.</p>
Threadsafe OMMProvider::submit()	<p>The <a href="#">submit()</a> method of the OMM provider is thread safe.</p>
No Event Queue Support for OMM Provider	<p>The OMMProvider handles null event queue (No event queue) on <a href="#">registerClient()</a> interfaces.</p>
RDMFieldDictionary Dictionary Fragmentation	<p>The RDMFieldDictionary class supports fragmentation and verbosity. The <a href="#">encodeRDMFieldDictionary</a> and <a href="#">encodeRDMEnumDictionary</a> methods provide automatic fragmentation of the dictionary for use with multipart refresh messages. In addition, the user can specify verbosity.</p>
Provide-Specified Service IDs	<p>The OMM provider allows the provider application to explicitly set a serviceID on each service they publish. By default, RFA generates a serviceID for each service name that the provider specifies. This feature allows the client the flexibility to specify both the service name and service ID. This can be particularly useful if the provider application creates Refinitiv Data Access Control System Locks.</p>
Outbound NIC Binding for OMM Consumer	<p>Consumer clients with multiple NIC cards can specify which interface to use for outbound requests. The configuration variable 'interfaceName' can be found in the Refinitiv Source Sink Library Connection configuration.</p>
Value Added Admin Component and Domain Representations	<p>Value Added Components aim to provide an alternate and simpler entry point to leverage RFA features with more ease-of-use and simplicity. They are offered alongside the native RFA APIs to optimize user experience, allowing developers to more easily and rapidly develop RFA applications.</p>

FEATURE	DESCRIPTION
Multicast Non-Interactive Provider	Clients can choose to publish each message simultaneously to multiple ADHs (Refinitiv Real-Time ADH 2.2 or higher) across a multicast channel. With the addition of other features in ADS/Refinitiv Real-Time ADH 2.2 as well as Refinitiv Data Access Control System 6.3, clients can control who and what can be published. For more details, refer to the appropriate API, ADH/ADS, and Refinitiv Data Access Control System Documentation.
Batch Reissue, Batch Close	Clients can close and pause/resume multiple Items from a single request.
Normal to Private Stream item-based Recall	RFA supports switching from normal streams to private streams. In the case where a normal request is made to a provider but the provider responds as a private stream, RFA will close the item and inform the consumer. The client can then re-request the item as a Private Stream.
Post User Rights	This feature indicates whether the posting user is allowed to create or destroy items in the cache of record. This is also used to indicate whether the user has the ability to change the permission data associated with an item in the cache of record.
Domain Message Validation	RFA has an interface to validate RDM messages. This is a debugging tool to help users verify that the RDM messages being used are properly formatted. Note that enabling this feature negatively affects performance and should only be used for testing and troubleshooting.
Refinitiv Domain Model	RFA allows clients to consume RDM-provided data. Applications requesting information are referred to as “Service Consumers” that consume services, and applications providing information are “Service Providers” providing services. The service consumers and service providers communicate via messages. The service consumers send request messages and receive response messages. The service providers receive request messages and send response messages.
Hybrid Applications	RFA supports hybrid applications, which are applications that receive request messages from a consumer and forward the same request message to another provider. Similarly, a hybrid application receives response messages from a provider and forwards the same response message to the originating consumer. The contents of the messages can be tuned when desired.
Generic Message	RFA can send Generic messages to either an OMMConsumer from an OMMProvider (or to an OMMProvider from an OMMConsumer) and can contain any data payload as needed (from a simple byte buffer to a complex, nested hierarchy comprised of OMM data constructs). To send Generic messages, RFA must first establish a standard stream. For more information on generic messages, refer to Sections 12.2.5, 12.3.8.4, and 13.5.
Private Streams	Using Private Streams, applications can privately access and exchange market information. Private streams are exclusively established between two points: a consumer and a provider. In contrast with standard streams, data on a private stream (for example, data related to transactions) is not shared with any other consumers.
OMM and RFA Support	Open Message Models is a design approach similar to RDM that separates message representation from data representation. It provides extensible message header information and extensible payload information. Support for OMM includes: the Message Package, the Data Package, new definitions (RDM definitions), and new interfaces in the existing packages (the Session Layer Package and Common Package). RFA continues to support existing RFA packages (the Configuration Package, the Logger Package and external packages AnsiPage API and DACSLOCK API) as before.
Horizontal Scaling	Applications take advantage of RFA’s Horizontal Scaling feature to create multiple instances of consumers and / or providers on multi-core processors. In so doing, these applications can dynamically scale the number of Session and Adapter instances in use. Because each instance of a horizontally-scaled adapter processes messages on its own thread (independent from other adapter instances), applications can use this feature to increase their response and request message throughput. This feature is available for OMM consumer-type applications as well as OMM provider-type applications (both interactive and non-interactive).

FEATURE	DESCRIPTION
	For more information, refer to Section 14.5.6.
Posting	<p>Post messages are messages that can be published into the Refinitiv Real-Time Distribution System or to any cache of record (analogous to MarketData inserts). OMM applications can use the Post capability to send Post messages through the Refinitiv Real-Time Distribution System. Post messages are an OMM message type that can contain any OMM data container type (e.g., Map, ElementList, etc.), opaque data, or another OMM message as their payload. Posting enables end-to-end publishing with no restrictions on data, message size, message type, or domain. For example, posting is available for refresh, status, and update messages on MarketByPrice, MarketPrice, MarketByOrder, SymbolList, MarketMaker, and user-defined domains. Lastly, post messages can be single-part or multi-part messages, and, if requested, Post messages can be sent with an acknowledgement requested from the Posting provider.</p> <p>When using Refinitiv Real-Time Distribution System components (ADS / ADH), message translation between OMM and MarketFeed is supported. For example, posting OMM Level 1 data on the Refinitiv Real-Time Distribution System can result in a conversion to MarketFeed data format.</p> <p>For more information, refer to Sections 13.6 and 6.9.</p>
Event Queue Monitoring	<p>When an application runs with a statistics event queue, the application can be notified when the event queue reaches a user-defined depth. Applications can now query the OMM event to determine how long it has been in the RFA.</p> <p>For more information, refer to Sections 7.9.1 and 7.3.2.2.</p>
Batch	<p>An RFA application can use a single request message to specify interest in multiple items via an itemname list. In response to this message, an RFA consumer will receive a response from RFA comprised of multiple, fully-functional, independent item streams: one for each item specified in the itemname list of the Batch request message. The batch request supports any message model type, except those identified as being unsupported by the RDM Usage Guide (e.g., Login, Directory, and Dictionary).</p> <p>Batch functionality also allows clients to close multiple items in a single close request message. A batch reissue feature also allows client to make a single call to reissue multiple items and at the same time change Pause and Resume state, View, Priority, or request a Refresh.</p> <p>For more information, refer to Section 3.2.3.</p>
Dynamic Views	<p>An application can request a subset of Field(Entry)s or Element(Entry)s of a particular item. Typically, the request with a view specification will receive a response message that contains only those fields or entries defined by the requested View. The use of views increases consumer performance by reducing bandwidth through significantly reducing the field list or element list size per response message. By reducing the number of entries per field or element list, views also reduce overall decoding time.</p> <p>For more information, refer to Section 13.3.</p>
Pause and Resume	<p>Using RFA's Pause and Resume feature, applications can pause updates for subscribed items (e.g., on a window or tab of data hidden or otherwise out of the page's view) and then resume updates at a later time. Using Pause and Resume can save applications bandwidth and the logic of having to unsubscribe and resubscribe to entire sets of items.</p> <p>Applications can also "Pause all" and "Resume all" (only the MarketPrice domain supports Pause and Resume).</p> <p>For more information, refer to Sections 3.2.7 and 13.4.</p>
Optimized Pause and Resume	<p>Optimized Pause and Resume enhances existing Pause and Resume functionality. It extends Pause and Resume rules for just-in-time conflation of the MarketPrice domain to any domain, whether the domain model is Refinitiv or a customer-defined message model. When a provider receives an "optimized" Pause on any domain, it pauses the data flow (with the exception of any state related messages).</p> <p>To help reduce bandwidth spikes, optimized pause and resume also supports a single Pause All and Resume All messages which replace the previous message fanout within RFA on every item in the connection's watchlist.</p> <p>For more information, refer to Section 13.4.</p>

FEATURE	DESCRIPTION
Warm Standby	<p>Warm Standby allows failover to a standby stream in the event the primary stream fails. Because the standby stream is already aware of items watched by the user, during a failover RFA does not need to re-request open items between an OMM provider and consumer. For this reason, Warm Standby reduces overall recovery time.</p> <p>For more information, refer to Section 13.14.</p>
Connection Redirection	<p>Load Balancing addresses the issue of dynamic and balanced provider discovery. Rather than manually configuring an RFA application to use a particular provider, load-balanced RFA can redirect itself to a different provider based on server information it receives from a provider at login. The redirection itself is transparent to the application.</p>
RDM Definitions and RDMFieldDictionary Utility	<p>RFA provides the <b>FieldDictionary</b> utility and <b>FidDef</b> classes, which parse, cache and access field, enumeration, and data definition dictionaries for OMM . These utility classes are used for OMM data in single-threaded environments. Multi-threaded environment Applications should incorporate their own locking mechanism.</p> <p>For more details refer to the <i>RFA Java Reference Manual</i>. The <b>Provider_Interactive</b> and <b>Consumer</b> examples illustrate how to use this utility.</p>

**Table 4: Existing Features of RFA**

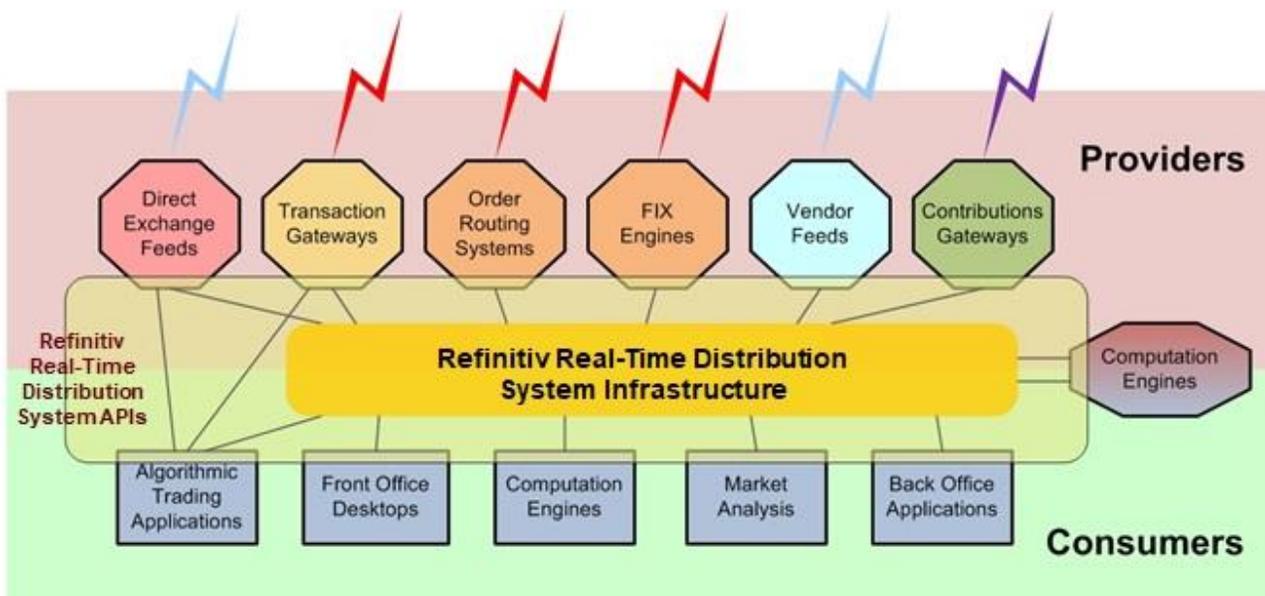
## Chapter 3 Concepts: Consumer / Provider

### 3.1 Consumer / Provider Overview

The following sections detail general concepts regarding consumers and providers. When this document refers to consumers or providers it is referring to OMM consumers and OMM providers unless otherwise stated. The term OMM will be defined in a later section.

OMM Consumer and OMM Provider applications can be written using either the RFA or Transport APIs. For those familiar with previous API products or concepts from Refinitiv Real-Time Distribution System, this section shows how the Transport API and RFA implement the same functionality.

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



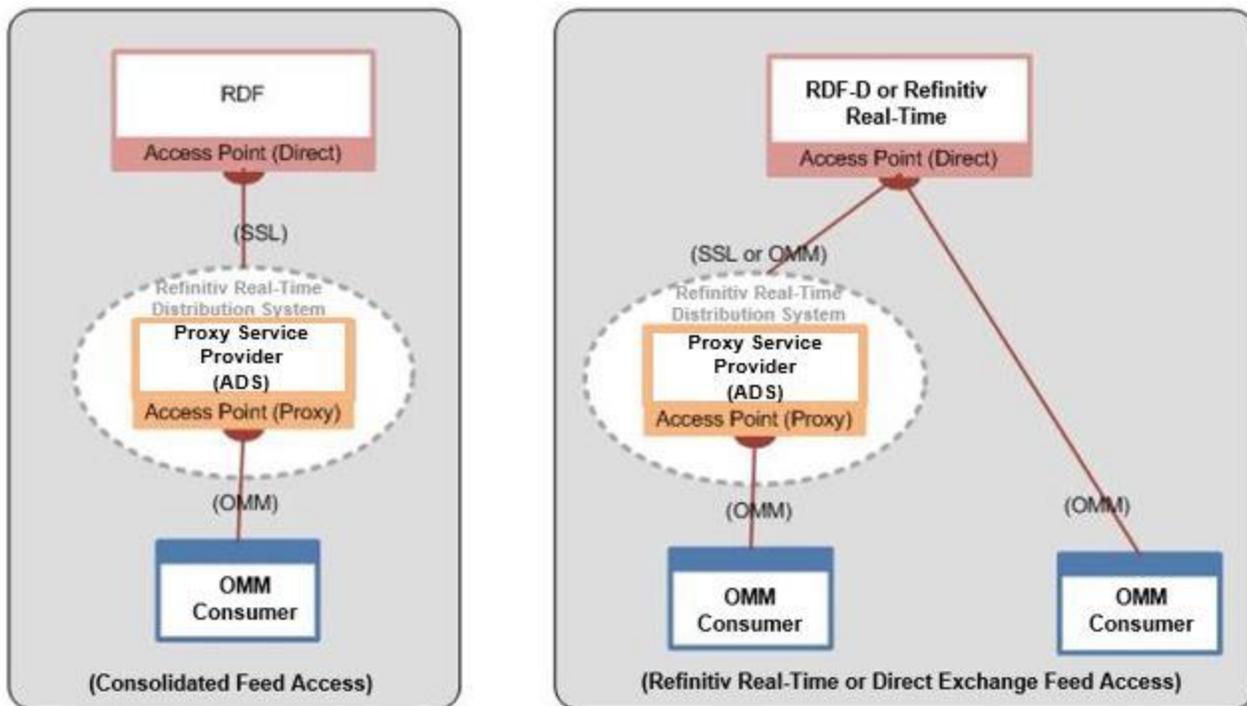
**Figure 4: Refinitiv Real-Time Distribution System Infrastructure**

To access capabilities, consumers always interact with a provider, either directly or via Refinitiv Real-Time Distribution System. However, placing the Refinitiv Real-Time Distribution System between the provider and consumer allows for more complex implementations supporting integration of multiple providers, managing local content, automated resiliency, scalability, control, and protection.

### 3.2 Consumers

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

- A direct consumer access point is implemented by the service-provider application when it supports direct connections from consumers. In the diagram, this is illustrated by a consumer connecting to Refinitiv Real-Time via the direct access point.
- A proxy consumer access point is point-to-point based and implemented by a ADS. In the diagram, this is illustrated by a consumer connecting to the provider by first passing through the proxy access point.



**Figure 5: Consumer**

Examples of consumers include:

- An application that subscribes to data via Refinitiv Real-Time Distribution System, Refinitiv Data Feed Direct, or Refinitiv Real-Time.
- An application that posts data to Refinitiv Real-Time Distribution System or Refinitiv Real-Time. This functionality includes such concepts as Contributions/Inserts or Local Publication into a cache.

### 3.2.1 Subscription: Request/Response

After a consumer application successfully logs into ADS or Refinitiv Real-Time, the consumer can then subscribe and receive data information. This subscription is made to a Service. A Service provides a set of items to its clients.

A consumer application subscription can be either streaming or non-streaming:

- A non-streaming request is a request for one response without the intent to receive updates. This is also called a snapshot request. The data stream is considered closed after the data is received by the consumer because the request has been fulfilled.
- A streaming request results in multiple responses. After such a request, initial data called a refresh or image is returned to the consumer. Any changes or “updates” to the data are subsequently received by the client application. The data stream is considered open until such time as the consumer closes it or the ADS (i.e., Refinitiv Real-Time) closes it. This type of request is typically done when a user subscribes for an item and wants to receive changes for the life of the stream.

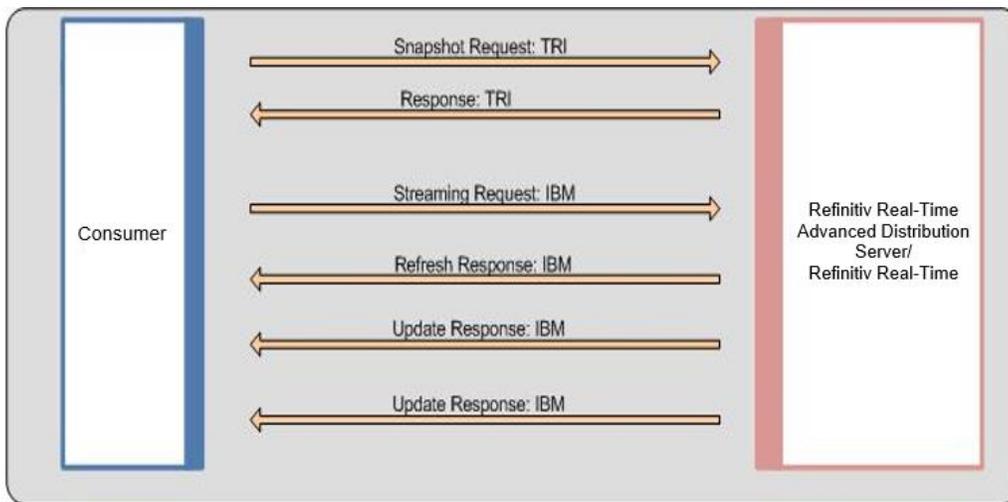


Figure 8: Request/Response

### 3.2.2 Watchlist

RFA maintains a list of all open streaming item requests and their current state made by the client. This list of open items is called a watchlist. The watchlist is used by the item recovery feature described later. The watchlist is also used to help reduce network traffic by eliminating multiple requests for the same item. RFA will never request the same exact item multiple times from an upstream provider. If a client or multiple clients request the same item multiple times, RFA will only request the item once from the provider and then fanout the data to the multiple requestors.

### 3.2.3 Batch

A consumer can request many items with a single client-based request called a **Batch** request. The consumer provides a handful of item names in a single request to ADS/Refinitiv Real-Time. The ADS then sends the items as if they were opened individually so they can be managed as individual items. Batch requests can be streaming or non-streaming.

In the example below, the client makes a non-streaming single batch request for “TRI”, “GE”, and “INTC.O”.

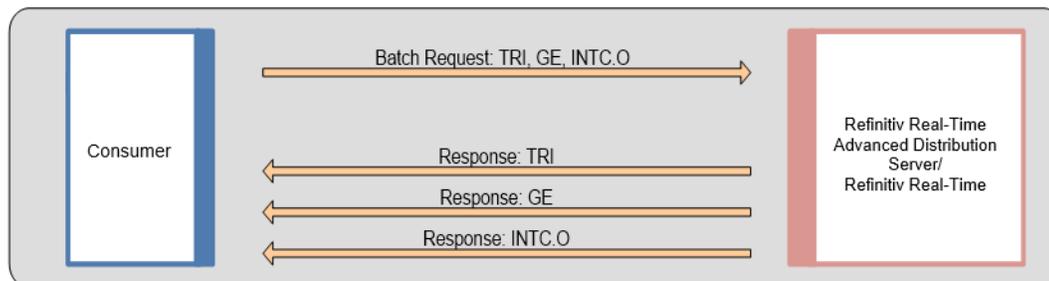
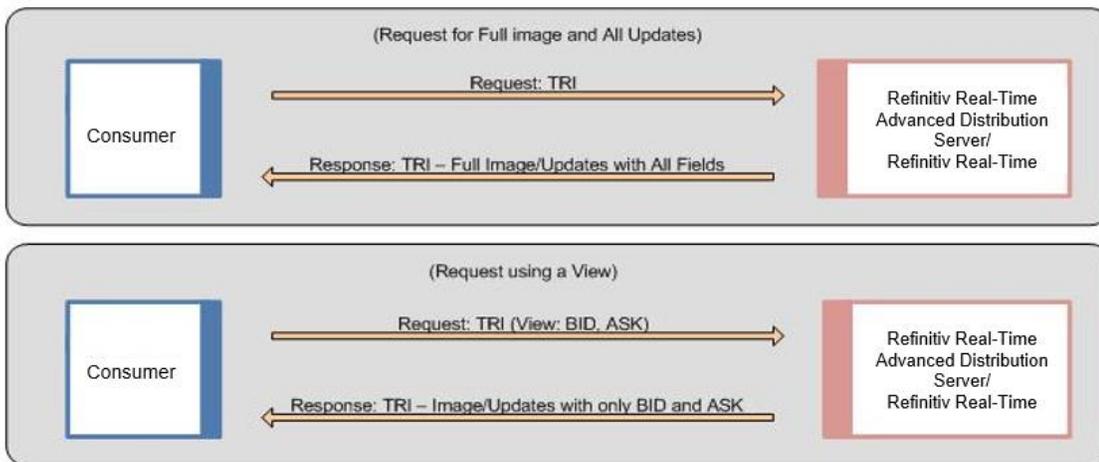


Figure 9: Batch Request

### 3.2.4 Views

One method of reducing the amount of data flowing across the network is to use the **Filtering** capabilities of the Refinitiv Real-Time Distribution System. Filtering out certain **fields** uses bandwidth more efficiently by reducing network traffic. When filtering is enabled in the Refinitiv Real-Time Distribution System, all consumer applications see the same subset of fields for a given item. This is also called Server-Side Filtering.

The consumer application can also perform its own filtering via **Views**. Using a view, a consumer requests a subset of fields with a client-based item request. The consumer specifies a set of fields in a request to the ADS/Refinitiv Real-Time. When the ADS/Refinitiv Real-Time receives the requested fields, it sends the subset back to the consumer. This is also called Consumer-Side (or Request-Side) Filtering. Views are specified per request. Multiple items can be requested each with its own view using multiple item requests.



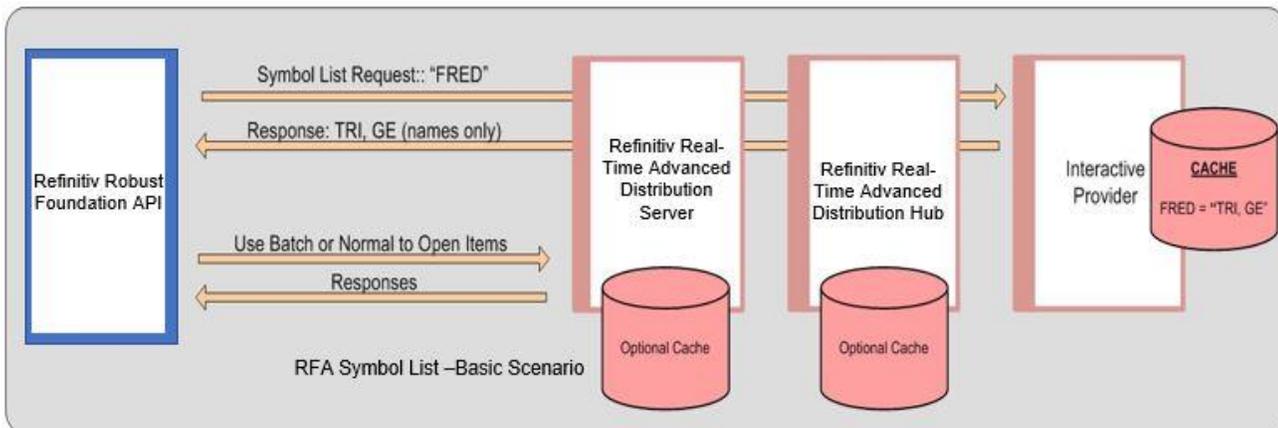
**Figure 6: View Request Diagram**

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

### 3.2.5 Symbol List

Applications use the Symbol List feature to request all item names associated with a Symbol List name and to specify all item data in the same single Symbol List request. This feature allows for OMM counterpart functionality comparable to the functionality clients had for use with the Source Lnk Library 4.5 API with Criteria-Based Requests (CBR).

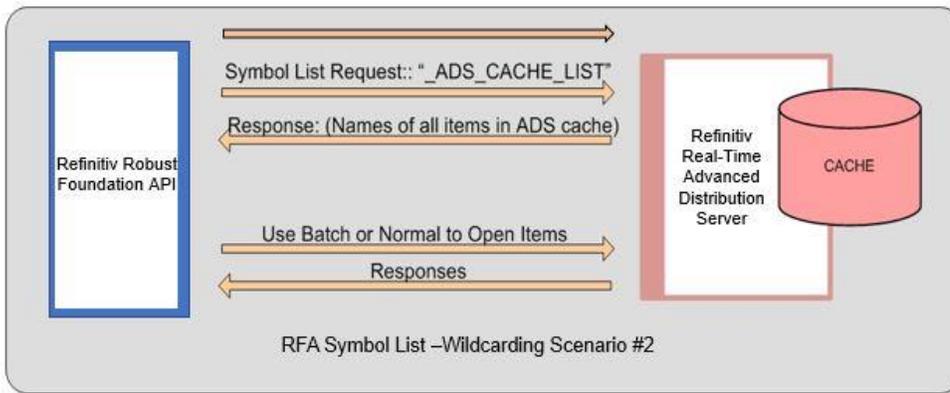
For example, the client has a key name: "FRED". The client makes a Symbol List request for "FRED". The request flows thru the platform to a Provider capable of resolving "FRED". The Provider sends back a list of names that map to "FRED". RFA internally opens the items individually, or makes a batch request to open many items on the client's behalf. The batch or individual item request is resolved by the first cache that contains the data. The following figure illustrates this scenario.



**Figure 7: Symbol List: Basic Scenario**

Symbol Lists can also be used as a migration tool for customers that had a wildcarding capability in their API and Infra component. This feature was previously supported by the STIC/STIC API. Wildcarding is the ability to make a request and have the corresponding Infra component respond with the names of items in its cache. For example:

The ADS provides an additional symbol list (`_SERVER_LIST`) to obtain lists of items stored in specific ADH instances. For further details, refer to the ADS and ADH Software Installation Manuals.

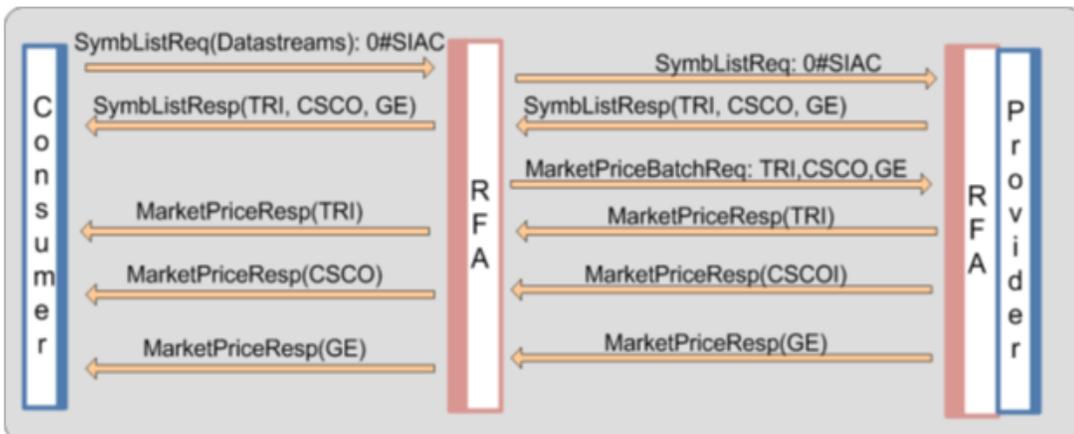


**Figure 8: Symbol List: Wildcarding with an ADS Cache**

### 3.2.6 Enhanced Symbol List

The consumer application can ask for data along with names when making a request for a symbol list. Whenever the consumer application sends a symbol list request and specifies that it wants data with the item list, RFA parses the symbol list response from the upstream provider and opens individual items from the symbol list on the application's behalf. For details on the symbol list request payload, refer to the *RFA Java Edition RDM Usage Guide*.

The following example illustrates how an application gets data along with names on the symbol list domain.



**Figure 9: Symbol List Request for Named Data**

### 3.2.7 Pause/Resume

The **Pause/Resume** feature optimizes network bandwidth. Pause/Resume can be used to reduce the volume of data flowing across the network for items that are already streaming to a client.

To initiate a Pause/Resume, the client 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 is still in the ADS Cache. The ADS continues to receive messages from its upstream device (or feed) and updates the item in its cache (but because of the client's pause request, does not send 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 refresh to the client. From the moment it resumes sending, the ADS sends all item-specific messages as it receives them. By using the Pause/Resume feature, a client can eliminate cases of multiple open/close scenarios, which can disrupt the ADS and impact performance.

There are two main scenarios for this feature:

- Backend Processing
- Display Applications

### 3.2.7.1 Pause / Resume Scenario: Backend Processing

This scenario assumes a client application performs a lot of backend processing.

In this scenario, a client application has multiple items open and is just barely able to keep up with the downstream update rate. The client now needs to run a specialized report, or do some other backend processing. The increased workload on the client application cuts into its ability to effectively process downstream message traffic. The client does not want to ignore the update messages from the ADS and risk having the ADS abruptly cut its connection. Nor does the client want to close its connection or the items because that would require the client to re-connect and/or re-open all items when its backend processing is done.

In this case, the consumer application:

- Sends a single PAUSE\_ALL message to the ADS to pause all the items it has open.
- Performs all needed backend processing.
- Sends a RESUME\_ALL request to resume all the items it had paused.

The ADS then sends a conflated update (refer to Section 5.8.2 Response Message) to the client for all paused items and then continues to send all subsequent messages from the point it resumed.

### 3.2.7.2 Pause / Resume Scenario: Display Applications

This scenario 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 Windows 2 is moved back into view.

The ADS then sends a conflated update to the client for the item "INTC.O" and then continues to send all subsequent update messages from the point it resumed.

## 3.2.8 Posting

Posting replaces the contributions/insert mechanism for publishing into a cache. With Posting, consumers can easily push content into any cache within the Refinitiv Real-Time Distribution System.

When posting, consumer applications use their existing sessions to publish content to any cache residing within the Refinitiv Real-Time Distribution System. 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 Post:** Before a client application can send an On-Stream Post, the client must first open (request) a data stream for an item. After the data stream is opened, 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 configuration of the Refinitiv Real-Time Distribution System.

### 3.2.8.1 Local Publication Scenario

The following diagram helps illustrate the usefulness of posting. The services in green and red 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 red services, the ADS component has enabled caching. In this case, posted messages received from connected applications are cached and distributed to local applications before being forwarded (re-posted) up into the ADH cache. Posting to provider applications via the Purple service can also be done if supported by the provider.

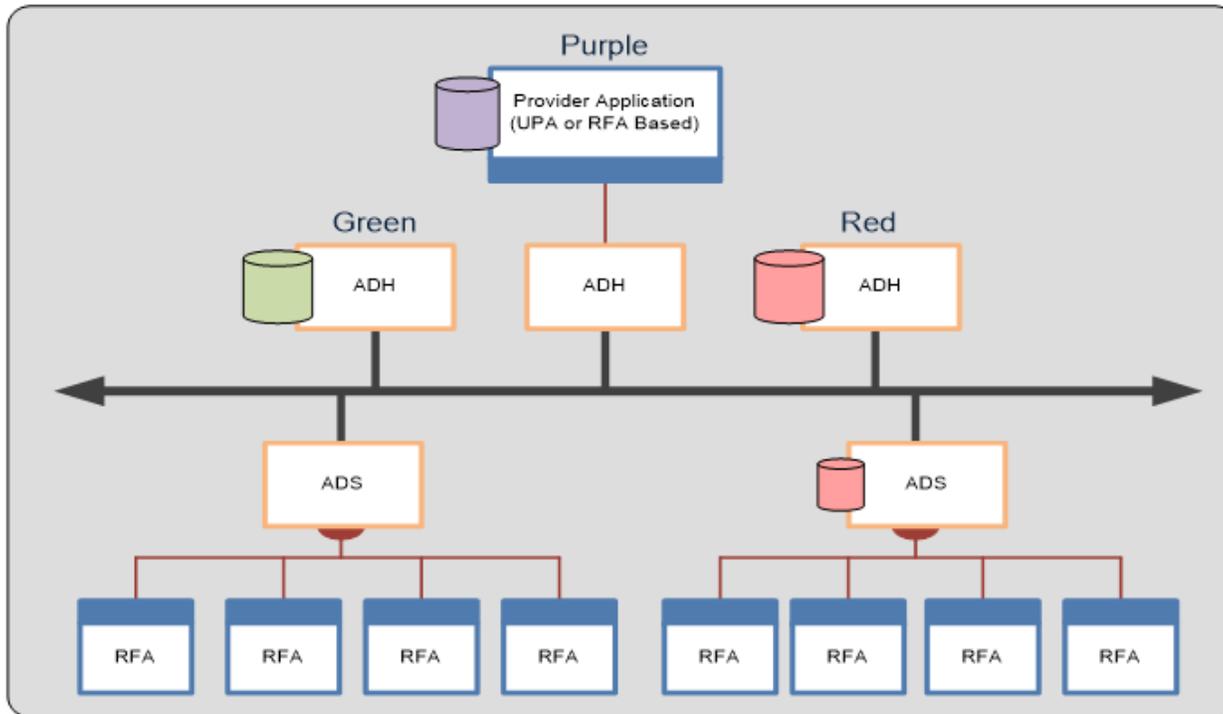
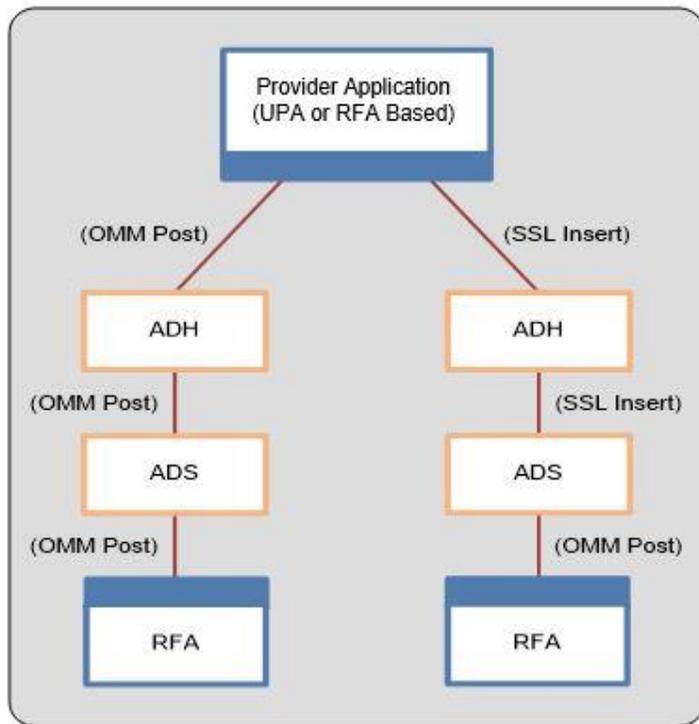


Figure 10: Posting into a Cache

### 3.2.8.2 Contribution/Inserts Scenario

Posting also supports *Open Message Model-Based Contributions*. Through such posting, clients can contribute data to a device on the head-end or a custom-provider. In the following example, the OMM consumer sends an OMM post to the Refinitiv Real-Time Advanced Transformation Server Application.



**Figure 11: OMM Post with Legacy Inserts**

Components (such as the ADS / ADH) in Refinitiv Real-Time Distribution System can convert posts into Source Link Library Inserts for legacy connectivity. This functionality is provided for purposes of migration.

### 3.2.9 Connection Recovery

RFA automatically performs connection recovery for consumers and non-interactive providers. If a connection is lost, RFA periodically attempts to reconnect until the connection is re-established. The retry interval is configurable and defaults to 15 seconds. Once a connection is re-established, RFA will relogin on behalf of the application using the same credentials used during the initial login. If the application is a consumer and the application requested SingleOpen in the login request, RFA will also perform item recovery.

#### 3.2.10 Item Recovery and SingleOpen

SingleOpen is an attribute set when a client logs in, specifying that the client would like automatic item recovery performed for all streaming item requests that the client opens. For example, if a consumer connection to a provider is lost, all open streaming items requested from that provider are unavailable until the connection is reestablished. If the client had requested SingleOpen, then RFA will rerequest all streaming items from the provider as soon as the connection is reestablished. If SingleOpen is not request by the client, the client is responsible for rerequesting all items when the connection is reestablished.

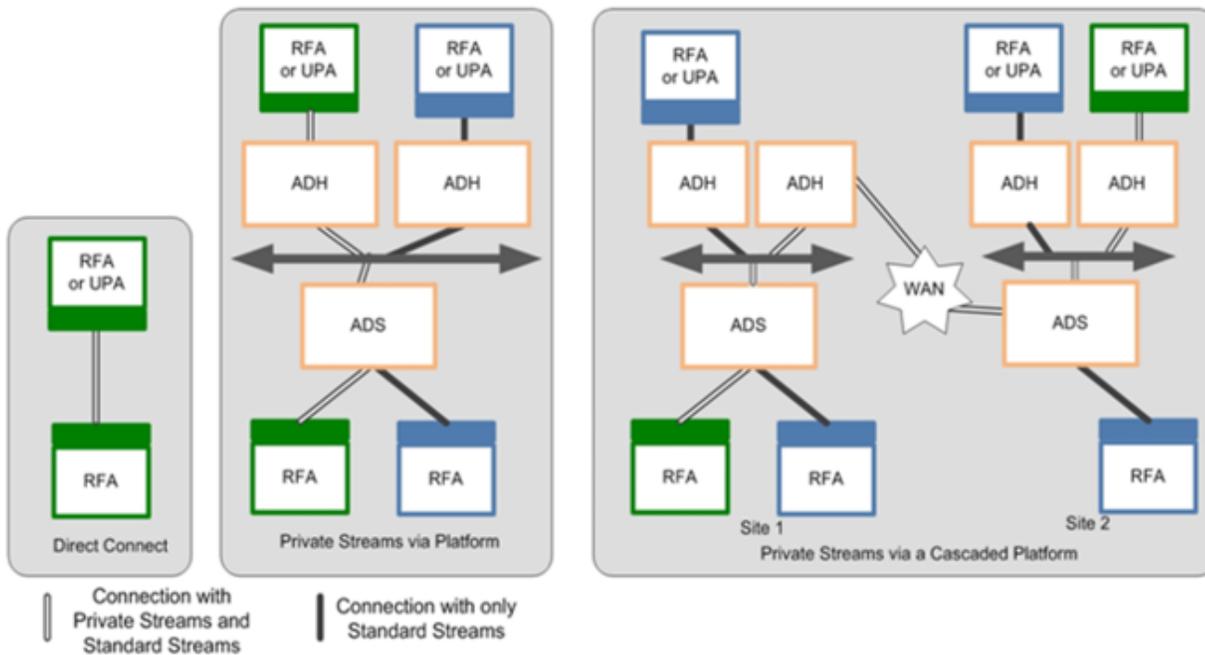
#### 3.2.11 Warm Standby

Warm Standby allows failover to a standby connection in the event the primary connection fails. Because the standby connection is already aware of items watched by the user, during a failover RFA does not need to re-request open items between an OMM provider and consumer. For this reason, Warm Standby reduces overall recovery time.

#### 3.2.12 Private Streams

In contrast to RFA's standard streams, Private Streams provide applications with the ability to establish connections exclusively between two points or users. Data flowing on private streams is not shared with other users. This allows applications to provide, for example, a transactional capability to their users.

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 connect, through the Refinitiv Real-Time Distribution System, or via a cascaded set of Platforms. The following diagram illustrates these configurations.



**Figure 12: Private Stream Scenarios**

A virtual private connection is made up of the existing individual point-to-point and multicast connections in the system. Messages exchanged via a Private Stream flow between a Consumer and an Interactive Provider using these existing underlying connections. However, unlike a regular stream, these messages are not fanned out by the RFA or by any Refinitiv Real-Time Distribution System components to be shared with other consumers or providers.

In the above diagrams, a Green Consumer wishes to create a Private stream with a Green Provider. The Private Stream creates a White path over the existing Black connections and a Private Stream is established. When established, any communications over the Private Stream will flow only between the Green Consumer and the Green Provider. No Blue Providers or Consumers will 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. It is the consumer’s responsibility to re-establish the connection and re-request any data from a Provider. Any type of requests, functionality, or Domain Models can flow across a Private Stream. This includes, but is not limited, to:

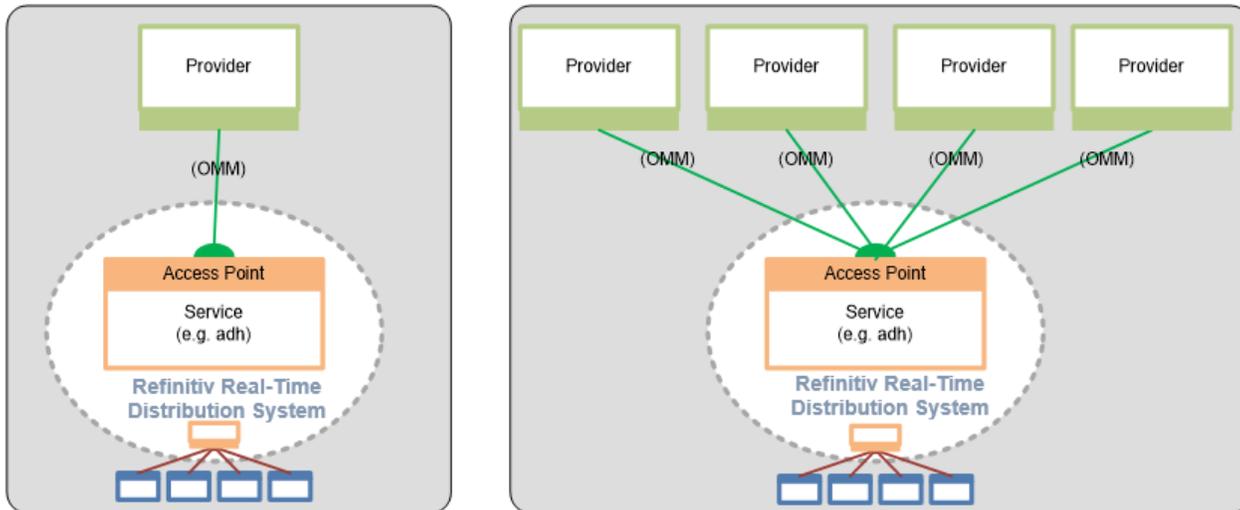
- Streaming Requests
- Snapshot Requests
- Posting
- Generic Messages
- Batch Requests
- Views
- All RDMs & Custom Domain Models

### 3.2.13 Load Balancing

Load balancing enables a system to distribute consumers across multiple providers to balance the load across all available providers. Rather than manually configuring an RFA application to use a particular provider, RFA can redirect itself to a different provider based on server load information (received at login). This redirection is transparent to the application.

## 3.3 Providers

**Providers** make their services available to consumers through Refinitiv Real-Time Distribution System components. To interact with consumers, provider-based applications must connect to a provider access point. All provider access points are considered concrete/direct and are implemented by a Refinitiv Real-Time Distribution System component (like the ADH).



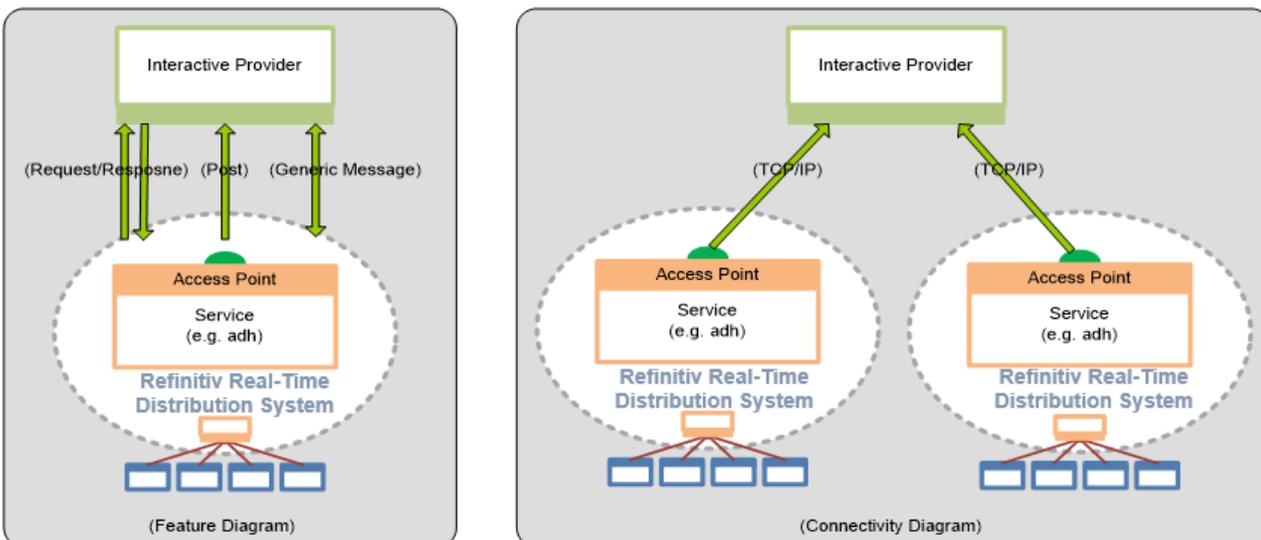
**Figure 13: Provider Access Point**

Examples of Providers include users who:

- Receive a subscription request from Refinitiv Real-Time Distribution System.
- Publish data into Refinitiv Real-Time Distribution System, whether in response to a request or using a broadcast-publishing style.
- Receive post data from Refinitiv Real-Time Distribution System. Providers can handle such concepts as receiving requests for Contributions/Inserts, or receiving Publication requests.

### 3.3.1 Interactive Providers

An **Interactive Provider** accepts and manages multiple connections with components running on the Refinitiv Real-Time Distribution System. The following diagram illustrates this concept.



**Figure 14: Interactive Providers**

An interactive provider receives requests from the Refinitiv Real-Time Distribution System. The interactive provider can send information as to what services, domains, and capabilities it can provide or for which it can receive requests. It can also send information about its data dictionary, describing the format of expected data types. For early Refinitiv Real-Time Distribution System adopters, the interactive provider concept is similar to what was once called a Sink-Driven Server or Managed Server Application. Interactive providers act like servers in a client-server relationship. An interactive provider can accept connections from multiple Refinitiv Real-Time Distribution System components and allow the Interactive Provider Application to manage those connections.

### 3.3.1.1 Request /Response

In a standard request / response scenario, an interactive provider can receive requests from consumers on Refinitiv Real-Time Distribution System (e.g., provide data for "item TRI"). The consumer then expects the interactive provider to send 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. Request and response behaviour is not limited to Market Data-based Domains. Any domain can have this type of behavior.

Interactive providers can receive any consumer-style request described in the consumer section of this document, including item requests, batch requests, views, symbol lists, pause / resume, etc.

### 3.3.1.2 Posts

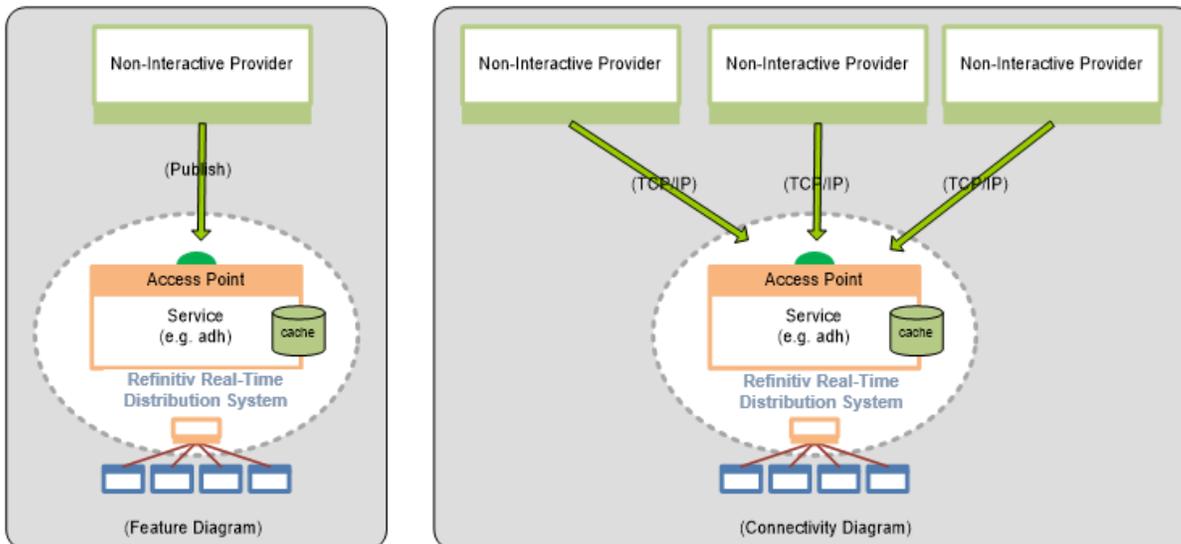
In a standard post scenario, an interactive provider can receive a post message from Refinitiv Real-Time Distribution System. The message will state whether an acknowledgment is required. If required, Refinitiv Real-Time Distribution System will expect the interactive provider to provide a response, in the form of a positive or negative acknowledgement (ACK or NACK). Post behavior is not limited to Market Data-based Domains. Any domain can have this type of behavior. When the interactive provider connects to Refinitiv Real-Time Distribution System and publishes the supported domains, it states whether post capability is supported. Scenarios for posting can be found in Section 3.2.8 .

### 3.3.1.3 Private Streams

In a standard 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 acknowledgement or response if the interactive application cannot provide the expected response to a request.

## 3.3.2 Non Interactive Providers

A **Non-Interactive Provider** connects to Refinitiv Real-Time Distribution System and sends a specific set non-interactive information such as services, domains, and capabilities.

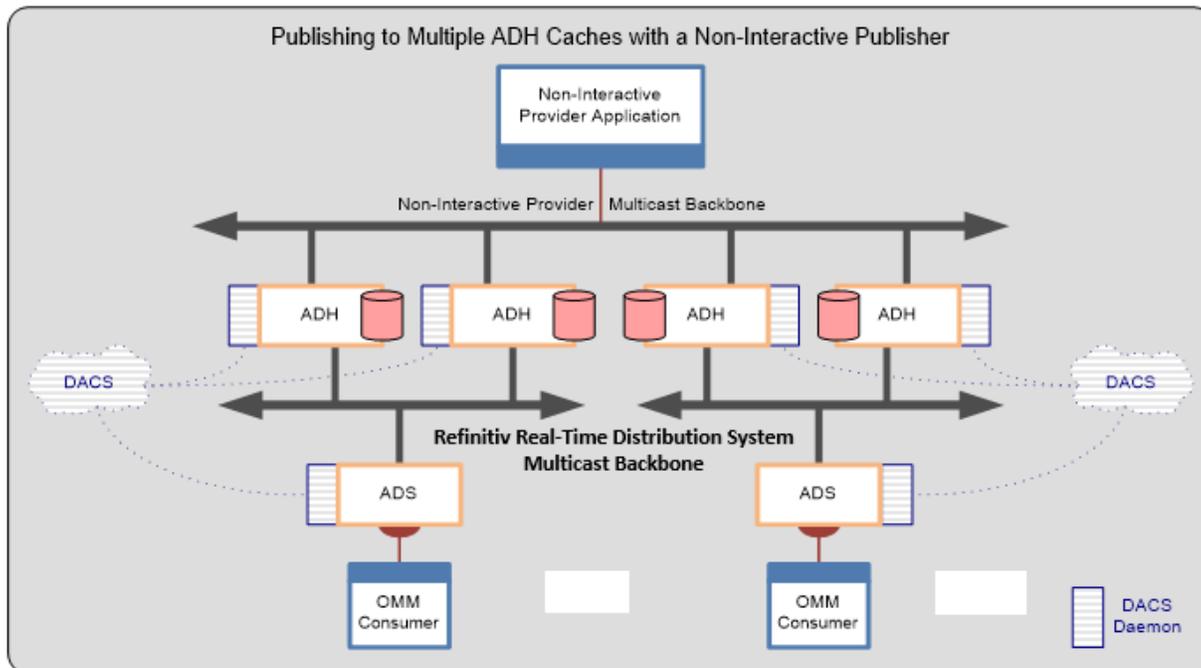


**Figure 15: Non-Interactive Providers**

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

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

Non-interactive providers support both the TCP-based model and the UDP-based multicast model. A UDP-based non-interactive provider can establish a multicast connection to a multicast backbone that can connect multiple ADH components. This allows the application to maintain a single connection, but still have the ability to provide content to multiple devices simultaneously.



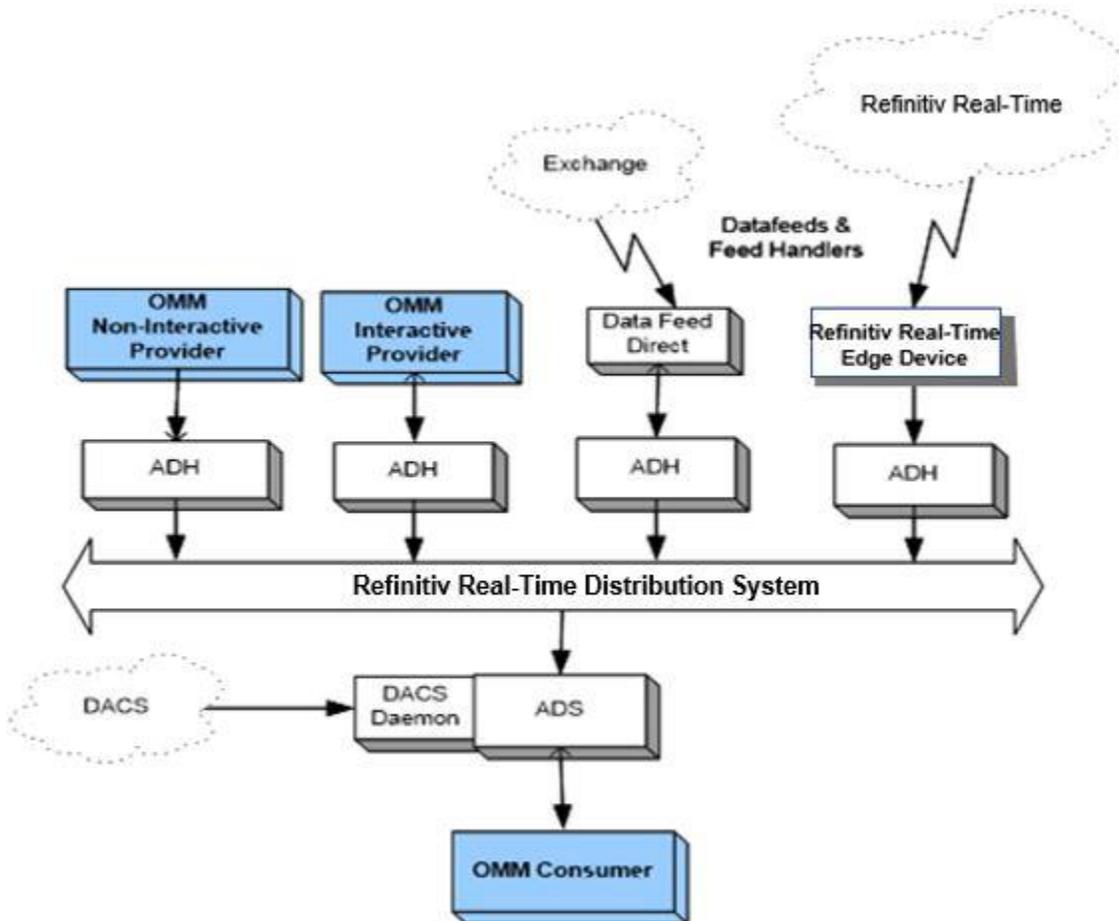
**Figure 16: Publishing to Multiple ADH Hub Caches with a Non-Interactive Publisher**

A single multicast backbone supports connectivity between multiple non-interactive providers and multiple ADH components. The ADH uses a cache synchronization mechanism that allows an ADH to communicate with other ADHs to build or rebuild its cache, allowing it to establish a good cached data state with no impact to any non-interactive providers.

## Chapter 4 System View

### 4.1 Overview of Network System Architecture

Consumer applications typically request and receive information from the network while provider applications typically write information to the network. An interactive provider application receives and interprets request messages and responds back with the needed information. A non-interactive provider application publishes data, regardless of any user requests or which applications consume the data. OMM Provider and OMM Consumer applications can be created using either RFA or the Transport API.

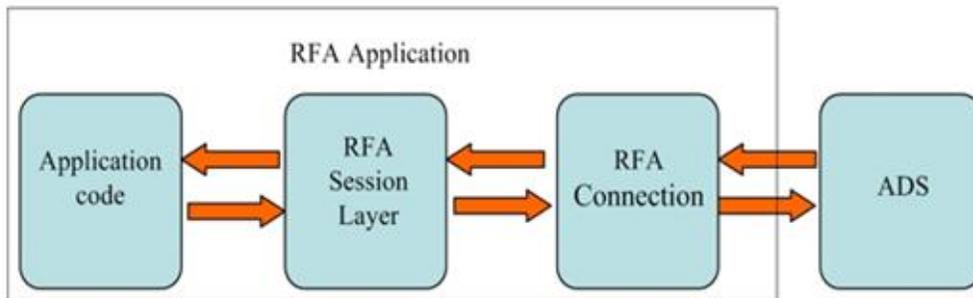


**Figure 17: Typical Components in a Refinitiv Real-Time Distribution System Network**

Figure 17 illustrates a typical deployment of a Refinitiv Real-Time Distribution System and some of its components.

## 4.2 RFA High Level Architecture Overview

RFA is designed to connect to different types of Refinitiv Real-Time Distribution System components. This is accomplished by isolating the component-specific code into **Connection Layer** and providing a common interface to the application with the **Session Layer**.



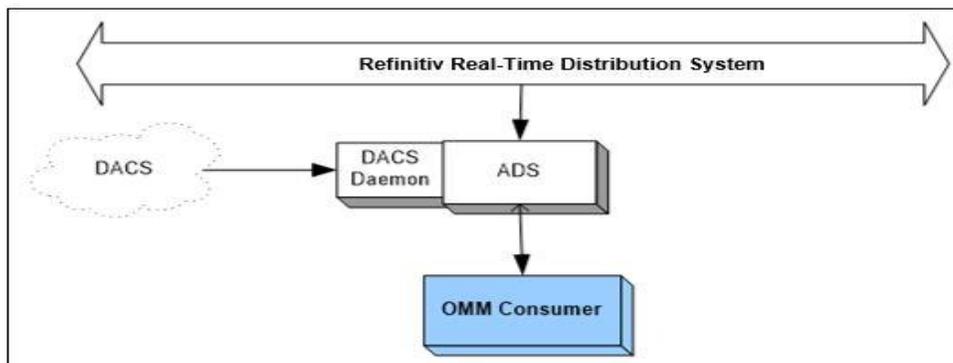
**Figure 18: RFA Components**

The figure above shows the architecture of a simple RFA consumer application communicating with a ADS. The “RFA Connection” provides the proper interface for the ADS and passes the messages to the session layer component. The session layer component provides an API for the application to use. Using this scheme, applications can be configured to communicate with multiple ADS using the single interface that the session layer provides. In most scenarios, multiple connections can be used concurrently. Different types of connections can be used to connect to different types of data feeds. A similar architecture is also used by RFA providers.

The session layer passes data from the ADS to the application using an **Event Distribution Mechanism**. The application receives an **Event** to indicate that there is data for it to process. The application processes the event to get the data. Events are given to the application using either a queue or using a callback mechanism. In this type of general software model, the session layer is called an **Event Source** since it is the source of the events given to the application. If a queue is used for the events, it is called an **Event Queue**. The application **Dispatches** the events in order to receive them and process them.

## 4.3 Refinitiv Real-Time Advanced Distribution Server

The ADS provides a consolidated point-to-point OMM market data distribution solution for trading room systems.

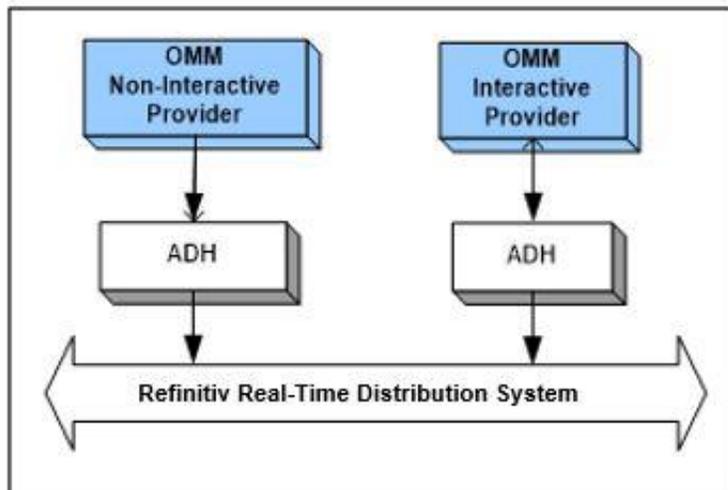


**Figure 19: Refinitiv Real-Time Advanced Distribution Server**

As a distribution device for market data, the ADS delivers data to the consumer from the ADH. Because the ADS leverages multiple threads, the ADS can support many more client applications than any previous Refinitiv’ point-to-point distribution solutions.

## 4.4 Refinitiv Real-Time Advanced Distribution Hub

The ADH is a networked, data distribution server that runs in the Refinitiv Real-Time Distribution System. It consumes data from content providers and reliably fans this data out to multiple ADS over a backbone network (using either multicast or broadcast). OMM non-interactive or interactive provider applications can publish content directly into an ADH, thus distributing data more widely across the network.

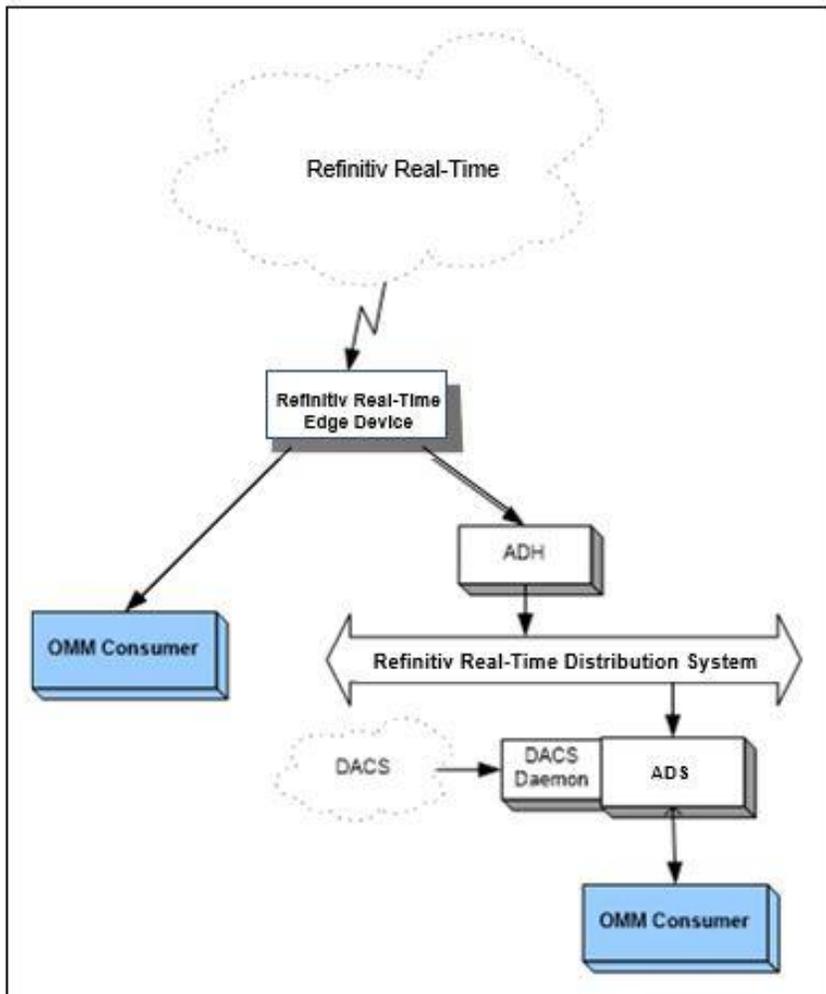


**Figure 20: Refinitiv Real-Time Advanced Distribution Hub**

The ADH leverages multiple threads for both inbound traffic processing and outbound data fan out. By leveraging multiple threads, the ADH can offload the overhead associated with request and response processing, caching, data conflation, and fault tolerance management. By offloading its overhead in such a fashion, the ADH can support higher throughputs than previous Refinitiv' distribution hub solutions.

## 4.5 Refinitiv Real-Time

**Refinitiv Real-Time** is an open, global, ultra-high-speed network and hosting environment that financial firms use to access and share data. Refinitiv Real-Time provides market information from a wide network of exchanges, where all exchange data is normalized using OMM .

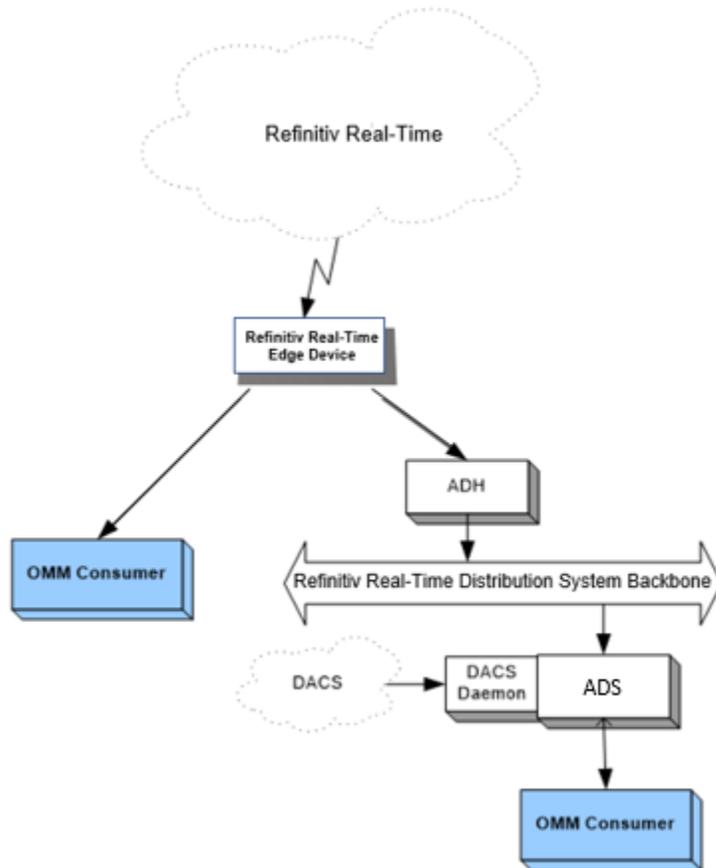


**Figure 21: Refinitiv Real-Time**

The Refinitiv Real-Time Edge Device, based on ADS technology, is the access point for consuming this data. To access this content, consumer applications can connect directly to the Edge Device or via a cascaded Refinitiv Real-Time Distribution System (see Figure 21).

## 4.6 Refinitiv Data Feed Direct

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



**Figure 22: Refinitiv Data Feed Direct**

To access this content, consumer applications can connect directly to the Data Feed Direct or via a cascaded Refinitiv Real-Time Distribution System.

## 4.7 Internet Connectivity via HTTP and HTTPS

Consumer and provider applications can establish connections via tunneling through the Internet.

- ADS and OMM interactive provider applications can accept incoming Refinitiv Source Sink Library Transport connections tunnelled via HTTP (such functionality is available across all supported platforms). Establishing an HTTPS connection to a provider requires the use of a Source Link Library/TLS accelerator between the consumer connection and the providing application.
- Consumer and non-interactive provider applications can establish connections via HTTP tunneling.
- Consumer applications can leverage HTTPS to establish an encrypted tunnel to certain Refinitiv hosted solutions performing key and certificate exchange.
- Consumer-side functionality leverages JDK **java.security**. Users can configure certificates and proxies using configuration tools. Tunneling is supported on all platforms.

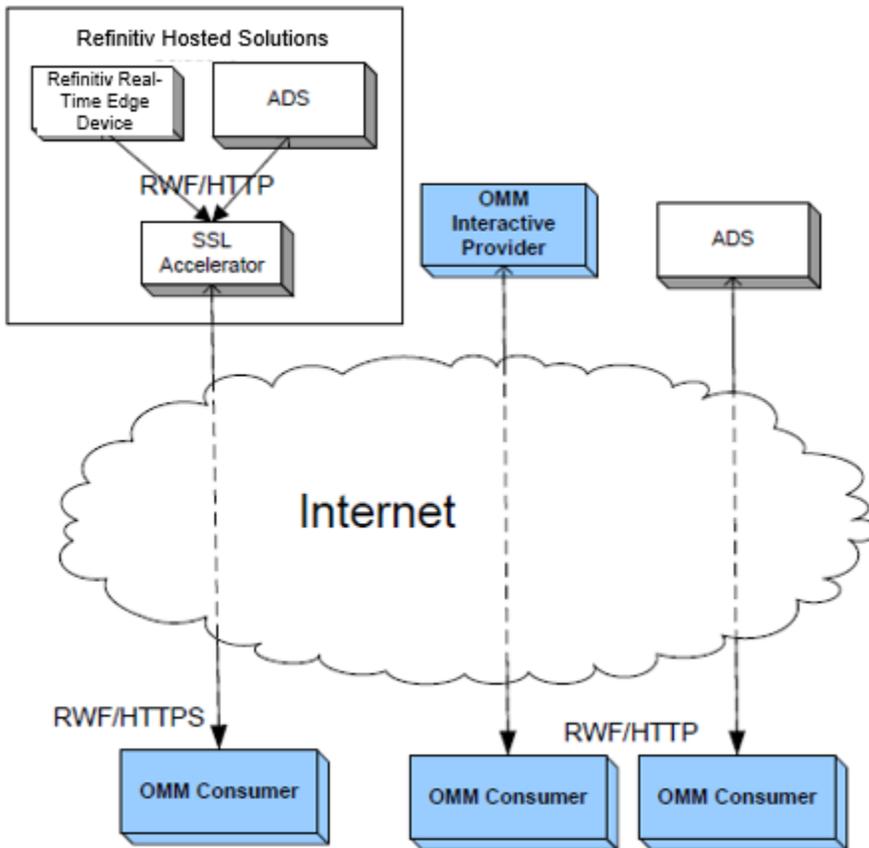


Figure 23: Internet Connectivity

## 4.8 Direct-Connect

RFA and the Transport API allow interactive provider applications and consumer applications to direct-connect. The following diagram illustrates various direct-connect combinations.

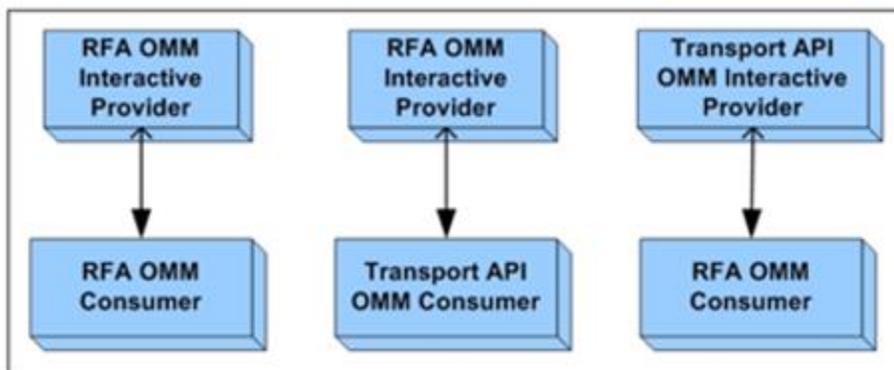


Figure 24: Direct-Connect

## Chapter 5 Model Overview

### 5.1 Open Message Model

The *Open Message Model* is a collection of message headers and data constructs. OMM data constructs can be combined in various ways to model data ranging from simple primitive types to complex data structures. Integer, date, string, map, fieldlist, and series are examples of the data types defined by OMM and will be detailed in Chapter 1.

The layout and interpretation of any specific OMM (also referred to as a domain model) is described in the 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. RFA and the Transport API provide support for OMM constructs and manage the RWF binary-encoded representation of the OMM .

### 5.2 Domain Message Model

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

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

Unless a Domain Message Model specifies otherwise, any implicit market logic associated with a message still applies (e.g., an Update message indicates that it contains previously received data which it is now modifying).

### 5.3 Refinitiv Domain Model

*Refinitiv Domain Model* refers to domain message models defined by Refinitiv. These models support actions such as 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).

### 5.4 User-Defined Domain Model

A *User Defined Domain Model* is a Domain Message Model defined by a third party. These might be defined to solve a need specific to a user or system in a particular deployment and which is not resolved through the use of a RDM.

Customers can have their domain model designer work with Refinitiv to define their model as a standard RDM. By working directly with Refinitiv, customers can help ensure model interoperability with future RDM definitions and with other Refinitiv products.

### 5.5 Refinitiv Wire Format

*Refinitiv Wire Format* is the encoded representation of OMM . RWF is a highly-optimized, binary format designed to reduce the cost of data distribution as 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 rich content types. RWF can distribute field identifier-value pair data (similar to Marketfeed), self-describing data (similar to Qform), as well as more complex, nested hierarchal content.

### 5.6 Refinitiv Domain Models Overview

This section provides a brief overview of the most commonly used domain message models defined by Refinitiv and concepts common to them. For more detailed information and a complete list of RDMs, refer to the *RFA Java RDM Usage Guide*.

The following table provides a high-level overview of the currently available RDMs.

The Login, Directory, and Dictionary domains are considered administrative domains because they are primarily used for managerial tasks instead of sending and receiving market data.

DOMAIN	PURPOSE
Login	<p>Authenticates users and advertises/requests features that are not domain-specific.</p> <p>Use and support of this domain is required for all OMM applications.</p> <p>This is considered an administrative domain. Many Refinitiv components require content, which is also expected to conform to the domain model definition.</p>
Service Directory (also Source Directory)	<p>Advertises information about available services and their state, Quality of Service, and capabilities. This domain also conveys any group status and group merge information.</p> <p>Interactive and Non-Interactive OMM Provider applications require support for this domain.</p> <p>This is considered an administrative domain. Many Refinitiv components require content, which is also expected to conform to the domain model definition.</p>
Dictionary	<p>Used to provide dictionaries that may be necessary to decode data.</p> <p>Dictionary domain use is optional, it is recommended for Provider applications to support this.</p> <p>This is considered an administrative domain. Many Refinitiv components require content, which is also expected to conform to the domain model definition.</p>
MarketPrice	<p>Provides access to Level I market information such as trades, indicative quotes, and top of book quotes. Content includes information such as volume, bid, ask, net change, last price, high, and low.</p>
MarketByOrder	<p>Provides access to Level II full order books. Contains a list of orders, keyed by order Id along with information related to that order, such as price, whether it is a bid/ask order, size, quote time, and market maker identifier.</p>
MarketByPrice	<p>Provides access to Level II market depth information. Contains a list of price points (using the price and its bid/ask side as its key) along with information related to that price point.</p>
MarketMaker	<p>Provides access to market maker quotes and trade information. Contains a list of market makers (using the market maker's ID as its key) along with information such as that market maker's bid and asking prices, quote time, and market source.</p>
SymbolList	<p>Provides access to a set of symbol names and data, typically from an index, service, or cache. This domain must contain symbol names and optionally can contain additional cross-reference information such as permission information, name type, or other venue-specific content.</p>

**Table 5: Overview of RDM**

## 5.7 OMM Data Constructs

This section provides an overview of OMM and RWF data types. Data types in OMM are the building blocks for messages discussed in Section 5.6. For more detailed information, refer to Chapter 1.

RFA data can be represented in OMM form or in encoded RWF form. OMM is used for programmatic manipulation, whereas RWF is a more condensed form used to transmit data on the wire.

- Encoding data refers to the act of converting OMM data into RWF so that it can be transmitted more efficiently on the wire.
- Decoding data is the act of converting data received in RWF form to its OMM representation so that it can be programmatically manipulated more easily.

For example, a consumer would receive data from the wire and decode it to examine the contents. The Message and Data packages provide support for encoding and decoding data.

### 5.7.1 Data Types

OMM offers two categories of data types:

- A **Primitive Type** represents some type of simple information. Primitive types represent values like integers, dates, ASCII string buffers, etc.
- A **Container Type** can be thought of as a list of other data types. Entries in the list can be primitive types or even other container types. The container type determines whether entries are required to be all of the same type or can be of different types.
  - Container types that require all entries to be of the same type are considered uniform or homogeneous.
  - Container types that allow entries of different types are considered non-uniform or heterogeneous.

The concept of a container type containing another container type is called nesting.

**NOTE:** Users familiar with Java collections classes will find that many OMM data types look familiar. However, OMM data types are customized for RWF and contain many differences.

### 5.7.2 Primitive Types

The following table provides an introductory description of each of the RFA primitive type.

PRIMITIVE	DESCRIPTION
Integer and Unsigned Integer	Int and UInt are used for signed and unsigned integers. Both data types can contain binary values.
Float and Double	OMM supports the use of the Float and Double types based on the IEEE-754 standard. However, OMM uses the Real primitive type for price information because Float and Double are imprecise for prices.
Real	Real can be used for decimals that have strict requirements on decimal precision, for fractional values, and for exponentials. It's ideally suited for financial values, which typically have fixed-precision requirements. Real is composed of an integer value and a hint which explains how to convert the integer to a floating point value.
Date, Time, and Datetime	Date represents the date (month, day, and year). Time includes information for hours, minutes, seconds, and milliseconds. DateTime is a combination of a Date and a Time.
String and Buffer	Buffers can contain any length-specified data, including strings.
Array	Array is a simple, ordered collection of another primitive data type. Array is considered a primitive type because all entries must be a primitive type.

PRIMITIVE	DESCRIPTION
	An Array can contain zero to N <sup>1</sup> primitive type entries, with zero indicating an empty Array.
Quality of Service	Quality of Service provides a classification of the tier of service, divided into orthogonal sets of distinct properties. The two properties that make up Quality of Service include: <ul style="list-style-type: none"> <li>• Timeliness: Age of the data</li> <li>• Rate: Maximum period of change in data (for streaming events)</li> </ul>
State	State contains information used to convey information like data and stream health information.
Enumeration	Enumeration is a two-byte unsigned value that can be expanded to a specific string. Enums are ideal for values like currencies and exchange IDs and are defined by the domain message model where they are used.

Table 6: RFA Primitive Types

### 5.7.3 Container Types

This section provides an overview of the container types defined by OMM . In general, the container types are described in order of complexity with the simpler container types described first.

CONTAINER TYPE	DESCRIPTION
Series	Series is a simple list of uniform entries. Each entry in the list contains a value without keys or indexing. Each entry must contain the same container type. A Series can contain zero to N <sup>2</sup> entries, with zero indicating an empty Series.
Vector	Vector is a uniform container type of index-value pair entries, each known as a <a href="#">VectorEntry</a> . Each <a href="#">VectorEntry</a> contains an index correlating to the entry's position within the list. The index for each entry is a primitive integer type. The value for each entry must be a container type. A Vector can contain zero to N <sup>3</sup> entries, with zero indicating an empty Vector.
Map	Map is a uniform container type of associated key-value pair entries known as <a href="#">MapEntry</a> . Each <a href="#">MapEntry</a> contains an entry key which is a primitive type and a value. The value must be a container type.

<sup>1</sup> An [Array](#) 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 [ArrayEntry](#) is bound by the maximum encoded length of the primitive types being contained. These limitations can change in subsequent releases.

<sup>2</sup> 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 a [SeriesEntry](#) has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

<sup>3</sup> 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 could be changed in subsequent releases.

CONTAINER TYPE	DESCRIPTION
	A Map can contain zero to N <sup>4</sup> entries, with zero indicating an empty Map.
FieldList	<b>FieldList</b> is a non-uniform container of field identifier-value paired entries known as a <b>fieldEntry</b> . A Field Identifier, also known as a fieldId (or Field IDentifier), is a signed value referring to a specific name and information type defined by an external field dictionary, such as the RDMFieldDictionary. Values in a <b>FieldList</b> can be a primitive or container type. A <b>FieldList</b> can contain zero to N <sup>5</sup> entries, with zero indicating an empty <b>FieldList</b> .
ElementList	ElementList is a non-uniform container of name-value pairs known as ElementEntry. Each ElementEntry contains an element name, dataType, and value. The type of data is specified by the dataType and can be primitive or container type. An element list is similar to a FieldList where the name and type information is present in each element entry instead of a FieldId. An Element List can contain zero to N <sup>6</sup> entries, with zero indicating an empty Element List.
FilterList	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 presence of specific entries through the inclusion of the filterId in the attribInfo's filter member. Each entry must contain a container type. A FilterList can contain zero to N <sup>7</sup> entries, where zero indicates an empty FilterList, though this type is typically limited by the number of available of filterId values.

Table 7: RFA Container Types

## 5.7.4 Summary Data

Map, Vector, and Series Container Types allow for the use of summary data. Summary data conveys information that applies to every entry housed in the container. This allows information to be conveyed once, instead of included with each entry. Currency type is an example of information that maybe communicated via summary data because it is likely the same for all entries in the container. 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 any content information. When included, the container type of the summary data should match the container type of the payload.

<sup>4</sup> 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.

<sup>5</sup> A **FieldList** 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 4 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of each **FieldEntry** has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

<sup>6</sup> An **ElementList** 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 **ElementList** has a maximum encoded length of 65,535 bytes. These limitations can change in subsequent releases.

<sup>7</sup> 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 a **FilterEntry** has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

## 5.7.5 Defined Data

A local data Definition allows additional optimizations to be performed on the contents of a FieldList or ElementList that result in fewer bytes transmitted on the network. A data definition defines the overall layout of data in the Map, Vector, or Series to which each entry conforms. This eliminates the need for each FieldList or ElementList in the container to have type information specified. The result is that type information only needs to be transmitted once.

## 5.7.6 Iterators

Iterators are used to step through each entry in a container. Read iterators decode entries in a container, while write iterators encode data types into a container. An iterator can be defined to be specific to a container type or independent of container type. Iterators independent of container type are called “single iterators” because only one iterator is needed to encode or decode a message.

Consider the situation where a FieldList contains a FieldEntry which is of type ElementList. Assuming container type-specific iterators were used, a FieldList iterator would be needed to decode the fieldList and an ElementType iterator would be needed to decode the FieldEntry of type ElementList. However, if a container type-independent iterator were used, then only one iterator is needed to decode the entire FieldList.

## 5.8 OMM Messages

This section provides a description of some key RDM message types. A more detailed list and description can be found in Chapter 1. Message concepts apply to both consumers and providers. For example, a request message is called a request message from both a provider or consumer application viewpoint.

### 5.8.1 Request Message

A consumer sends a request message to a provider to request data. In RFA this is known as *interest specification*. Data is specified by the request message using an OMM item interest specification. Requests can be streaming or non-streaming (which are also known as snapshot requests). Requests can also be reissued if the application wishes to change a characteristic of the original request.

### 5.8.2 Response Message

A response message is sent by the provider to the consumer in response to a request. The response message contains data requested by the consumer. There are three types of response messages:

- Refresh (also sometimes called an image)
- Update
- Status

Non-streaming requests are fulfilled by a provider sending a response message called a refresh. A refresh contains all of the data for the item as requested by the consumer.

Streaming requests are typically fulfilled by the provider sending a one or more refresh messages to the consumer, followed by update messages. Update messages are sent only when there is a change to the item. For example, if the consumer sends a streaming request for the item “IBM”, the provider will send back a refresh message containing its current data regarding “IBM”. After sending the refresh, the provider will send update messages to the consumer each time data for “IBM” (such as the current price) changes. For more details on response messages, refer to Sections 6.5, 6.6, and 6.7.

Some providers can aggregate the information from multiple update messages into a single update message using a technique called *conflation*. Conflation typically occurs whenever a stream is paused or if a consuming application cannot keep up with a stream’s data rates.

The consumer provider receives status messages whenever there is a change in a request’s status. A common type of status change occurs when a request is closed. A close status message indicates that the provider will not be sending any more messages for a streaming item request.

### 5.8.3 Generic Message

Using a *Generic Message*, an application can send or receive a customized, bi-directional message between providers and consumers. A Generic message can contain any OMM primitive inside it. One advantage to using Generic Messages is its freedom

from the traditional Request/Response data flow. Generic messages can be sent from both consumers to providers and from providers to consumers.

## 5.9 Services, Concrete Services, and Service Groups

A concrete service is a named grouping of data items supplied by a single data vendor. Providers often provide data about multiple items which can be grouped together into a service identified by a service name and service ID referring to the vendor providing the data. For example, price information regarding IBM can be provided by multiple providers each using a unique service name and ID. When a consumer requests price information about IBM, the consumer will need to specify the service (and therefore which vendor) from which they would like to get this information.

A service group contains a combination of multiple concrete services. Multiple services providing data for the same items are grouped together providing redundancy in case one of the services goes down.

The term service is a somewhat generic term that can sometimes be used to refer to a single service or a service group. The term concrete service refers to single specific service which can be part of a service group.

## 5.10 Packages in RFA Java

The RFA Java API provides several public access packages. The packages are implemented as Java packages, where each of the package contains set of interfaces. The packages are listed below.

- OMM package
- Common package
- Session package
- Configuration package
- Logging package

### 5.10.1 OMM Package

The **OMM Package** defines the OMM messages that flow between various applications, such OMM Consumer, OMM Interactive Provider and OMM Non-interactive Provider. The package also defines data packages, encoders and decoders. These messages are known as OMM Messages.

An OMM Message is an abstract container comprised of a header and a payload. The header contains various attributes. Some of the attributes, such as message type, and message model type, determine what data may be included in the message and thus how the message is encoded and decoded. Some of the attributes contain information used by processing application or RFA. The OMM Message has also meta data – the version to which the message is encoded. The OMM Package includes **OMMMsg** interface that supports OMM messages.

The OMM package also includes hierarchical data interfaces that support a wide variety of data payloads. These data structures include field identifier / value pairs, associative key / value pairs, and self-describing / named value pairs. The data structures support a variety of capabilities and wire optimizations. These include arbitrary nesting hierarchies, fragmentation of 'large' data models, ability to split data definition from raw data content, primitive type packing and fixed place real types. The data structures model the primitive types and container types defined by OMM , such as FieldLists, ElementLists, and Maps.

The OMM package uses RDM constants and dictionary utilities defined in the RDM package and Dictionary package.

### 5.10.2 Common Package

The Common package implements generic functionality used by other RFA packages. Several Common package concepts directly map to abstract interfaces. Other RFA packages derive from these abstract interfaces to realize concrete interfaces.

The Common package implements:

- A **Context** interface that supports integration of RFA packages into a single application and coordinates the interaction between these packages.

- An **Event Distribution** mechanism to deliver asynchronous notification, that is thread-safe and thread-aware. The Event Distribution mechanism is comprised of several interfaces that are listed below:
  - Client
  - Dispatchable
  - Event, Event Queue, Event Queue Group, Event Source
  - Interest Spec, Handle
  - Token
- A **Dispatchable Notification Client** interface used to notify the application of newly available Events. The application typically uses a Notification Client for integrating with the AWT thread.
- A **Status** interface, which is a base class for other package-specific interfaces that provide access to detailed status information.
- A set of **Exception** classes.
- RFA Version Info, an interface supporting version negotiation between RFA and connected entities.
- **Principal Identity**, an interface supporting **User Entitlements**.
- Interfaces to support **Quality of Service**.

For further detail on this RFA package, refer to chapter Chapter 1, "[Common Package](#)".

### 5.10.3 Session Package

The **Session Layer** is the central component of RFA. Its main responsibilities include:

- Allocating sessions and connections at the application's request.
- Keeping track of the application clients.
- Controlling the data exchanged between RFA and the application using the Event Distribution Model.
- Managing RFA internal processing, such as entitlements, Quality of Service, service information, item information, and data exchanged between session layer and connection layer of RFA.

The Session Package provides an application with access to the Session Layer. The application uses the Session Package interfaces to allocate a concrete session, register itself, and to exchange data to consume/subscribe, publish/provide and contribute information.

The Session package is comprised of the following interfaces and packages:

- Session interface
- OMM package
- Event package

### 5.10.4 Configuration Package

RFA Java utilizes the Java Preferences API and the Configuration package to manage its configuration. It includes configuration tools that simplify the configuration process.

The Configuration package provides a programmatic interface for configuring the application.

For the full list of configuration parameters, refer to 2 - *RFA Java Reference Manual*.

### 5.10.5 Logging

RFA Java utilizes the Java Logging API to manage its logging. See Chapter 1 for details.

## 5.10.6 Other RFA Packages

- Dictionary package  
Provides utilities to read dictionaries from files, and encode/decode dictionaries in network format. The package supports field dictionaries, enumeration dictionaries, and data definition dictionaries.
- RDM package  
Provides enumerations, constant values, and utility functions to handle RDM type data.
- ANSI Page package  
The ANSI Page package provides page-based encoding and decoding. This interface is described in a separate document [5] *AnsiPage API JAVA*.
- DACSLOCK package.  
The DACSLOCK package provides management of authorization lock information. Refer to [6] *DACS LOCK API* for details.

## Chapter 6 OMM Package

### 6.1 OMM Package Overview

**NOTE:** Reading the 8 *RFA Quick Start Guide* is highly recommended. It provides good introduction to the concepts described in the remaining of the document.

The OMM package supports OMM functionality through **messages**, **data**, **encoders** and **decoders**, and **memory pools**.

**Messages** communicate information between components in the system: e.g. to indicate status, provide permissioning, and a variety of other purposes. Many messages have associated semantics that allow efficient use in market data systems; messages are used to request information, respond to information, or provide updated information. Other messages have relatively loose semantics, allowing for more dynamic use either inside or outside market data systems.

The information exchanged via messages between consumer applications and provider applications is known as **data**. Data can be represented by simple types, like integers, or complex types, like maps. Data can be contained in a message or nested in other data.

Raw information needs to be converted into messages/data understandable by RFA by a process known as encoding. The opposite can be achieved by decoding.

RFA provides memory management for some types of objects. The OMM package provides a pooling mechanism.

### 6.2 OMM Message Interface: OMMMsg

For an overview of OMM Messages, refer to Section 5.6. OMM Message is supported though the [OMMMsg](#) interface. This interface contains set of accessor methods, used to set and get message data elements, and utility methods.

#### 6.2.1 OMM Message Elements

A message is comprised of a message header and an optional payload. The header contains the message type and message model type. Combinations of these two mandatory attributes uniquely identifies the message, and determines a set of mandatory and optional data elements that the message contains. The message model types are available in the [RDMMsgTypes](#) interface. The message types are available in the [OMMMsg.MsgType](#) interface.

Payloads are available in certain types of messages, in the form of [OMMData](#). The following table describes the [message](#) elements, and how to access them using OMMMsg interface.

ELEMENT	TYPE	DESCRIPTION
<b>Mandatory Message Header Elements</b>		
MsgType	byte	Identifies the specific type of a message. For more details about the various message types, refer to Table 10 in Section 6.2.3. Accessors methods: <ul style="list-style-type: none"> <li><a href="#">getMsgType()</a>: Returns a byte indicating message type</li> <li><a href="#">setMsgType()</a>: Takes byte parameter and sets the message type to this value</li> </ul> Message type values are defined in <a href="#">OMMMsg.MsgType</a> .
Msg Model Type	short	Identifies the specific domain message model type. If the value is less than <b>128</b> , domain is a Refinitiv defined domain model. If the value is <b>128 - 255</b> , domain is a user defined domain model. Domain model definition is decoupled from the API and domain models are typically defined in some type of specification document. Refinitiv defined domain models are specified in the <i>RFA Java RDM Usage Guide</i> . Accessors methods: <ul style="list-style-type: none"> <li><a href="#">getMsgModelType()</a>: Returns a short value, indicating message model type</li> <li><a href="#">setMsgModelType()</a>: Takes a short value parameter and sets the message model type to this value</li> </ul> Message model type values are defined in <a href="#">RDMMsgTypes</a> .
Indication Flags	int	Provides additional information about the characteristic of the message. For details, refer to Section 6.2.5. Accessors methods:

ELEMENT	TYPE	DESCRIPTION
		<ul style="list-style-type: none"> <li><code>isSet()</code>: Takes defined int parameter (for example <code>OMMMsg.Indication.VIEW</code>) and returns true if the flag is set on the message, otherwise false.</li> <li><code>setIndicationFlags()</code>: Takes an int value that represents all indication flags and sets the flags in the message.</li> </ul> Indication flags are defined in <code>OMMMsg.Indication</code> .
Hint Flags	boolean	Hints flags provide easy access to determine if a particular element is present in the message. Each flag is validated individually. For details, refer to Section 6.2.6. Accessors methods: <code>has()</code> : Takes defined int parameter (for example <code>OMMMsg.HAS_QOS</code> ) and returns true if the flag is set on the message, otherwise false The hint flags are set on the message when the corresponding element is set. Hint flags are defined in <code>OMMMsg</code> .
Data Type	short	Defines the type of <code>OMMData</code> in the payload. If the message contains no payload the interface returns <code>OMMTypes.NO_DATA</code> . For details, refer to Chapter 1. Accessors methods: <code>getDataType()</code> : Returns a short value, indicating data type OMM data types are defined in <code>OMMTypes</code> .
<b>Optional Message Header Elements</b>		
Attribute information	<code>OMMAttribInfo</code>	This is additional information about a message. The <code>OMMAttribInfo</code> optional properties are indicated by hint flags. For details, refer to Section 6.2.4. Accessors methods: <ul style="list-style-type: none"> <li><code>getAttribInfo()</code>: Returns attribute information element</li> <li><code>setAttribInfo()</code>: Two overloaded methods that set attribute information element</li> </ul>
Conflation Count	short	The number of updates that have been conflated. Accessors methods: <ul style="list-style-type: none"> <li><code>getConflationCount()</code>: Returns conflation count element</li> <li><code>setConflationInfo()</code>: Sets the conflation count element</li> </ul>
Conflation Time	int	The duration in seconds over which the updates are conflated. Accessors methods: <ul style="list-style-type: none"> <li><code>getConflationTime()</code>: Returns conflation time element</li> <li><code>setConflationInfo()</code>: Sets conflation time element</li> </ul>
ID	long	The post ID on post messages. Accessors methods: <ul style="list-style-type: none"> <li><code>getId()</code>: Returns ID element</li> <li><code>setId()</code>: Sets ID element</li> </ul>
Item Group	<code>OMMItemGroup</code>	A value assigned by a service to the stream. Used to efficiently update the state of many item streams that originate from a single provider. For details, refer to Section 10.9.1. Accessors methods: <ul style="list-style-type: none"> <li><code>getItemGroup()</code>: Returns Item group element</li> <li><code>setItemGroup()</code>: Sets Item group element</li> </ul>
Permission Expression	byte []	The optional authorization data for market information. It is used in conjunction with a login context to determine if a user is allowed access to some particular information. Typically, the permission expression is at stream scope and is accessible via the message interface.

ELEMENT	TYPE	DESCRIPTION
		<p>For applications that desire finer grain authorization, the permission expression may be optionally available in some entries within a dataformat. In this case, the permission expression is accessible via the entry interface.</p> <p>For further information, refer to Section 10.5.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getPermissionData()</code>: Returns byte array with permission data</li> <li>• <code>setPermissionData()</code>: Takes the byte array parameter and sets the permission expression element</li> </ul>
Principal Identity	PrincipalIdentity	<p>Provides publisher information such as the publisher address and id. This is available on the post OMM Message received by the provider application.</p> <p>It is optional on Update, Refresh, and Status Response Messages and is indicated by the <code>OMMMsg.Indication.HAS_PUBLISHER_INFO</code> flag.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getPrincipalIdentity()</code>: Returns principal identity element</li> <li>• <code>setPrincipalIdentity()</code>: Sets principal identity element</li> </ul>
Priority	OMMPriority	<p>The degree and type of interest in an item stream. For details, refer to Section 10.9.2.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getPriority()</code>: returns priority element</li> <li>• <code>setPriority()</code>: sets priority element</li> </ul>
Quality of Service	OMMQos	<p>The Quality of Service received by a consumer or sent by a provider. For details, refer to Section 10.3.5.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getQos()</code>: Returns quality of service element</li> <li>• <code>setQos()</code>: Sets quality of service element</li> </ul>
Quality of Service request	OMMQosReq	<p>A range request defining the desired Quality of Service. For details, refer to 10.9.3.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getQosReq()</code>: Returns quality of service request element</li> <li>• <code>setQosReq()</code>: Sets quality of service request element</li> </ul>
Response Type Number	short	<p>Additional information about the refresh reponse and update response messages. If the response has been provided as response to a request message, this is referred to as a <b>solicited</b>. Otherwise the response type is referred to as an <b>unsolicited</b>. For details, refer to Section 6.2.3.</p> <p>The Response Type Number element has defined two values:  <code>OMMMsg.RespType.SOLICITED</code> and <code>OMMMsg.RespType.UNSOLICITED</code>.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getRespTypeEnum()</code>: Returns short value that indicates response type element</li> <li>• <code>setRespTypeEnum()</code>: Sets the response type</li> </ul>
Sequence Number	long	<p>This may be populated by the application to indicate an order the messages were created. Used for multi-part post messages. Refer to Section 6.9 for details.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li>• <code>getSeqNum()</code>: Returns the sequence number</li> <li>• <code>setSeqNum()</code>: Sets the sequence number</li> </ul>
Secondary Sequence Number	long	<p>This may be populated by the application to indicate which message (determined by Secondary Sequence Number) was last received. Refer to Section 6.8 for more information.</p> <p>Accessors methods:</p>

ELEMENT	TYPE	DESCRIPTION
		<ul style="list-style-type: none"> <li><code>getSecondarySeqNum()</code>: Returns the secondary sequence number</li> <li><code>setSecondarySeqNum()</code>: Sets the secondary sequence number</li> </ul>
State	OMMState	<p>State Information used for message, service and item group. Refer to Section 10.3.6 for details.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li><code>getState()</code></li> <li><code>setState()</code></li> </ul>
User Rights	int	<p>This is used in Post-type messages. It indicates whether the posting user is allowed to create or destroy items in the cache of record. This is also used to indicate whether the user has the ability to change the permission data associated with an item in the cache of record. Refer to Section 6.2.7 for details.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li><code>getUserRightMask()</code></li> <li><code>setUserRightsMask()</code></li> </ul>
Version	byte[]	<p>Version related to this message. Version is meta data, as it is not an element that is part of encoded message, but determines how the message is encoded. Refer to Section 7.7 for information on RWF versioning.</p> <p>Accessors methods:</p> <ul style="list-style-type: none"> <li><code>getMajorVersion()</code>: Returns a byte representing major version the message is encoded according with</li> <li><code>getMinorVersion()</code>: Returns a byte representing minor version the message is encoded according with</li> <li><code>setAssociatedMetaData()</code>: Two overloaded methods that set the version info on the message. One method takes major and minor versions parameters. The second method take a login handle, from which RFA retrieves the version info.</li> </ul>
<b>Payload (Optional)</b>		
Payload	OMMData	<p>Information satisfying the business purpose, e.g. Level 1 data.</p> <p>Accessors methods:</p> <p><code>getPayload()</code>: returns the message payload</p>

Table 8: Message Elements

## 6.2.2 OMMMsg Utility Functions

OMMMsg interface provides the following utility functions:

FUNCTION NAME	DESCRIPTION
<code>clear()</code>	Clears the memory of the <b>Attribute Information</b> object. Used before reusing memory allocated by memory pool.
<code>getBytes()</code>	Returns the encoded message as a byte array.
<code>getBytes()</code>	Takes a byte array as an argument and fills the array with the encoded message. This method leaves the memory management to the client application.
<code>getEncodedLength()</code>	Returns the length of encoded data.
<code>initFrom()</code>	Initializes an instance of OMMMsg from an existing instance. Performs a deep copy, i.e., the referenced objects are copied into new objects with identical data.
<code>isFinal()</code>	Is used to test response message. Determines whether the response message results in closing the stream.

FUNCTION NAME	DESCRIPTION
isValid()	Validates whether the message properties conform to a message type and message model type.
Indication.indicationString()	Returns a string representation of indication flags that are set
MsgType.toString()	Returns a string representation of the message type definition
RespType.toString()	Returns a string describing the refresh response type

Table 9: OMMMsg Utility Functions

## 6.2.3 Message Types

The table below summarizes supported message types. The types are used by all domains.

MESSAGE TYPE	DEFINITION	DESCRIPTION
<b>Request Message Types</b>		
Request	OMMMsg.MsgType.REQUEST	Used by a Consumer to express interest in a new stream, modify some parameters on an existing stream, or express non-ongoing interest in an item (sometimes referred to as a “snapshot”). Indication flags are used with this message type to specify if the request is streaming or non-streaming, and if a refresh is requested or not. See Section 6.3 for more information.
Close Request	OMMMsg.MsgType.CLOSE_REQ	Used by a Consumer to indicate no further interest in a stream. The stream should be closed as a result. See Section 6.4 for more information.
<b>Response Message Types</b>		
Refresh Response	OMMMsg.MsgType.REFRESH_RESP	Used by an Interactive Provider to respond to a consumer’s request for information (solicited) or provide a data resynchronization point (unsolicited). Used by Non-Interactive Providers to initiate data flow on a new item stream. Conveys state information, Quality of Service, stream permissioning information, and group information in addition to payload. See Section 6.5 for more information.
Update Response	OMMMsg.MsgType.UPDATE_RESP	Used by Interactive or Non-Interactive Providers to convey changes to information on a stream. Update messages typically flow on a stream after a refresh has been delivered. See Section 6.6 for more information.
Status Response	OMMMsg.MsgType.STATUS_RESP	Used to indicate changes to the stream or data properties. It is used by Providers to close streams and can also be used to indicate successful establishment of a stream when there is no data to convey. This message can indicate changes in <a href="#">streamState</a> or <a href="#">dataState</a> , changes in a stream’s permissioning information, and changes to the item group that the stream is part of. See Section 6.7 for more information.
Ack Response	OMMMsg.MsgType.ACK_RESP	Used by a provider to inform a consumer of success or failure for a specific Post message. See Section 6.10 for more information.
<b>Other Message Types</b>		
Generic	OMMMsg.MsgType.GENERIC	A bi-directional message that does not have any implicit interaction semantics associated with it, thus the name generic. Once a stream is established via a request-refresh interaction, this message can be sent from Consumer to Provider as well as from Provider to Consumer and can also be leveraged by Non-Interactive Providers. See Section 6.8 for more information.

MESSAGE TYPE	DEFINITION	DESCRIPTION
Post	OMMMsg.MsgType.POST	Used by a Consumer to push content to upstream components. This information can be applied to a Refinitiv Real-Time Distribution System cache or routed further upstream to the source of data. Once posted information is received, the upstream components can republish data to downstream consumers. See Section 6.9 for more information.

Table 10: OMM Message Types

## 6.2.4 Attribute Information

Attribute information provides additional message properties. It is supported by the [OMMAttribInfo](#) interface. The message properties in combination with the message model type uniquely identify a subscription stream, e.g. Item Name, Service Info. The Attribute information element has different data for different domains. For details, refer to the *RFA Java Edition RDM Usage Guide*.

### 6.2.4.1 Message Attribute Information Members

MEMBER	TYPE/INTERFACE	DESCRIPTION
flags	boolean has()	A combination of bit values to indicate the presence of optional <a href="#">attribute</a> members. See Table 12: Message Attribute Information Flags for more information about flag values.
serviceID	int getServiceID() setServiceID()	An identifier associated with a service. This value should correspond to the service content being requested from, or the service content being provided. A service is a logical mechanism that provides or enables access to a set of capabilities. In the scope of RFA, 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 things like Quality of Service, specific domain types available, and any dictionaries required to consume information from the service. This and much more information can be obtained via the Source Directory domain model. For details, refer to Section 6.2.4.4 and the <i>RFA Java Edition RDM Usage Guide</i> .
serviceName	String getServiceName() setServiceName()	A string representing the service name. <b>serviceName</b> maps to a <b>serviceID</b> . For details, refer to Section 6.2.4.4.
nameType	short getNameType() setNameType()	Indicates the type of the <a href="#">name</a> member. Examples are 'User Name' or 'RIC' (Refinitiv Instrument Code). This information is defined on a per-domain model basis and more information are be present as part of the specific domain model definition. Values associated with RDMs can be found in the <a href="#">RDMUser.java</a> file.
name	String getName() setName()	The name associated with the contents of the stream. The specific type and contents of the <a href="#">name</a> should comply with the rules associated with the <a href="#">nameType</a> member. Maximum length for name member should not be more than 255 characters.
filter	int getFilter() setFilter()	Combinations of <a href="#">filterId</a> bit values are used to describe content for domain model types that use an <a href="#">OMMFilterList</a> payload. When specified on a <a href="#">Request Message</a> , the <a href="#">filter</a> conveys information about the desired filter entries present in responses. When specified on any message housing an <a href="#">OMMFilterList</a> payload, the <a href="#">filter</a> conveys information about which filter entries are present. Filter identifier values are defined by the corresponding domain model specification. For more information see the <i>RFA Java RDM Usage Guide</i> .
identifier	int getId() setId()	A user-specified numeric identifier. This information is defined on a per-domain model basis and more information should be present as part of the specific domain model definition.
attribType	short getAttribType() setAttribType()	Identifies the content type of the <a href="#">Attrib</a> element of attribute information. Can indicate presence of an OMM container type (value 129 - 224) or some type of customer defined container type (225 - 255). For more details about container type definitions and use, refer to Chapter 1.

MEMBER	TYPE/INTERFACE	DESCRIPTION
Attrib	OMMData getAttrib()	Additional encoded message key attribute information. If populated, contents are described by the <code>attribType</code> member. Additional attribute information typically allows for further features in identifying of a message. Any additional attribute content is defined on a per-domain model basis and more information should be present as part of the specific domain model definition.

Table 11: Attribute Information Members

#### 6.2.4.2 Message Attribute Information Flags

INFORMATION FLAG	MEANING
HAS_SERVICE_ID	Indicates presence of the <code>serviceId</code> member.
HAS_SERVICE_NAME	Indicates presence of the <code>serviceName</code> member.
HAS_NAME	Indicates presence of the <code>name</code> member.
HAS_NAME_TYPE	Indicates presence of the <code>nameType</code> member.
HAS_FILTER	Indicates presence of the <code>filter</code> member.
HAS_ID	Indicates presence of the <code>identifier</code> member.
HAS_ATTRIB	Indicates presence of the <code>attribType</code> and <code>Attrib</code> members.

Table 12: Message Attribute Information Flags

#### 6.2.4.3 Message Attribute Information Utility Function

OMMAttribInfo interface provides the following utility functions:

FUNCTION NAME	DESCRIPTION
clear()	Clears a memory of the Attribute Information object. Used before reusing memory allocated by memory pool.
initFrom()	Initializes an instance of <code>OMMAttribInfo</code> from an existing instance. Performs a deep copy.

Table 13: Attribute Information Utility Functions

#### 6.2.4.4 Service Info

For opening an event stream associated with an item, the application needs to specify the service that it is interested in. Each service is defined by a **Service Info** which comprises of string `serviceName` and a numeric value `serviceID`. The collection of Service Infos that a provider offers is delivered by the source directory message at the start of a consumer application.

Applications use Service Info in combination with the item name to make item requests or post off-stream messages. The application can also use Service Info with ack messages, generic messages, item refresh messages, or for on-stream posting. An application may use either `serviceName` or `serviceID`, but not both.

The consumer application sets the Service Info as follows:

- In an item request message: Either Service name or Service ID is set. If using service name, the name must be advertised by the provider in the source directory message. If using service ID, the ID must map to a valid service name.
- In a generic message: A generic message is sent on the open stream, for which service info is already known. The Service ID in a generic message can be used by an application to set a custom numeric data. This number does not need to map to a service name.
- In a post message: The off-stream post message has either Service name or Service ID. If using service name, the name must be included in the source directory message. If using service ID, the ID must map to a valid service name. If the service is not offered by the provider, RFA will send cmd error event to the application.

The consumer application handles Service Info as follows:

- Received in a refresh response message: The received Service Info should match the one included in the request message. This validation is enforced by the RFA, so the application does not need to validate.

- Received in a generic message: The Service ID is custom data that may have some meaning to the receiving application and the handling is application defined.
- Received in an acknowledgement message: The Service ID is custom data that may have some meaning to the receiving application and the handling is application defined.

The provider application sets Service Info as follows:

- In a generic message: The Service ID in a generic message can be used by an application to set a custom numeric data. This number does not need to map to a service name.
- In an acknowledge message: The Service ID in an acknowledge message can be used by a provider application to set a custom numeric data. This number does not need to map to a service name

The provider application handles Service Info as follows:

- Received in an item request message: The application copies the received ServiceInfo into the refresh response message.
- Received in a generic message: The Service ID is custom data that has some meaning to the receiving application, and the handling is application defined.
- Received in a post message: The provider application ignores the data.

## 6.2.5 Indication Flags

Indication flags in a message indicate requests for special behaviors or the presence of optional elements. The application receiving the message must process it according to the indication flags. The indication flags are available in [OMMMsg.Indication](#).

The flags are set on a message using the [setIndicationFlags\(\)](#) method. The presence of an indication flag can be verified using the [isSet\(\)](#) method.

INDICATION FLAG / DEFINITION	DESCRIPTION
<b>Request Message Flags</b>	
ATTRIB_INFO_IN_UPDATES <a href="#">OMMMsg.Indication.ATTRIB_INFO_IN_UPDATES</a>	Used in streaming REQUEST type messages. Indicates the consumer preference to receive the full Attribute Information in update messages delivered on this stream. Presence of this flag does not guarantee that it will be present and absence of this flag does not guarantee that it will not be present in update messages. Whether this information is present is at the discretion of the provider application, and a consumer application should be capable of handling the different cases. When specified on a request to a ADS, the ADS ensures that this will be fulfilled.
BATCH_REQ <a href="#">OMMMsg.Indication.BATCH_REQ</a>	Used in streaming and non-streaming REQUEST type message. Indicates that the request message payload contains a list of items of interest, all with matching Attribute Information. See Section 13.2 for more information about Batch Request use.
CONFLATION_INFO_IN_UPDATES <a href="#">OMMMsg.Indication.CONFLATION_INFO_IN_UPDATES</a>	Used in streaming REQUEST type messages. Indicates the consumer's preference to receive conflation information in update messages delivered on this stream. Presence of this flag does not guarantee that conflation information will be present in update messages and absence of this flag does not guarantee that conflation information will not be present in update messages. Whether this information is present is at the discretion of the provider application, and a consumer application should be capable of handling conflation information in any update message. See 6.6 for details about conflation information on update messages.
NONSTREAMING <a href="#">OMMMsg.Indication.NONSTREAMING</a>	Used in REQUEST type messages. Indicates that the request is non-streaming. The absence of this indicates that the request is streaming. See Section 6.3 for more information.
PAUSE_REQ <a href="#">OMMMsg.Indication.PAUSE_REQ</a>	Used in streaming REQUEST type messages. Indicates the consumer's preference to pause the stream, although this does not guarantee that the stream will be paused. The absence of this indicates the consumer's preference to not-pause the stream (a.k.a. resume). See Section 13.4 for more information.

INDICATION FLAG / DEFINITION	DESCRIPTION
PRIVATE_STREAM <a href="#">OMMMsg.Indication.PRIVATE_STREAM</a>	Used in streaming and non-streaming REQUEST type message. Indicates that this stream is identified as a private stream. See Section 13.7 for more information about Private Stream.
REFRESH <a href="#">OMMMsg.Indication.REFRESH</a>	Used in REQUEST type messages. Indicates that a Refresh Response is requested.
VIEW <a href="#">OMMMsg.Indication.VIEW</a>	Used in streaming and non-streaming REQUEST type message. Indicates that the request message payload may contain a Dynamic View, indicating the specific information the application wishes to receive, or wishes to continue receiving a previously specified View. If this flag is not present, any previously specified view is discarded and the full View will be provided. See Section 13.3 for more information about Dynamic View use.
<b>Response Message Flags</b>	
CLEAR_CACHE <a href="#">OMMMsg.Indication.CLEAR_CACHE</a>	Used in STATUS_RESP and REFRESH_RESP type messages. Indicates that the data stored previously for this stream should be cleared. If the message has a payload, the data from the payload should be cached by the receiving application.
DO_NOT_CACHE <a href="#">OMMMsg.Indication.DO_NOT_CACHE</a>	Used in STATUS_RESP and REFRESH_RESP type messages. Indicates that the data from the message should not be cached. This can be used for cases when the message is transient or when the message does not have standard caching semantics (e.g. news headline broadcast stream).
DO_NOT_CONFLATE <a href="#">OMMMsg.Indication.DO_NOT_CONFLATE</a>	Used in UPDATE_RESP type messages. Indicates that the data from the message should not be conflated. This can be used for cases when the conflation semantics does not apply to the received data (e.g. news headline broadcast stream).
DO_NOT_RIPPLE <a href="#">OMMMsg.Indication.DO_NOT_RIPPLE</a>	Used in UPDATE_RESP type messages. Indicates that the contents of this message should have no rippling applied. Rippling is typically associated with an <a href="#">OMMFieldList</a> . Refer to Section 10.4.4 for additional information.
PRIVATE_STREAM <a href="#">OMMMsg.Indication.PRIVATE_STREAM</a>	Used in REFRESH_RESP and STATUS_RESP type messages. Indicates that this stream is identified as a private stream. See Section 13.7 for more information about Private Stream.
REFRESH_COMPLETE <a href="#">OMMMsg.Indication.REFRESH_COMPLETE</a>	Used in REFRESH_RESP type messages. When set, flag indicates that this is the final part of the Refresh message. For a single-part refresh (also referred to as an atomic refresh), this flag value should be set. On a multi-part refresh, this flag value should be set only on the final part. See Section 6.5 for more information about multi-part message handling.
<b>Generic Message Flags</b>	
GENERIC_COMPLETE <a href="#">OMMMsg.Indication.GENERIC_COMPLETE</a>	Used in GENERIC type messages. When set, the flag indicates that this is the final part of the generic message. For a single-part generic message (also referred to as an atomic generic message), this flag value should be set. On a multi-part generic message, this flag value should be set only on the final part. See Section 6.8 for more information about multi-part message handling.
<b>Post Message Flags</b>	
NEED_ACK <a href="#">OMMMsg.Indication.NEED_ACK</a>	If present, the consumer wants the provider to send an acknowledge message to indicate that the post message has been processed properly.

INDICATION FLAG / DEFINITION	DESCRIPTION
POST_COMPLETE <a href="#">OMMMsg.Indication.POST_COMPLETE</a>	Indicates the final part of the posting message. For a single-part post message (also referred to as an atomic post message), this flag value should be set. On a multi-part post message, this flag value should be set only on the final part. See Section 6.9 for more information about multi-part message handling.
POST_INIT <a href="#">OMMMsg.Indication.POST_INIT</a>	Indicates the first part of the posting message. For a single-part post message (also referred to as an atomic post message), this flag value should be set. On a multi-part post message, this flag value should be set only on the first part. See Section 6.9 for more information about multi-part message handling.

Table 14: OMM Message Indication Flags

## 6.2.6 Hint Flags

In general, an [OMMMsg](#) object has the capability to contain various kinds of properties, e.g. message type, message model type, attribute information, Quality of Service, item group, priority, etc. Certain properties such as message model type and message type are mandatory; while the rest, such as attribute information, priority, etc., are optional.

Hint flags are used to indicate the availability of the optional properties in a message and begin with the prefix **HAS\_**; e.g. **HAS\_ATTRIB\_INFO** indicates the availability of the attribute information in a message. The presence of a hint flag can be verified using the [has\(\)](#) method which returns a Boolean value. This verification is required prior to accessing the property; e.g. an application wishing to obtain attribute information from a message must first check for the presence of the **HAS\_ATTRIB\_INFO** flag using the [has\(\)](#) method. If available, the attribute information is obtained using [getAttribInfo\(\)](#) method.

The following table presents the hint flags supported by OMMMsg interface.

HINT FLAG	DESCRIPTION
HAS_ATTRIB_INFO	Indicates that the message has an Attribute Information element
HAS_CONFLATION_INFO	Indicates that the message has conflation information.
HAS_HEADER	Indicates that the message has extended header data.
HAS_ID	Indicates that the message has ID data.
HAS_ITEM_GROUP	Indicates that the message has an item group data.
HAS_PERMISSION_DATA	Indicates that the message has permission data.
HAS_PRIORITY	Indicates that the message has an Item priority data.
HAS_PUBLISHER_INFO	Indicates that the message has publisher information, such as publisher address and ID.
HAS_QOS	Indicates that the message has quality of service data.
HAS_QOS_REQ	Indicates that the message has specified the quality of service it requires from a provider.
HAS_RESP_TYPE_NUM	Indicates that the message has response type data.
HAS_SECONDARY_SEQ_NUM	Indicates that the message has secondary sequence number data.
HAS_SEQ_NUM	Indicates that the message has sequence number data.
HAS_STATE	Indicates that the message has state data.
HAS_USER_RIGHTS	Indicates that the message has user rights mask data.

Table 15: OMM Message Hint Flags

## 6.2.7 User Rights

`OMM.UserRights` data defines the User Rights values for the OMM Post Message. The value represents the combined rights for the user.

The following table presents the User Rights flags supported by OMMMsg interface.

USER RIGHTS FLAG	DESCRIPTION
CREATE_FLAG	User is allowed to create items in the cache of record.
DELETE_FLAG	User is allowed to remove items from the cache of record.
MODIFY_PERMISSION_DATA_FLAG	User is allowed to modify the permData associated with items already in the cache of record.

**Table 16: OMM User Rights Flags**

## 6.2.8 Item Group

An Item Group represents a group of items within a service which may change state together. It can be set on the message using the `setItemGroup()` interface which accepts an integer or an `OMMItemGroup`. For example, `setItemGroup(2)` sets 2 as the groupID directly on the message.

When creating `OMMItemGroups`:

- You can create an `OMMItemGroup` from a groupID, which in turn is set on the OMMMsg.

```
OMMItemGroup itemGroupID = OMMItemGroup.create(2); // an itemGroup object is created with
// a value of 2
ommmmsg.setItemGroup( itemGroupID ) // The itemgroup object is set on the OMMMsg
```

- You can create an `OMMItemGroup` can be created from a groupID and an existing Item Group. For example, an existing item group can be retrieved from a refresh message as in the case of a hybrid app.

```
// create itemGroup object from an existing itemgroup retrieved from an OMMMsg and a groupID
OMMItemGroup itemGroupID = OMMItemGroup.create( inMsg.getItemGroup(), 2);
ommmmsg.setItemGroup( itemGroupID ) // The itemgroup object is set on the OMMMsg
```

## 6.3 OMM Message Type: Request

A REQUEST message type is used by an OMM Consumer to express interest in a new stream, express non-ongoing interest in an item, or modify parameters on an existing stream. Several indication flags are used in conjunction with the REQUEST message type to indicate a specific interaction.

INTERACTION	INDICATION FLAGS	DESCRIPTION
Streaming Request	OMMMMsg.Indication.REFRESH	Express interest in a new stream. Typically results in a Refresh Response, followed by Update Response message, or a Status Response being delivered.
Nonstreaming Request	OMMMMsg.Indication.NONSTREAMING OMMMMsg.Indication. REFRESH	Express non-ongoing interest in an item, sometimes referred to as a "snapshot". Typically results in a Refresh Response, or in a Status Response being delivered.
Modify Existing Stream (a.k.a. reissue)	OMMMMsg.Indication.REFRESH (optional)	Modify some parameters on an existing stream. Optionally a Refresh Response may be requested with OMMMsg.Indication.REFRESH, which results in a Refresh Response.

**Table 17: Request Message Type and Indication Flags Interaction**

**NOTE:** Additional Request message types were deprecated in rfaj7.2.1 and replaced with REQUEST message type. To use the REQUEST message type, nothing extra is needed. However to instruct RFA to decode request messages to the REQUEST message type, RSSL\_PROV connections will need to set the key `useDeprecatedRequestMsgs` to the value of `false`, in the configuration. Refer to the RSSL\_PROV section of *RFA - Java Edition - Configuration and Logging Guide*.

### 6.3.1 Streaming Request

A REQUEST message type with REFRESH indication (a.k.a. Streaming Request) is used by an OMM Consumer to express interest in a particular information stream. The request's Attribute Information members help identify the stream, and priority information can be used to indicate the streams importance to the consumer. Quality of Service information can be used to express either a specific desired Quality of Service or a range of acceptable qualities of service that can satisfy the request (See Section 7.8.3).

When a streaming request message is issued for a particular interest, this is considered as opening an **event stream**. If requested information is available and the consumer is entitled to receive the information, this typically results in an initial refresh message being delivered to the consumer (though a status message is also possible: either message can be used to indicate an event stream is open). If information is not available or the user is not entitled, a status message is typically delivered to provide more detailed information to the consumer.

Issuing a streaming request on an existing event stream allows a consumer to modify some parameters associated with the stream (refer to Section 6.3.2: Changeable Event Stream Attributes). Also known as a **reissue**, this can be used to Pause or Resume a stream (see Section 13.4), change a Dynamic View (see Section 13.3), increase or decrease the streams priority, or request that a new refresh is delivered. The header of a Streaming Request message must include the element:

- Attribute Information and the **HAS\_ATTRIB\_INFO** hint flag

The header of a Streaming Request message can also include the following optional elements:

- Conflation Count, Conflation Time, and the **HAS\_CONFLATION\_INFO** hint flag
- Priority and the **HAS\_PRIORITY** hint flag
- Quality of Service Request and the **HAS\_QOS\_REQ** hint flag

### 6.3.2 Changeable Event Stream Attributes

A select number of attributes may change during the life of an event stream. A Consumer can change attributes via **reissue**, a subsequent streaming request message that uses the same event stream as previous requests. An Interactive or Non-Interactive Provider can change attributes via a subsequent solicited or unsolicited refresh response message.

The message attribute information **filter** member, although not typical, may change between the Consumer request and Provider response. This would likely be due to a difference between the filter entries that the Consumer asks for and the filter entries that the Provider can provide. This behavior is allowed within some domains. More information should be present as part of the specific domain model definition.

Contents of the message attribute information's **Attributes** may change. This behavior is allowed within some domains. More information should be present as part of the specific domain model definition.

A Consumer may change the **priorityClass** or **priorityCount** via reissue messages to indicate more or less interest associated with the event stream. For more information, refer to Section 10.9.2.

If a Quality of Service range is requested, the provided refresh message will only include the concrete Quality of Service, which may be different from the best and worst specified. If a **dynamic QualityofService** is supported, the provided quality of service may occasionally change over the life of the event stream, however, this should stay within the range that was requested on the streaming request message.

A consumer can pause an event stream via reissue message that includes **PAUSE\_REQ** indication flag. A consumer can then resume the event stream via a reissue message that **does not include PAUSE\_REQ**. If the REFRESH flag is set, the consumer will receive a refresh.

A consumer can request limited information on specific domains by including the **VIEW** indication flag and including view data in the payload of the streaming request message. A consumer may change interest in view data via reissue messages that include the **VIEW** indication flag and new view data in the payload. If a reissue is being done for a different reason (pausing a stream, priority change, etc) and the user wants to keep the current view, the **VIEW** indication flag should be set with no view data in the payload. To remove a view, the user can perform a reissue with no **VIEW** indication flag set.

### 6.3.3 Nonstreaming Request

A REQUEST message type with NONSTREAMING and REFRESH indication (a.k.a. nonstreaming request), called also Snapshot Request, is used by an OMM Consumer to express one time interest in a particular information. A nonstreaming request message, issued for a particular interest, opens an event stream. If requested information is available and the consumer is entitled to receive the information, this will typically result in a complete one time refresh message, called an Image, being delivered to the consumer. If information is not available or the user is not entitled, a status message is typically delivered to provide more detailed information to the consumer. After either of the responses is delivered to the application, the stream is considered closed throughout the system.

The complete image can be delivered by multiple refresh messages (this may occur if the response contains a lot of content). The refresh message containing the last part of the image has the **REFRESH\_COMPLETE** indication flag set.

- The header of a Nonstreaming Request message **must** include attribute Information and the **HAS\_ATTRIB\_INFO** hint flag
- The header of a Nonstreaming Request message can **optionally** include a Quality of Service Request and the **HAS\_QOS\_REQ** hint flag

### 6.4 OMM Message Type: Close Request

A CLOSE\_REQ message type is used by an OMM Consumer to indicate no further interest in an event stream. The stream should be closed throughout the system as a result. This way of closing stream is not a typical case. It is recommended that the OMM Consumer use the `unregisterClient()` method, to close a stream.

### 6.5 OMM Message Type: Refresh Response

Provided as an initial response or when an upstream source requires a data resynchronization point, a REFRESH\_RESP message type contains payload information along with state, Quality of Service, permissioning, and group information. If provided as a response to a request message, this is referred to as a **solicited refresh**. If some kind of information change occurs, possibly an error detected on a stream, an upstream provider can generate a refresh response message to downstream consumers - this is referred to as an **unsolicited refresh**. Typically, solicited refresh messages are delivered only to the requesting consumer application while unsolicited refresh messages are delivered to all consumers with the relevant stream open.

Response messages support the capability of refresh fragmentation. Refresh fragmentation is the ability for an image to be split across multiple independently distributed messages. Each response message is known as a **refresh**. The collective set of refreshes forming an image is known as a multi-part refresh. The final refresh of multi-part or single-part message can be indicated using a flag known as the **refresh complete** flag. The final refresh is specified by setting the indication flag to `OMMMsg.Indication.REFRESH_COMPLETE`.

The header of a Refresh Response message must include the following elements:

- Attribute Information and the **HAS\_ATTRIB\_INFO** hint flag (required on first refresh, but optional on subsequent refreshes)
- Item Group data and the **HAS\_ITEM\_GROUP** hint flag
- Response Type and the **HAS\_RESP\_TYPE\_NUM** hint flag
- State and the **HAS\_STATE** hint flag

The header of a Refresh Response message can also include the following optional elements:

- Publisher Info and the **HAS\_PUBLISHER\_INFO** hint flag
- Permission data and the **HAS\_PERMISSION\_DATA** hint flag
- Quality of Service and the **HAS\_QOS** hint flag
- Sequence Number and the **HAS\_SEQ\_NUM** hint flag

### 6.6 OMM Message Type: Update Response

An UPDATE\_RESP message type is used by OMM Interactive and OMM Non-Interactive Providers to convey changes to data associated with an event stream. When streaming, update messages typically flow after an initial refresh has been delivered. It is possible for update messages to be delivered between parts of a multi-part refresh message.

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 Quality of Service is requested (see Section 10.3.5), a stream is paused (see Section 13.4), or if a consuming application is unable to keep up with the data rates associated with the stream. If conflation is occurring, specific information can be provided with the update response message via the optional conflation information.

The header of an Update Response message must include the following element:

- Response Type and the **HAS\_RESP\_TYPE\_NUM** hint flag

The header of an Update Response message can also include the following optional elements:

- Attribute Information and the **HAS\_ATTRIB\_INFO** hint flag
- Conflation Count, Conflation Time data and the **HAS\_CONFLATION\_INFO** hint flag
- Publisher Info and the **HAS\_PUBLISHER\_INFO** hint flag
- Permission data and the **HAS\_PERMISSION\_DATA** hint flag

## 6.7 OMM Message Type: Status Response

A STATUS\_RESP message type is used to indicate changes to the event stream or data properties. This message can convey changes in [streamState](#) or [dataState](#) (see Section 10.3.6), changes in a stream's permissioning information (see Section 10.5), and changes to the item group that the event stream is part of (see Section 10.9.1). A Provider application uses the status response message to close streams to a Consumer, both in conjunction with an initial request or at some point after the stream has been established. A status response message can also be used to indicate successful establishment of a stream, even though the message may not contain any data - this can be useful when establishing a stream solely to exchange bi-directional generic messages.

The header of a Status Response message can include the following optional elements:

- Attribute Information and the **HAS\_ATTRIB\_INFO** hint flag
- Item Group data and the **HAS\_ITEM\_GROUP** hint flag
- Publisher Info and the **HAS\_PUBLISHER\_INFO** hint flag
- Permission data and the **HAS\_PERMISSION\_DATA** hint flag
- Sequence Number and the **HAS\_SEQ\_NUM** hint flag
- State and the **HAS\_STATE** hint flag

## 6.8 OMM Message Type: Generic

A GENERIC message type is a bi-directional message that does not have any implicit interaction semantics associated with it, thus the name generic. Once an event stream is established via a request-refresh/status interaction, a generic message can be sent from consumer to provider as well as from provider to consumer, and can also be leveraged by non-interactive provider applications. Generic messages are transient and are typically not cached by any Refinitiv Real-Time Distribution System components. The attribute information element of a generic message does not need to match the attribute information associated with the stream the generic message is flowing on. This allows for the attribute information to be used independently of the stream. Any specific message usage, attribute information usage, expected interactions, and handling instructions are typically defined by a domain message model specification.

Generic messages support fragmentation. Generic message fragmentation is the ability for an arbitrary set of data to be split across multiple independently distributed generic messages. Depending on the usage of the generic message defined by the OMM type for this event stream, the generic msg (or contained payload) may be atomic or multi-part. An atomic generic message is one in which a single generic message and contained payload are not dependent on any subsequent generic message to complete the transfer of a set of data (payload). A collective set of generic messages completing a well defined data set or behavior (as defined by the relevant OMM type) is known as a multi-part generic. The final of multi-part or single-part message can be indicated using a flag known as the **generic complete** flag. The final generic message is specified by setting the **OMMMsg.Indication.GENERIC\_COMPLETE** indication flag.

The header of a Generic message can include the following optional elements:

- Attribute Information and the **HAS\_ATTRIB\_INFO** hint flag
- Secondary Sequence Number data and the **HAS\_SECONDARY\_SEQ\_NUM** hint flag

## 6.9 OMM Message Type: Post

The POST message type allows a Consumer application to push content to upstream components. This information can be applied to a Refinitiv Real-Time Distribution System cache or routed further upstream to the source of data. Once received, the upstream components can republish data to downstream consumers.

RFA supports posting single-part and multi-part messages. A post message can be identified by an ID (known as a **Post ID**). For purposes of identification, multi-part post messages require ID and a sequence number. The last message of a multi-part post message must contain **OMMMsg.Indication.POST\_COMPLETE** indication flag. The first post message contains **OMMMsg.Indication.POST\_INIT** indication flag, either in single-post, or multi-post message.

The consumer application can indicate whether it requires an acknowledge response to the posted message by setting the **OMMMsg.Indication.NEED\_ACK** indication flag. If an Acknowledge response is desired, the message must contain a Post ID. In response, the provider application must send either a positive acknowledge response or negative acknowledge response to the consumer application on the event stream over which the post message was received. If the provider application does not respond within a specified time period, RFA sends a negative acknowledge response to the consumer application. The acknowledge response message must contain the Post ID and the sequence number, if it is a response to a part of a multi-part post message.

The header of Post message can include the following optional elements:

- Attribute Information and the **HAS\_ATTRIB\_INFO** hint flag
- ID and the **HAS\_ID** hint flag
- Permission Data and **HAS\_PERMISSION\_DATA** hint flag
- Sequence Number and the **HAS\_SEQ\_NUM** hint flag
- User Rights and the **HAS\_USER\_RIGHTS** hint flag
- Publisher Data (retrieved by casting to the `PublisherPrincipal` Identity) and the **HAS\_PUBLISHER\_INFO** hint flag

### 6.9.1 Publisher Data

The consumer can include publisher data in the `OMMMsg` by using the following methods in combination:

- Using the `setPrincipalIdentity(PublisherPrincipalIdentity)` method to set the **HAS\_PUBLISHER\_INFO** hint flag.
- Using the `setPublisherAddress(long)` method to set the publisher address on the `PublisherPrincipalIdentity`.
- Using the `setPublisherId(long)` method to set the publisher ID on the `PublisherPrincipalIdentity`.

---

**NOTE:** If the **HAS\_PUBLISHER\_INFO** hint flag is not set on the post message, RFA will insert its own publisher data into the `OMMMsg` before passing the message to the provider application.

---

### 6.9.2 On-stream and Off-stream Posting

Post messages can be routed along a specific event stream associated with an item, referred to as on-stream posting, or along a user's Login event stream, referred to as off-stream posting. A post message can contain any RFA container type, including other messages. User identification information can be associated with a post message and can be provided along with the content that was posted. See Section 13.6 for more details.

## 6.10 OMM Message Type: Acknowledge Response

An `ACK_RESP` message type can be sent from a provider to a consumer to indicate receipt of a specific message. The acknowledgment carries success or failure (negative acknowledgment or nak) information to the consumer. Currently, a consumer can request acknowledgment for post messages and close request messages.

The header of an Acknowledge response message must include the following element:

- State and **HAS\_STATE** hint flag

The header of an Acknowledge response message can also include the following optional elements:

- Attribute Information and **HAS\_ATTRIB\_INFO** hint flag
- ID and **HAS\_ID** hint flag
- Sequence Number and **HAS\_SEQ\_NUM** hint flag

## 6.11 OMM Encoder

OMMEncoder manages data and encoding state for encoding OMMMsg object or OMMData objects into a buffer. The OMMData is encoded into a specific version of RWF format. The version is an attribute of the encoded object.

### 6.11.1 Encoding Sequence

The encoding process has to follow a sequence:

1. The first step is initialization. The `initialize()` method takes two parameters: a type of the encoded object, and estimated size of the object when encoded. The size can be greater than the actual size, but can't be smaller.

An overloaded `initialize()` method takes an additional parameter, `DataDefDictionary`. This method is used for encoding `OMMElementList` and `OMMFieldList` data types that have defined data.

2. The next step is initialization of the top container that contains the data. For example `encodeArrayInit()` sets the encoder to encode an `OMMArray`. Refer to section for encoder data initialization.
3. The next step is encoding the data that was initialized in the previous step. This can take multiple initialize- encode, since the top container can be composed of nested dataformats. Refer to for encoding data formats.
4. If the encoded container is a dataformat that implements `OMMIterable`, the encoder needs to mark the end of the dataform. A method `encodeAggregateComplete()` completes encoding a dataform.
5. The encoding is done with the `acquireEncodedObject()` call. This method returns encoded object as an `OMMData` type and clears the encoder. The encoder can be reused or released back to the memory pool.

An alternative method `getEncodedObject()` also returns encoded object as an `OMMData` type but does not clear the encoder.

Another way to retrieve encoded object is through `getBytes()` method. The method returns a copy of current data in the encoder, as a byte array. Another overloaded method takes a byte array and an integer as parameters and fills the byte array with the encoded data at the position indicated by the second parameter. This method does not clear the encoder.

### 6.11.2 Encoder Utility Methods

The OMMEncoder supports other utility methods, summarized in the table below.

METHOD	DESCRIPTION
<code>getEncodedSize()</code>	Returns the size of the currently encoded data in the buffer.
<code>getEncoderState()</code>	Returns a value of <code>OMMTypes</code> for the next expected state to be called on an encoder.
<code>isComplete()</code>	Returns true if the encoder is in a complete state.
<code>printState()</code>	Prints the encoder state to the <code>PrintWriter</code> parameter. If the second parameter is true, it print also a hex representation of the current data in the buffer. It is safe to invoke this method at any time (e.g. while data is still being added to encode).
<code>useSize()</code>	This method takes an integer parameter, which identifies <code>LARGE_SIZE</code> or <code>SMALL_SIZE</code> . By default the encoder is set to the <code>LARGE_SIZE</code> . With this method it is possible to optimize the size of the buffer used by the encoder. The <code>SMALL_SIZE</code> can be used when the encoded data size is less than 128 bytes.

**Table 18: OMMEncoder Utility Methods**

## 6.12 Versioning Support

RWF is a family of highly optimized binary message and data formats used to represent any data format or type supported by OMM. RWF includes major and minor versions to support future enhancements and extensions. RFA `OMMData` and `OMMMsg` interfaces provide versioning support to let clients encode data with the negotiated RWF version.

In systems with components supporting various versions, the RFA application must follow these guidelines to optimize usage and performance:

- When a consumer establishes a connection with a provider with a different RWF version, the lower version is considered the negotiated version for both the consumer and provider on the established connection.
- RFA performs version checking. If the version of encoded data in a message does not match the negotiated version, the RFA adapter might re-encode the message with the negotiated version before sending it out on the wire. This creates latency which

can negatively affect performance, especially if the client application deals with large volumes of data. It is recommended that clients pass version information to OMMMsg object when encoding data in a client application to prevent the RFA from decoding/re-encoding the data.

- To optimize RFA performance, a client application should call the `setAssociatedMetaInfo()` method to specify the supported version.
- The handle passed as a parameter to the `setAssociatedMetaInfo()` function on an OMMMsg object can be:
  - A login handle for a non-interactive client provider application
  - A client session handle for OMM provider and Generic Message provider applications.
  - An open item stream handle for generic message consumer.
- To acquire OMMData and/or OMMMsg from pre-encoded data, it is recommended to pass the negotiated version information to the `acquireDataFor()` and `acquireMsgFor()` methods along with pre-encoded data. The client application can invoke `getMajorVersion()` or `getMinorVersion()` on OMMMsg or OMMData to retrieve the relevant version information, cache it, and later pass it to `acquireDataFor()` and `acquireMsgFor()` functions.
- To pass version information to data objects such as Map, Vector, Series, FieldList, ElementList, FilterList, or Array; the user should call `setAssociatedMetaInfo()` on data objects and pass version info. Version information can be retrieved from `getMajorVersion()/getMinorVersion()` method of the OMMMsg object.
- The session's configuration parameter `OMMAllowMultiVersion` sets whether the client application supports multiple versions between physical connections in one session.
- If `OMMAllowMultiVersion` is set to `AllowMultiVer`, the client application supports all connections, regardless of whether the versions are same or different. By default, `OMMAllowMultiVersion` is set to `AllowMultiVer`.
- If `OMMAllowMultiVersion` is set to `NotAllowMultiVer` then the client supports only the negotiated version from the first established connection (called the "first come, first served rule"). Other, different versions will be disconnected. In this case, a client application might not receive data because the first established connection might not provide proper services or capabilities suitable to the client's needs. If a hybrid application has different versions between provider-side connection and consumer-side connection, the hybrid application might not receive or publish data due to a disconnection caused by the "first come, first served" rule. To solve such issues, the client either can change the RFA and infrastructure components to use same RWF version, or change the value of `OMMAllowMultiVersion` to "`AllowMultiVer`".
- For other OMM connection types (except CPROV connections), a client application always encodes its data with the connection's negotiated version according to results of the `setAssociatedMetaInfo()` method invocation using handles.
- Passing version information occurs mainly on data-encoding side of client application. Some applications have versioning support implementations in their package release.

The following samples illustrate how to pass version information in applications in scenarios where multiple versions exist.

Passing version information on a generic message by a provider application:

```
OMMMsg outmsg = _pool.acquireMsg();
outmsg.setMsgType(OMMMsg.MsgType.GENERIC);
outmsg.setMsgModelType(RDMMMsgTypes.MARKET_BY_ORDER);

// set negotiated version info in OMMMsg using handle
Handle handle = event.getHandle();
outmsg.setAssociatedMetaInfo(handle);
```

Passing version information using a login handle containing negotiated version information by a consumer application:

```
OMMMsg ommmsg = pool.acquireMsg()
ommmsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommmsg.setMsgModelType(RDMMMsgTypes.MARKET_BY_ORDER);
ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);

// Setting OMMMsg with negotiated version info from login handle
ommmsg.setAssociatedMetaInfo(_loginHandle);

// Passing version information to pre-encoded buffer to OMMMsg:
OMMMsg msg = (OMMMsg) encoder.getEncodedObject();
Byte [] encbytes = msg.getBytes();
OMMMsg newMsg = pool.acquireMsgFor(encbytes, msg.getMajorVersion(), msg.getMinorVersion());
```

In the above example, *enbytes* contains the encoded buffer. To create *OMMMsg* from *enbytes*, the application should pass version information for RFA to decode message properly.

## 6.13 Pooling

Pooling of OMM objects is achieved through the *OMMPool* object factory. The *OMMPool* is a factory for creating OMM objects and provides an efficient means for releasing and reusing OMM objects created frequently. The *OMMPool* provides copy methods to copy OMM objects. Objects created by *OMMPool* may be changeable by the application, or read only.

An application has several choices when using objects created from an *OMMPool*:

- It can remove all references to the object and let the Java GC recycle it.
- It can release the object back into the *OMMPool* with one of the *OMMPool*'s release methods. This is typically more efficient than the Java GC for objects that are created very often.

When the object is released, it must be released into the same pool that created it.

- It can call the *clear()* method on changeable objects and reuse the same object. This eliminates the need to release and acquire the object from the pool.

### 6.13.1 Managing the Pool

The *OMMPool* object is typically created by an application once, using a static *create()* method and stored as a member. This pool can be created as either a single-threaded or thread-safe *OMMPool* via the *create(int threading)* method. A single-threaded *OMMPool* is used only when all acquire and release methods are called from the same thread. A single-threaded pool is more efficient, as it does not implement any locking mechanism. A thread-safe pool is the default *OMMPool* type and should be used in all other scenarios. In general, if available, an existing pool can be reused. The *OMMPool* can be destroyed with the *destroy()* method. If there are objects that were created using this pool, and not returned, the objects are destroyed.

```
// creating a thread-safe pool
private static OMMPool safe_pool = OMMPool.create(OMMPool.THREAD_SAFE);

// creating a single-threaded pool
private static OMMPool pool = OMMPool.create(OMMPool.SINGLE_THREADED);

// destroying pool
pool.destroy();
```

#### Example 1: Creating and Destroying an OMMPool

### 6.13.2 Managed Objects

The *OMMPool* provides memory management to the following objects:

- *OMMAttribInfo*
- *OMMMsg*
- *OMMData*
- *OMMState*
- *OMMEncoder*

The following table summarizes methods that support memory management of these objects:

OBJECTS	SUPPORTING METHODS	
OMMAttribInfo	OMMPool.acquireAttribInfo()	Returns a writable instance of OMMAttribInfo, non-initialized.
	OMMPool.acquireCopy()	This method takes two parameters: an OMMAttribInfo instance, which the new object copies data from, and a boolean flag. It returns a read-only instance of OMMAttribInfo, when the flag is set, and writable instance, otherwise.
	OMMPool.releaseAttribInfo()	It recycles the OMMAttribInfo object to the pool. The data is cleared, i.e., set to uninitialized state.
OMMMsg	OMMPool.acquireMsg()	Returns a writable instance of OMMMsg, non-initialized.

OBJECTS	SUPPORTING METHODS	
	OMMPool.acquireMsgFor()	Returns a read-only instance of OMMMsg, non-initialized. There are four overloaded methods, described below: <ul style="list-style-type: none"> <li>• Takes a byte array parameter. The parameter is set as the OMMMsg bytes.</li> <li>• Takes byte array and two byte parameters. The first parameter is set as the OMMMsg bytes, and the two byte parameters are used as major and minor version.</li> <li>• Takes a byte array, integer, and two bytes parameters. The first parameter is set as the OMMMsg bytes, and the two byte parameters are used as major and minor version. The integer indicates the length of the buffer.</li> <li>• Takes a byte array and String parameter. The bytes are set as the OMMBytes, and the String is used as the Service Name.</li> </ul>
	OMMPool.acquireCopy()	This method takes two parameters: an OMMMsg instance, that the new object copies data from, and a boolean flag. It returns a read-only instance of an OMMMsg, when the flag is set, and writable instance, otherwise.
	OMMPool.releaseMsg()	It recycles the OMMMsg object to the pool. The data is cleared, i.e., set to the uninitialized state.
OMMData	OMMPool.acquireDataFor()	Returns a read-only instance of OMMData, non-initialized. There are two overloaded methods, described below: <ul style="list-style-type: none"> <li>• Takes byte array parameter and short. The parameter is set as the OMMData bytes, and the short as the data type.</li> <li>• Takes a byte array, a short, and two byte parameters. The bytes are set as the OMMData, the short as the data type, and the two byte parameters are used as the major and minor version.</li> </ul>
	OMMPool.acquireCopy()	This method takes two parameters: an OMMData instance, that the new object copies data from, and a boolean flag. It returns a read-only instance of OMMData, when the flag is set, and a writable instance, otherwise.
	OMMPool.releaseData()	It recycles the OMMData object to the pool. The data is cleared, i.e., set to an uninitialized state.
OMMState	OMMPool.acquireState()	Returns a writable instance of OMMState, non-initialized.
OMMState	OMMPool.releaseState()	It recycles the OMMState object to the pool. The data is cleared, i.e., set to uninitialized state.
OMMEncoder	OMMPool.acquireEncoder()	Returns a writable instance of OMMEncoder, non-initialized.
OMMEncoder	OMMPool.releaseEncoder()	It recycles the OMMEncoder object to the pool. The data is cleared, i.e., set to uninitialized state.

Table 19: OMMPool Methods

## 6.14 Creating the Message

The following steps outline the process of creating OMM Message.

- Allocate memory for the message from the pool.
- Set the header data. Part of the header may be encoded.
- Optionally, encode the payload.

### 6.14.1 Allocate Memory

The memory for an OMMMsg instance is allocated from the OMMPool. Refer to Section 6.13.1 for description of memory pooling.

The memory is allocated using the following methods: `acquireMsg()`, `acquireMsgFor()`, or `acquireCopy()`. Refer to Table 19 for a description of these methods.

### 6.14.2 Set Header

All messages have mandatory data, such as the message type and message model type. The optional message elements are set depending on the application. Refer to Table 8 for all message elements and the accessor methods, which are used to set the message's elements.

Two optional elements: OMMState and OMMAttribInfo are memory managed objects. There are two different ways to set the elements on the message.

- These objects can be created using the an OMMPool, as shown in the Table 19. Once the objects are initialized with the data and copied into the OMMMsg object using `OMMMsg.setAttribInfo()` or `OMMMsg.setState()` methods, the memory should be returned to the pool with `OMMPool.releaseAttribInfo()` or `OMMMsg.releaseState()` methods.

The example code below demonstrates allocating the memory for the objects and setting the elements on a message.

```
// assume the pool and message have been created

// allocate memory for OMMAttribInfo
OMMAttribInfo ai = pool.acquireAttribInfo();
// set attribute information data
ai.setName("TRI.N");
ai.setServiceName("DIRECT_FEED");
// set the attribute information on the message
message.setAttribInfo(ai);
// release the memory allocated for ai object
pool.releaseAttribInfo(ai);

// allocate memory for OMMState
OMMState state = pool.acquireState();
// set state data
state.setStreamState(OMMState.Stream.OPEN);
state.setDataState(OMMState.Data.OK);
// set the state on the message
message.setState(state);
// release the memory allocated for state object
pool.releaseState(state);
```

#### Example 2: Setting Attribute Information and State Using Pool

- These objects can be set directly on the message. The values are passed to the set methods. The example code below illustrates the alternate way of setting the message elements:

```
// assume the message has been created

// attribute information data
String serviceName = "DIRECT_FEED");
String itemName = "TRI.N";
short nameType = RDMInstrument.NameType.RIC;
// set the attribute information on the message
message.setAttribInfo(serviceName, itemName, nameType);

// state data
byte streamState = OMMState.Stream.OPEN;
byte dataState = OMMState.Data.OK;
short code = OMMState.Code.NONE;
// set the state data on the message
message.setState(streamState, dataState, code, "");
```

### Example 3: Setting Attribute Information and State Using Set Methods

There are two message elements that are encoded in the message: payload and some types of `Attrib` element of Attribute Information. Refer to Table 8 and Table 11 for list of message elements and attribute information elements respectively. If the message does not have payload and the attribute information element does not contain an `attrib` element that needs to be encoded, the message can be created using the set methods. Otherwise, an encoder must be utilized.

`OMMEncoder` is memory managed object. Refer to Table 19 for the description and methods how to create the encoder.

The steps to create a message using an encoder are as follows:

- Set all elements on the message using set methods. The exception may be an `attrib` element in attribute information, if it needs to be encoded. Refer to Section 6.2.1 for the accessor methods.
- Acquire an encoder.
- Initialize the encoder with the object type and size. The object type is `OMMTypes.MSG`, and the size is the estimated size of the encoded message. If the allocated size is smaller than the size of encoded message, the RFA will throw an exception.
- Initialize the encoder with the populated message, the type of attribute information, and the type of payload. The type `OMMTypes.NO_DATA` means that this element is not encoded. For example, if the type of the attribute information is element list, and the message has no payload, the initialization is as shown below:

```
Encoder.encodeMsgInit(message, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA)
```

This method copies the message data to the encoder space. Once the message has been copied to the encoder, it is not used anymore. The memory should be returned to the pool by calling `OMMPool.releaseMsg()` method.

- Encode the `attrib` element if needed.
- Encode the payload if needed.
- Retrieve the message from encoder with using either `getEncodedObject()` or `acquireEncodedObject()` into another `OMMMsg` object. These methods return the encoded object. The first method does not clear the encoder, where the second does. This object is not allocated from the pool. In this step the encoded content from the encoder is copied to the new message. This message is then ready to be sent out.
- Clean up the encoder. If the encoder is intended for reuse, it needs to be cleaned up by calling the `OMMEncoder.initialize()` method before the next usage. Otherwise the encoder should be returned to the pool by invoking the `OMMPool.releaseEncoder()` method.

### 6.14.3 Encoding Payloads

If a payload is present, the encoder must be initialized with the type of the payload. After the initialization, the data is encoded according to the data type. For OMMData encoding, refer to Chapter 1.

### 6.14.4 Message Encoding Examples

Depending on the message elements, the application will create the message according to a routine presented in one of the examples below.

#### 6.14.4.1 Using Set Methods

The following routine can be used if all of the following are true:

- the message does not contain any payload
- if it contains attribute information, the attribute information does not contain an attrib element that has to be encoded

```
// allocate a message from the pool
OMMMsg msg = pool.acquireMsg();

// set the mandatory elements
msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setMsgModelType(RDMMsgTypes.MARKET_PRICE);
ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);

// set other elements
msg.setAttribInfo("DIRECT_FEED", "TRI.N", RDMInstrument.NameType.RIC);
msg.setPriority((byte)1, 1); // set priority information

// after all elements are set, the message can be sent out
```

#### Example 4: Creating Message Using Set Methods

#### 6.14.4.2 Encoding Attributes

The following routine can be used if the message does not contain any payload, but contains attribute information with an attrib element, which has to be encoded.

```
// allocate a message from the pool; assume the pool has been created
OMMMsg msg = pool.acquireMsg();

// set the message elements
msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setMsgModelType(RDMMsgTypes.LOGIN);
ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);

// set some elements in attribute information
msg.setAttribInfo(null, "userName", RDMInstrument.NameType.USER_NAME);

// get the encoder and initialize
OMMEncoder encoder = pool.acquireEncoder();
encoder.initialize(OMMTypes.MSG, 500);
// The attrib part of attribute information is an element list, and the message has no payload
encoder.encodeMsgInit(msg, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA);

// Release the initial message
pool.releaseMsg(msg);

// encode the element list containing three standard elements
encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
encoder.encodeElementEntryInit(RDMUser.Attrib.ApplicationId, OMMTypes.ASCII_STRING);
```

```

encoder.encodeString("256", OMMTypes.ASCII_STRING);
encoder.encodeElementEntryInit(RDMUser.Attrib.Position, OMMTypes.ASCII_STRING);
encoder.encodeString("1.1.1.1/net", OMMTypes.ASCII_STRING);
encoder.encodeElementEntryInit(RDMUser.Attrib.Role, OMMTypes.UINT);
encoder.encodeUInt((long)RDMUser.Role.CONSUMER);
encoder.encodeAggregateComplete();

// get the encoded message from the encoder
OMMMsg encodedMessage = (OMMMsg)encoder.getEncodedObject();

// the encoder may be released
pool.releaseEncoder(encoder);

```

### Example 5: Creating Message With Encoded Attributes

#### 6.14.4.3 Encoding Payload

The following routine can be used if the message contains a payload, but does not need to an encode attrib element:

```

// allocate a message from the pool; assume the pool has been created
OMMMsg msg = pool.acquireMsg();

// set the message elements
msg.setMsgType(OMMMsg.MsgType.REFRESH_RESP);
msg.setMsgModelType(RDMMsgTypes.DIRECTORY);

msg.setState(OMMState.Stream.OPEN, OMMState.Data.OK, OMMState.Code.NONE, "");
msg.setItemGroup(1);
// set other elements

// get the encoder and initialize
OMMEncoder encoder = pool.acquireEncoder();
encoder.initialize(OMMTypes.MSG, 1000);

// The attrib element of Attribute Information is not encoded, thus NO_DATA type
// The message has payload of the Map type
encoder.encodeMsgInit(msg, OMMTypes.NO_DATA, OMMTypes.MAP);

// Release the initial message
pool.releaseMsg(msg);

// encode the payload
encoder.encodeMapInit(flags, keyType, dataType, countHint, keyFieldId); // set the init data appropriately
// encode the map entries (not shown)
encoder.encodeAggregateComplete();

// get the encoded message from the encoder
OMMMsg encodedMessage = (OMMMsg)encoder.getEncodedObject();

// the encoder may be released
pool.releaseEncoder(encoder);

```

### Example 6: Creating Message With an Encoded Payload

### 6.14.4.4 Encoding Attributes and Payload

The following routine can be used if the message contains a payload and attribute information with an attrib element, which has to be encoded:

```
// allocate a message from the pool; assume the pool has been created
OMMMsg msg = pool.acquireMsg();

// set the message elements
msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setModelType(RDMMsgTypes.LOGIN);
ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);

// set some elements in attribute information
msg.setAttribInfo(null, "userName", RDMInstrument.NameType.USER_NAME);

// get the encoder and initialize
OMMEncoder encoder = pool.acquireEncoder();
encoder.initialize(OMMTypes.MSG, 500);
// The attrib is an element list, and the message has payload type of Vector
encoder.encodeMsgInit(msg, OMMTypes.ELEMENT_LIST, OMMTypes.VECTOR);

// Release the initial message
pool.releaseMsg(msg);

// encode the attrib element as element list
encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
// encode the element list entries (not shown)
encoder.encodeAggregateComplete();

// encode the payload as vector
encoder.encodeVectorInit(flags, dataType, countHint);
// encode the vector (not shown)
encoder.encodeAggregateComplete();

// get the encoded message from the encoder
OMMMsg encodedMessage = (OMMMsg)encoder.getEncodedObject();

// the encoder may be released
pool.releaseEncoder(encoder);
```

#### Example 7: Creating Message With Encoded Attributes and Payload

## 6.15 Decoding Messages

The message elements can be retrieved using the accessors methods. The mandatory data are present in every OMM message. The presence of optional data can be tested using the hint flags. Refer to Sections 6.2.1 and 6.2.6 for information on message elements and hint flags. The attrib part of the attribute information element may be encoded on the message as a dataformat. In this case, this element needs to be retrieved as OMMData and decoded. If the message contains payload, it needs to be retrieved as OMMData and decoded.

### 6.15.1 Decoding the Message Header

The message type and message model type are retrieved using the methods `getMsgType()` and `getModelType()`, respectively.

The optional elements have hint flags associated with them to indicate their availability. The presence of these flags needs to be verified using the `has()` method prior to obtaining the associated data. For example, the presence of priority information is verified using the `OMMMsg.HAS_PRIORITY` flag. If available, the priority information is retrieved using `getPriority()` method, as illustrated in the following example.

## 6.15.2 Message Header Decoding Example

```
// message header
short type = message.getMsgType();
short model = message.getMsgModelType();

// priority information
if(message.has(OMMMsg.HAS_PRIORITY))
{
    OMMPriority priority = message.getPriority();
    byte priorityClass = priority.getPriorityClass();
    int priorityCount = priority.getCount();
}
```

### Example 8: Message Header Decoding

## 6.15.3 Decoding the Attribute Information

The presence of attribute information is verified using the `OMMMsg.HAS_ATTRIB_INFO` flag. If available, the attribute information is obtained as an `OMMAttribInfo` object using `getAttribInfo()` method. Refer to Sections 6.2.4 and 6.2.4.2 for information on attribute information elements and indication flags.

The availability of attribute elements is verified by checking for the associated flags, and the information is decoded using `getXXX()` methods. The `attrib` element is retrieved as `OMMData` object. This object is decoded using methods described in Chapter 1.

In the following example, the presence of name is verified by checking for the `OMMAttribInfo.HAS_NAME` flag. If available, it is obtained using `getName()` method. The `attrib` element is decoded using an iterator. This is the typical routine used to decode dataformats:

```
if(_message.has(OMMMsg.HAS_ATTRIB_INFO))
{
    OMMAttribInfo ai = message.getAttribInfo();
    String name = null;
    int supportPost = 0;

    if(ai.has(OMMAttribInfo.HAS_NAME))
        name = in.getName();

    if (ai.has.(OMMAttribInfo.HAS_ATTRIB))
    {
        OMMData attrib = ai.getAttrib();
        // In this acse the application expects the type to be element list
        if (attrib.getType == OMMTypes.ELEMENT_LIST)
        {
            OMMElementList list = (OMMElementList)attrib;
            OMMElementEntry element;

            for (Iterator iter = list.iterator(); iter.hasNext(); )
            {
                Element = (OMMElementEntry) iter.next();
                OMMData data = element.getData();
                if (element.getName().equals(RDMUser.Attrib.SupportOMMPost))
                    int supportPost = (int) ((OMMNumeric)data).toLong();
            }
        }
    }
}
```

### Example 9: Decoding Attribute Information

## 6.15.4 Decoding Publisher Principal Identity Information

The presence of `PublisherPrincipalIdentity` information (visible publisher identifier) is verified using `OMMMsg.HAS_PUBLISHER_INFO` flag. If available, `PublisherPrincipalIdentity` information is obtained as an `PublisherPrincipalIdentity` object using the `getPrincipalIdentity()` method. The example below shows how to extract `PublisherPrincipalIdentity` from an `OMMMsg`.

```
if ((msg.has(OMMMsg.HAS_PUBLISHER_INFO)) || (msg.getMsgType() == OMMMsg.MsgType.POST))
{
    PublisherPrincipalIdentity pi = (PublisherPrincipalIdentity)msg.getPrincipalIdentity();
    if (pi != null)
    {
        System.out.println("Publisher Address: 0x" + Long.toHexString(pi.getPublisherAddress()));
        System.out.println("Publisher Id: " + pi.getPublisherId());
    }
}
```

**NOTE:** Visible publisher identifier information is also available via Field Identifiers defined from the publisher component. For details, refer to the publishing component's documentation.

From the `StarterConsumer_Post` example:

```
<-- PostItemManager: Received for IBM.N OMM_ITEM_EVENT/MARKET_PRICE
   (com.reuters.rfa.internal.session.omm.OMMSubHandleImpl@e9625d) MsgType.REFRESH_RESP
MESSAGE
Msg Type: MsgType.REFRESH_RESP
Msg Model Type: MARKET_PRICE
Indication Flags: REFRESH_COMPLETE
Hint Flags: HAS_ATTRIB_INFO | HAS_ITEM_GROUP | HAS_QOS | HAS_RESP_TYPE_NUM | HAS_PUBLISHER_INFO | HAS_STATE
State: OPEN, OK, NONE, "All is well!"
Qos: (RT, TbT)
Group: 00011928003d685f
RespTypeNum: 1 (RespType.UNSOLICITED)
Publisher Address: 0xa5ba142
Publisher Id: 4988
AttribInfo
  ServiceName: DIRECT_FEED
  ServiceId: 6440
  Name: IBM.N
  NameType: 1 (RIC)
Payload: 20 bytes
FIELD_LIST
  FIELD_ENTRY 1/PROD_PERM: 400
  FIELD_ENTRY -2000/POST_USER_ID: 4988
  FIELD_ENTRY -2001/POST_USER_ADDR: 173777218
```

### 6.15.5 Decoding the Payload

You can verify the presence of payload information by checking the data type using `getDataType()`. A value of `OMMTypes.NO_DATA` indicates that the message does not contain a payload.

If available, you can obtain the payload using the `getPayload()` method, which returns an `OMMData` object. This object is decoded as a `DataFormat`, as described in Chapter 1.

### 6.15.6 Decoding Custom OMMData

To support parsing of custom `OMMData` types, the `OMMData` interface provides the `getEncodedLength()` method and two `getBytes()` methods. The `getEncodedLength()` method returns the number of bytes that represent this `OMMData` instance. The bytes can be obtained by invoking one of the two `getBytes()` methods. One method returns a byte array with the data; the other method takes an array, and length and fills the array with the data (up to the specified length). An application receiving the custom data types should be capable of parsing this data.

# Chapter 7 Common Package

## 7.1 Common Package Overview

Refer to Section 5.10.2 for the Common Package overview.

## 7.2 Context

The Common package implements the concept of a “Context”. The Context is used to integrate multiple RFA packages into a single application, and coordinate the interaction between these packages.

An application uses the Context to manage the initialization and shutdown of RFA. Additionally, the Context provides versioning information for all RFA packages and its underlying libraries.

All methods in the Context interface are static.

### 7.2.1 Context Scenarios

An application must initialize the `Context` via the `initialize()` method before using any other RFA interfaces:

```
// Initialize RFA Context
Context.initialize();
```

#### Example 10: Initialize Context

Just before the application is ready to terminate, it should call `uninitialize()`:

```
// Uninitialize RFA Context
Context.uninitialize();
```

#### Example 11: Uninitialize Context

An application can use the `getPackageNames()` method to get the names of the currently loaded packages. An application can then use the `getPackageVersion()` method to retrieve versioning information about an individual package.

The following example assumes the application has initialized the Context, and depicts it using the `getPackageVersion()` and `getPackageNames()` methods:

```
// Get Package Versions
// Obtain the vector of package names
String[] packageNames = Context.getPackageNames();

// Iterate over the package names, extracting the version numbers
for (int i = 0; i < packageNames.length; i++)
{
    System.out.print("Package Version: " );
    System.out.println(Context.getPackageVersion(packageNames[i]));
}
```

#### Example 12: Get Package Versions

In this example, the application obtains a vector of package names via the `getPackageNames()` method. Next the application iterates over the vector extracting the version of each package via the `getPackageVersion()` method. It is important to note that the set of package names not only includes RFA packages, but also any underlying libraries referenced by RFA packages.

The Context also provides a `getRFAVersionInfo()` method to get the RFA product version:

```
// get RFA product version
String productVersion = Context.getRFAVersionInfo().getProductVersion();
System.out.println("RFA Product Version = " + productVersion);
```

### Example 13: Get RFA Product Version

The Context also provides a `getName()` method, which always returns the string "RFA". RFA packages use this name for configuration and logging.

The following UML sequence diagram shows how an application should use the `Context` interface.

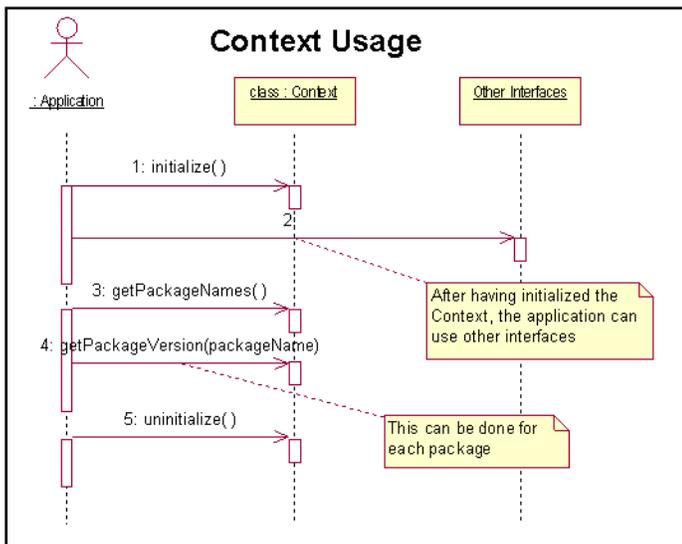


Figure 25: Context Initialization and Usage

## 7.2.2 Context Usage Guidelines

An application can make multiple calls to `Context initialize()` and `uninitialize()`. This feature is useful in an application built as a collection of independent components, as it allows for each component to initialize and uninitialize the Context independently of other application components. For a clean shutdown, the number of the `uninitialize()` calls must balance out the number of the `initialize()` calls.

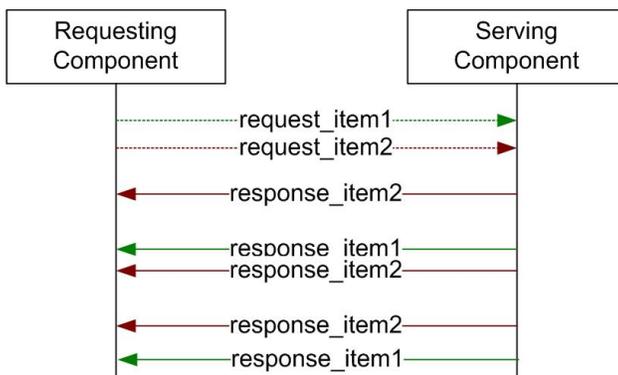
## 7.3 Event Distribution

### 7.3.1 Overview

For market information, the most typical interaction pattern for a consuming application can be described as numerous streams of items; each stream described as a “single request followed by multiple asynchronous responses”. A providing application has the reverse role in this pattern. It receives the requests and replies with the multiple responses to each request. The requests are received asynchronously.

The applications are facilitated by an RFA Session Layer. The RFA facilitating the consuming application is capable of receiving the single request and providing the multiple asynchronous responses. This replicates the provider behavior. The RFA facilitating the providing application is capable of sending the initial request and receiving the asynchronous responses. This replicates the consumer behavior.

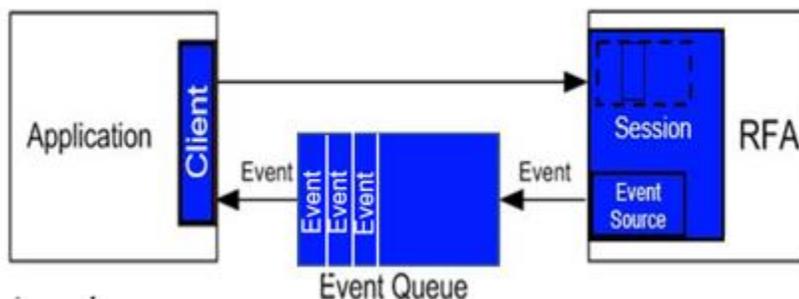
The diagram below shows an example interaction between the requesting component and a serving component. The requesting component, such as consumer application or RFA facilitating the provider application, requests information for two items in this example. The serving component, such as provider application or RFA facilitating the consumer, provides multiple requests for each item.



**Figure 26: Interaction Pattern Example**

The Event Distribution mechanism has been designed to serve this pattern. As pointed before, both the requesting and serving components, receive asynchronous events. The Event Distribution mechanism uses Event Queue, as a buffer, to handle the asynchronous events, such as requests handled by a provider application, and responses handled by a consumer application. The Event Queue is created and owned by an application. The Session layer of RFA also implements an internal queue to handle the incoming events. As an internal object, this queue is not visible component of the Event Distribution mechanism.

The diagram below shows the main components of Event Distribution mechanism: Client, Session, Event Queue and EventSource.



**Figure 27: Event Distribution Components**

The above diagram applies to any application. An application has a client object, which communicates with the Session layer. The client registers interest with the session and gets events from an event queue. The session receives the registration and uses event source to put events that the client registered for into the event queue. The event queue is created and owned by an application. The event source object is owned by a session.

The components: Client, Event Queue, Session, and Event Source form an infrastructure that needs to be set up at the application start up. Below is a list of steps to set up the Event Distribution mechanism:

- Application acquires a session.

Application specifies a session name and acquires a session with this name. The session has to be configured prior to the application start. The code below demonstrates application acquiring a session.

```
String sessionName = "mySession";
Session session = Session.acquire(sessionName);
```

The session is owned by the RFA. The application gets a reference to the session.

- Application creates Client and Event Queue

The code below demonstrates application creating an Event Queue and a client. The Client is typically a part of the application that implements Client interface. An application may use several clients. In this example below the class MyApplication implements Client and is instantiated at the start up.

```
EventQueue eventQueue = EventQueue.create("myQueue");
Client client = new MyApplication();
public class MyApplication implements Client
{
    public MyApplication()
    { ... }
}
```

- Application gets Event Source from the Session

Session has a static object factory EventSource that creates an EventSource instance. When obtaining the instance, the calling application needs to specify the type, which can be either OMMConsumer or OMMProvider. The Event Source object is owned by a Session, and the application acquiring it gets the reference. The code below demonstrates application obtaining OMMConsumer and OMMProvider Event Sources respectively. A Consumer application obtains OMMConsumer type Event Source, and the Provider application, OMMProvider.

```
EventSource eventSource = (OMMConsumer) session.createEventSource(EventSource.OMM_CONSUMER, "consumerES");
EventSource eventSource = (OMMProvider) session.createEventSource(EventSource.OMM_PROVIDER, "providerES");
```

- Application registers InterestSpec, Client, Event Queue with the Event Source.

Up to this point all the components are instantiated. The registration binds the components together. The code below demonstrates how to register a client.

```
InterestSpec interestSpec = getApplSpecificInterestSpec();
Handle handle = eventSource.registerClient(eventQueue, interestSpec, client, closure);
```

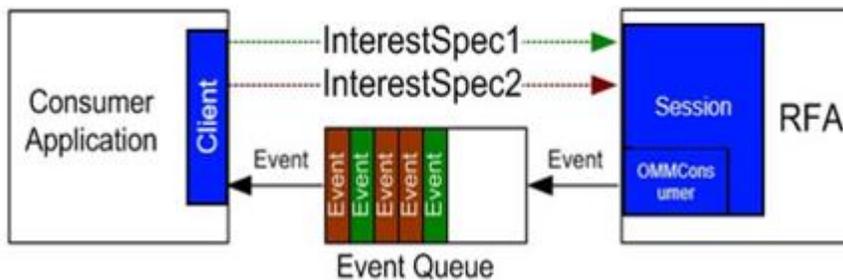
Using the [registerClient\(\)](#) call the application gives the EventSource reference to eventQueue and client. The EventSource within Session will use the eventQueue to pass the events that are generated for this client. In addition the application passes in interestSpec, which specifies which types of events the client should obtain. An Interest Specification encapsulates specific information of the interest. Optionally an object called closure is passed in. This is for an application defined use. The session returns a Handle object. A Handle is a unique object representing this registration of the Client within a session.

### 7.3.1.1 Consumer Application

The consumer application utilizes the Event Distribution mechanism to send requests and receive responses. Each request is conveyed to the RFA Session through `registerClient()` method. The consumer application uses the `OMMItemIntSpec` type of `InterestSpec`. The `OMMItemIntSpec` encapsulates a request message. The example code shows a consumer application requesting data.

```
OMMMsg requestMsg = getRequestMsg();
OMMItemIntSpec interestSpec = new OMMItemIntSpec();
interestSpec.setMsg(requestMsg);
Handle handle = consumer.registerClient(eventQueue, interestSpec, client, closure);
```

The consumer application uses this method to request data on any domain (i.e., Login, Directory, Dictionary, Market Price, etc.). Each of the request establishes an Event Stream. The application typically uses one `EventSource`, but it may instantiate multiple instances of `Event Sources` in the Session. The `Event Streams` correspond to the `Handles`. One `Client` instance can manage all or some of the `Event Streams`. An intuitive design is to have one client per domain. One event queue can be shared by all `Event Streams` or the application may create multiple `Event Queues`. In any arrangement, the application needs to manage the relations. The diagram below shows the Event Distribution mechanism applied to the example from Figure 26 used by a consumer application.



**Figure 28: Event Distribution Used by Consumer Application**

The diagram shows requests: `request_item1` and `request_item2`, encapsulated in `InterestSpec1` and `InterestSpec2` respectively. The `OMMConsumer` event source loads `Events` to the `EventQueue`. The `Client` gets the `Events` from the queue. Refer to Section 8.3.1 for the details on `OMMConsumer` Event Source.

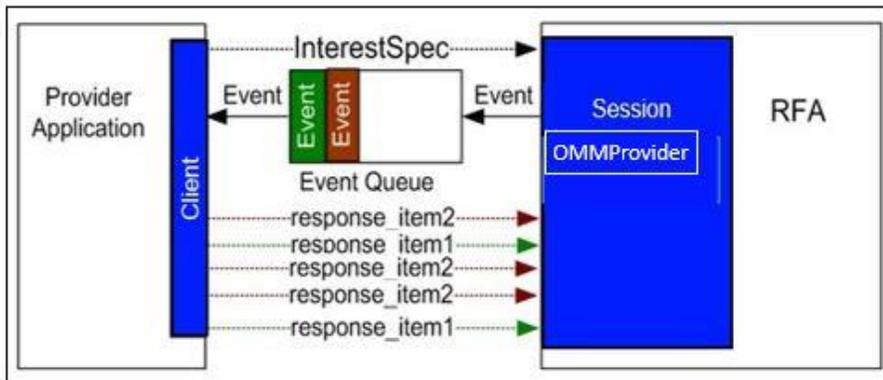
### 7.3.1.2 Provider Application

The provider application utilizes the Event Distribution mechanism to receive `Events` and send responses. In order to receive `Events` the provider has to register via `registerClient()` method. The `InterestSpec` used for the registration specifies the type of `Events` the provider will receive. A provider application in general registers with `OMMListenerIntSpec` and `OMMClientSessionIntSpec`. The `OMMListenerIntSpec` contains a connection name, from which the provider will receive `Events` when a consumer application attempts to connect. The `OMMClientSessionIntSpec` contains a handle identifying a client session. This client (i.e., provider) will receive `Events` representing requests from the connected consumer. The example code below shows a provider application registering to receive connection attempts and requests.

```
OMMListenerIntSpec listener = new OMMListenerIntSpec();
String connectionName = "configuredConnection";
listener.setListenerName(connectionName);
Handle listenerHandle = provider.registerClient(eventQueue, listener, listeningClient, closure);

// on connection attempt the provider receives Event containing a Handle
Handle handle = event.getClientSessionHandle();
OMMClientSessionIntSpec interestSpec = new OMMClientSessionIntSpec();
interestSpec.setClientSessionHandle(handle);
Handle clientSessionHandle = provider.registerClient(eventQueue, interestSpec, client, closure);
```

The provider application registers with `OMMClientSessionIntSpec` to obtain requests on any domain (i.e., Login, Directory, Dictionary, Market Price, etc.). Like in Consumer application, each of the request establishes an Event Stream. The application typically uses one `EventSource`, but it may instantiate multiple instances of Event Sources in the Session. The Event Streams correspond to the Handles. One client instance can manage all or some of the Event Streams. Typically the provider application allocates one client for each consumer connection. One Event Queue can be shared by all clients or the application may create multiple Event Queues. The application needs to manage the relations. The diagram below shows the Event Distribution mechanism applied to the example from Figure 26 used by a provider application.



**Figure 29: Event Distribution Used by Provider Application**

The diagram shows the provider application registering to receive requests, using the `InterestSpec`. The `OMMProvider` event source loads Events to the `EventQueue`. The client gets the Events from the queue, which are requests: `request_item1` and `request_item2`. For each request the client sends multiple responses. Refer to Section 8.3.2 for the details on `OMMProvider` Event Source.

### 7.3.1.3 Event Distribution Processing

Once the Event Distribution mechanism is set up by an application, the application starts receiving Events for which it registered. The events are placed in the event queue by the event source. The events are retrieved from the event queue via `dispatch()` method, called by application. The application implements own dispatching system. It may call the `dispatch()` method periodically or on a notification. The Event Distribution mechanism responds with calling `processEvent()` method on the client that registered for this event. This method presents the event to the client. By dispatching Events in its own context (or contexts) the application maintains full control over the number of the threads it has and their usage. The application can also maintain request/response affinity (i.e., it can make sure that all Events are processed in the same thread context that was used to make an original request). Both the Event Queue and the Event Handler to be used, for all events associated with a specific request, are specified when making the request.

One aspect of application processing is sending messages that are not part of Event Distribution mechanism (i.e., register-event), but rather response messages sent by the provider application, or messages sent on open stream by consumer application. This processing uses `OMMcmd` interface, defined by Session package. Refer to Section 8.4 for details.

## 7.3.2 Event Distribution Interfaces

The UML class diagram below presents a high-level view of all Event Distribution interfaces. The interfaces are described in detail in the following sections.

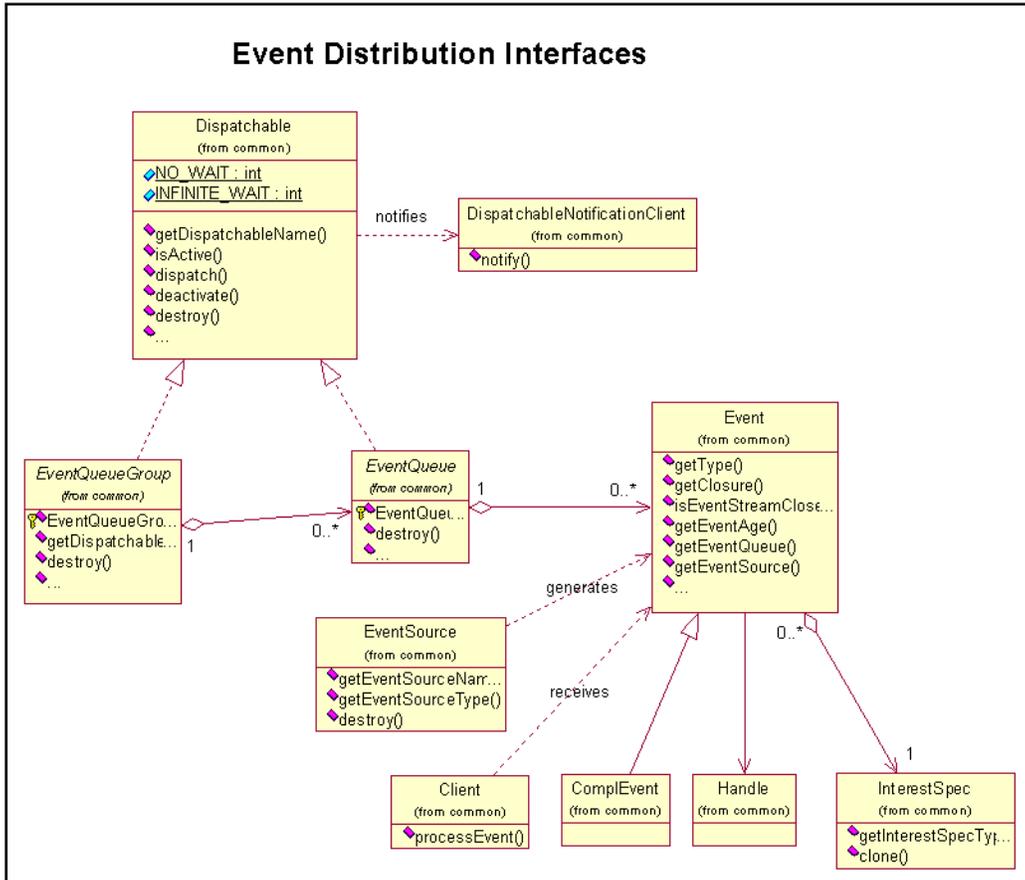


Figure 30: Event Distribution Interfaces

### 7.3.2.1 Dispatchable Interface

The Dispatchable interface is implemented by the EventQueue and the EventQueueGroup. The Dispatchable object provides the means to dispatch Events to Clients.

The table below lists methods supported by a Dispatchable interface. The interface supports also several methods that are used by a notification client, which are not listed here. These methods are described in Section 7.4.

METHOD	DESCRIPTION
<code>deactivate()</code>	This method deactivates the queue and in result no more events may be enqueued. The events that are already in the queue should be dequeued by calling the <code>dispatch()</code> method.
<code>destroy()</code>	Destroys the queue; no more events may be enqueued or dequeued.
<code>dispatch()</code>	This call dispatches the next event in the queue. It takes a parameter that specifies how long, in milliseconds, the thread will wait for an event, if none is in the queue. If the argument is set to <code>INFINITE_WAIT</code> value, the <code>dispatch()</code> method blocks until an Event becomes available or until the Queue (or the Queue Group) is deactivated. If set to <code>NO_WAIT</code> , the <code>dispatch()</code> method polls the Queue, or the Group, and returns immediately if there are no messages.  This method returns number of events left in the queue (application developers should not rely on the exact value, as it is only an estimate) or -1 if the specified time passed and there were no events.
<code>getDispatchableName()</code>	Each EventQueue and EventQueueGroup have a name. This method returns the name.

METHOD	DESCRIPTION
hasActiveEventStreams()	Returns true if the queue has any active event stream. Application developers should not rely on the exact value, as it is only an estimate.
isActive()	Returns true if the queue is active, false, if it has been deactivated.

**Table 20: Dispatchable Interface Methods**

### 7.3.2.2 Event Queue Interface

Event Queues are used by all RFA packages that need to deliver information asynchronously to an application. An application can have one Event Queue for all Events or multiple Event Queues that handle different types of Events (e.g., to handle responses of some type at a higher priority than responses of other types). Multiple Event Queues can be combined into a single Event Queue Group.

The table below lists methods supported by all instances of EventQueue.

METHOD	DESCRIPTION
create()	This method takes a String and creates an instance of EventQueue with this name. If a queue with the name already exists, another instance with this name is created.  An overloaded method takes additionally Boolean parameter, which enables statistic on the queue. This not should be used by high performing applications, since the statistic affects performance.
getEventQueueCurrentSize()	Returns the number of events in the queue.

**Table 21: EventQueue Methods**

When the EventQueue has statistic enabled, additional functionality is obtainable. The application can set the depth of the queue, i.e., the number of events the queue can hold. The application can also set the low and high threshold values. When the threshold is reached the application receives notification. It then can take an action, such as reset the queue.

The table below lists methods supported by a EventQueue instances with the statistic enabled.

METHOD	DESCRIPTION
getEventQueueHighThreshold()	Returns the high threshold value.
getEventQueueLowThreshold()	Returns the low threshold value.
getEventQueueMaxDepth()	Returns maximum number of events the queue can hold.
resetEventQueue()	Clears any remaining events from an event queue and restarts queuing.
setEventQueueMaxDepth()	Sets the size of the queue.
setEventQueueThreshold()	Sets the low and high threshold values.

**Table 22: EventQueue Statistic Methods**

### 7.3.2.3 Event Queue Group Interface

Multiple Event Queues can be combined into a single Event Queue Group by adding the Queues, one by one, via the [addEventQueue\(\)](#) method on the [EventQueueGroup](#) interface. The purpose of having Event Queue Groups is to simplify the application's dispatching of Events from multiple Queues. If desired the application can use multiple Event Queue priorities, within an Event Queue Group, to provide fine-grained control over the dispatch order of Queues.

Once a Queue has been added to a Group, the application should call the [dispatch\(\)](#) method on the group, not the queue. This will result in dispatching an Event from one of the Queues within the Group. The Queue is determined by an algorithm used by the Group.

Presently the Group supports two algorithms specified by the [setAlgorithm\(\)](#) method using the [AlgorithmType](#). An [AlgorithmType](#) value of [ABACAB](#) implies RFA more frequently dispatches Events from higher priority Event Queues but also dispatches from lower priority Event Queues less frequently. More specifically, [ABACAB](#) implies RFA dispatches Events from Event Queues with a high priority about 50 percent of the time, a medium priority about 33.3 percent of the time and a low priority about 16.7 percent of the time. If two or more queues have identical priorities, subsequent [dispatch\(\)](#) calls rotate through them in a round-robin fashion.

An `AlgorithmType` type of `HIGH_TO_LOW` behaves similar to the `ABACAB` type. However, Events from lesser priority Queues will be interspersed with Events from high priority Queues. The lesser priority Events will be dispatched at a lower rate.

The table below lists methods supported by a `EventQueueGroup` interface.

METHOD	DESCRIPTION
<code>create()</code>	This method takes a String and creates an instance of <code>EventQueueGroup</code> with this name. If a group with the name already exists, another instance with this name is created.
<code>addEventQueue()</code>	The method takes two arguments: event queue and <code>QueuePriority</code> . It adds the queue to the group and sets the queue priority. The priority can be defined as: <code>HIGH</code> or <code>NORMAL</code> .
<code>hasEventQueue()</code>	It takes an event queue argument. It returns true if the queue is already in this group, and false otherwise.
<code>removeEventQueue()</code>	Removes specified queue from the group.
<code>setAlgorithm()</code>	Sets the dispatching algorithm. The algorithm can be set to <code>ABACAB</code> or <code>HIGH_TO_LOW</code> .

**Table 23: EventQueueGroup Methods**

#### 7.3.2.4 Event Interface

The RFA package delivers asynchronous notifications to the application as *Events*. An Event is an object that encapsulates all information to be delivered to the application, as well as the information identifying the application's *Interest Specification* that resulted in this Event. The sequence of Events associated with a specific request is called an *Event Stream*.

All RFA packages that need to deliver information asynchronously to an application use Events. Different interfaces are used to represent specific types of Events. All these event interfaces inherit from the common `Event` interface and are defined by the respective packages. The specific Events contain information related to this event. OMM supports the following Event types:

- `COMPLETION_EVENT`
- `OMM_ACTIVE_CLIENT_SESSION_PUB_EVENT`
- `OMM_CMD_ERROR_EVENT`
- `OMM_INACTIVE_CLIENT_SESSION_PUB_EVENT`
- `OMM_ITEM_EVENT`
- `OMM_LISTENER_EVENT`
- `OMM_SOLICITED_ITEM_EVENT`
- `PUB_ERROR_EVENT`
- `TIMER_EVENT`
- `UNDEFINED_EVENT`

The Common package defines a `Comp1Event` of a `COMPLETION_EVENT` type. The `Comp1Event` interface represents Completion Events that are optionally sent by the Event Distribution mechanism after it has finished the processing of a request cancellation. The other types are defined by Session package.

The UML diagram at the beginning of Section 7.3.2 shows the Event object with relation to numerous objects. Each Event has a reference to an associated Handle, related EventQueue, related EventSource and Interest Specification that initiated the Event.

The table below lists methods supported by a Event interface.

METHOD	DESCRIPTION
<code>getClosure()</code>	This method returns a reference to some opaque data, called <i>Closure</i> , provided by the application when registering an interest.
<code>getEventAge()</code>	The RSSL type connection (consumer) has configuration parameter <code>enableOMMEventAge</code> . If it is configured to true, this method returns time in milliseconds, elapsed since the event has been created. If configured to false, this method returns <code>-1</code> .
<code>getEventAgeNano()</code>	The Refinitiv Source Sink Library type connection (consumer) has configuration parameter <code>enableOMMEventAge</code> . If it is configured to true, this method returns time in nanoseconds, elapsed since the event has been created. If configured to false, this method returns <code>-1</code> .
<code>getEventQueue()</code>	Returns an <code>EventQueue</code> reference that the Event has been enqueue by.

METHOD	DESCRIPTION
getEventSource()	Returns the Event Source (OMMProvider or OMMConsumer) that handled the Event.
getHandle()	Returns Handle associated with this Event.
getInterestSpec()	Returns a reference to the InterestSpec that the application used to register for this Event.
getType()	Returns type of this Event.
isEventStreamClosed()	Returns true if the Event stream has been closed, false otherwise. The stream is closed in the following cases: <ul style="list-style-type: none"> <li>The event cancels initial request</li> <li>The Event Source closed the request on its own (i.e., because the request cannot be satisfied).</li> </ul>
toString()	A utility method, returns String representation of the Event type.

**Table 24: Event Methods**

### 7.3.2.5 InterestSpec Interface

Events are generated in response to an application's registration of some type of interest. Different interfaces are used to represent specific types of Interest Specifications. The Common package defines the [InterestSpec](#) interface, which is a base class for all package-specific Interest Specifications.

The Interest Specification types supported by OMM are listed below.

- OMM\_CLIENT\_SESSION\_INTEREST\_SPEC
- OMM\_ERROR\_INTEREST\_SPEC
- OMM\_ITEM\_INTEREST\_SPEC
- OMM\_LISTENER\_INTEREST\_SPEC
- TIMER\_INTEREST\_SPEC
- UNKNOWN

The Interest Specifications of the types specified above are defined in Session package. Refer to Section 8.5 for details. The specific InterestSpec object contain specific information, such as the OMM\_ITEM\_INTEREST\_SPEC type InterestSpec contains request message; the OMM\_LISTENER\_INTEREST\_SPEC type InterestSpec contains connection name.

The table below lists methods supported by a InterestSpec interface.

METHOD	DESCRIPTION
clone()	The InterestSpec implements Clonable and all types should define this method.
getInterestSpecType()	Returns the type of InterestSpec instance.

**Table 25: InterestSpec Methods**

### 7.3.2.6 Event Source

The [EventSource](#) interface is a common base class for all interfaces that can be used to register an interest of any type. Because application's interest registration typically results in getting one or more Events, all these interfaces represent some sort of an Event Source (thus the name for the base class). Individual RFA packages define Event Sources of package-specific types. These package-specific Event Sources are represented as interfaces that inherit from the [EventSource](#) interface.

The Event Source types supported by OMM are listed below.

- OMM\_CONSUMER
- OMM\_PROVIDER

The Event Sources of the types specified above are defined in Session package. Refer to Section 8.3 for details. The table below lists methods supported by a EventSource interface.

METHOD	DESCRIPTION
destroy()	Destroys the instance of EventSource. This method is used by an application thread that created the instance.
getEventSourceName()	Returns the Name of Event Source instance.
getEventSourceType()	Returns the type of Event Source.

**Table 26: InterestSpec Methods**

### 7.3.2.7 Client

A Client is some object in an application space. It provides a callback for asynchronous events. Refer to Section 7.3.1 for the diagram showing Client component in the Event Distribution architecture. A Client registers with the Interest Specification for Events, and when the events are available, they are delivered to the Client via dispatch mechanism. The Event is delivered using `processEvent()` method. This method should be implemented by the application client object.

### 7.3.2.8 Handle

A Handle is a unique request identifier in the Event Distribution mechanism. The Event Distribution mechanism assigns a distinct **Handle** each time an application makes a calls `registerClient()` method. Each event received as a result of a previously made request contains, among other things, the corresponding handle. The sequence of Events associated with a specific request form an **Event Stream**. Each Event Stream corresponds to a unique Handle. A Handle instance should implement `isActive()` method. This method returns true if the Event Stream identified by the handle is active. The stream is closed in the following cases:

- An event closing the request occurred.
- The Event Source closed the request on its own (i.e., because the request cannot be satisfied).

### 7.3.2.9 Token

A Token objects applied to the provider type of applications. The Token interface is used as a type –safe way of returning an object. It references and maintains state on an internal data structure. While the Handle interface is solely applicable for Interest Specifications, the Token interface application also includes non-interest specifications. Token interface supports `getHandle()` method, which returns Handle associated with this Token. It also supports `isActive()` method, which returns true, if the associated Handle is active, and false, otherwise.

## 7.4 Notification Client

An application can use a Notification Client to integrate the Event Distribution logic into an existing event dispatching mechanism; such as a Windows message loop in Microsoft Windows environment or a `select()` based event loop in Unix. To register a Notification Client, an application needs to declare a class that implements the `DispatchableNotificationClient` interface, instantiate an object of that class, and then register it with an Event Queue or an Event Queue Group using the `registerNotificationClient()` method supported by the `Dispatchable` interface. When the client is not needed anymore, an application can unregister it using the `unregisterNotificationClient()` method.

A Notification Client is an interface used to notify the application of newly available Events on an Event Queue or Event Queue Group. Typically, applications use Notification Clients to signal a synchronization object, which wakes up an alternate thread of control. The alternate thread of control then calls `dispatch()` knowing an Event will be available from the respective Event Queue or Event Queue Group. An application should call `dispatch()` until the queue is exhausted. The application typically uses a Notification Client for integrating with the AWT thread. In addition, an Event Queue with statistics allows for client notifications at low threshold, high threshold, and maximum depth levels.

It is important that applications not make any RFA method calls or perform any intensive processing in a Notification Client. This is because a Notification Client always executes in RFA's thread context. Performing intensive processing may negatively affect internal performance of RFA.

## 7.4.1 Dispatchable Interface Methods

The table below lists methods supported by a Dispatchable interface that apply to the Notification Client.

METHOD	DESCRIPTION
registerNotificationClient()	This method takes a DispatchableNotificationClient instance as a parameter and registers the client in this instance of Dispatchable (EventQueue or EventQueueGroup). The registered client will receive notification (i.e., the <code>notify()</code> method on this client is called) if any event is in the queue.
registerHighThresholdNotificationClient()	This method registers a client to be notified when the Event Queue high threshold value is reached. The threshold value can be set on a queue when statistics are set. Refer to Table 22.
registerLowThresholdNotificationClient()	This method registers a client to be notified when the Event Queue low threshold value is reached. The threshold value can be set on a queue when statistics are set. Refer to Table 22.
registerMaxDepthNotificationClient()	This method registers a client to be notified when the Event Queue maximum depth value is reached. The maximum depth value can be set on a queue when statistics are set. Refer to Table 22.
unregisterNotificationClient()	Unregisters the notification client.
unregisterHighThresholdNotificationClient()	Unregisters the notification client.
unregisterLowThresholdNotificationClient()	Unregisters the notification client.
unregisterMaxDepthNotificationClient()	Unregisters the notification client.

**Table 27: Dispatchable Methods for Notification Client**

The DispatchableNotificationClient interface supports `notify()` method. This method should be implemented by the application instance of DispatchableNotificationClient.

## 7.4.2 Notification Client Example

In this example of a Notification Client call-out method, the application defines the class `SimpleNotificationClient` by implementing the `DispatchableNotificationClient` interface. The application implements the `notify()` call-out method, which will be invoked by RFA just prior to calling the `processEvent()` method.

```
// Receive Notification - SimpleNotificationClient
public class SimpleNotificationClient implements DispatchableNotificationClient
{
    public void notify( Dispatchable dispSource, java.lang.Object closure )
    {
        //Signal another thread to call dispatch()
    }
};

SimpleNotificationClient mySimpleNotificationClient = new MySimpleNotificationClient();
```

**Example 14: Process Notification Client**

## 7.5 Status

The Common package provides a simple Status interface. Various RFA packages define specific status classes that implement the interface.

The interface defines the `getStatusText()` method, which returns a text that augments the numeric State and Status Code exposed by the derived classes. The `getStatusText()` method returns the text as an ASCII string and the `setStatusText()` method allows for setting this text string.

## 7.6 RFA Exceptions

RFA Common package defines the following exceptions:

- `DeactivatedException`
- `DispatchException`
- `DispatchQueueInGroupException`

See Javadoc for details.

## 7.7 Refinitiv Wire Format Versioning

The RFA VersionInfo interface provides the `getProductVersion()` method, which returns string with the RFA version information. This information is available to application after a session is acquired. The version information is set during context initialization. Refer to Section 7.2 to see how to use context to obtain the product version. Obtaining the version information using context is an alternative way to calling the `getProductVersion()` method.

The RWF is the foundation of OMM (OMM) based products. Future enhancements or extensions on RWF can result in Refinitiv issuing a new version. By versioning these interfaces, OMM messages of different RWF-encoded versions can flow between applications, RFA, and infrastructure components. Refer to Section 6.12 to see how the OMM package supports versioning.

Every connection has version information associated with it, and when a connection is established, that version will be negotiated between the provider and the consumer. In the event of differing versions, the lower version between the two will be used as the negotiated version. RWF data encoded using the negotiated version will be sent out through the network.

Before RWF data is sent to the network, RFA ensures that the data has been encoded with the negotiated version. If the version used to encode data does not match with the negotiated version from established connection, RFA will decode and re-encode data by using negotiated version.

The versioning interface from RFA allows client application to set and get version information on `OMMData` and `OMMMsg` objects. RFA will use the default version indicated in the configuration to encode or decode Refinitiv Wire Format data implicitly if no version information is passed by the application. In environments running multiple versions, it is better for the client application to pass version information to prevent a degradation in performance.

RFA releases with versioning support will have backwards compatibility with previous RFA and RFA releases with versioning support.

## 7.8 Quality of Service

### 7.8.1 Overview

In network terminology, **Quality of Service** (more specifically referred to as Differentiated Class of Service) is a categorical method of classifying traffic into separate tiers to provide differentiated services within a network. With respect to RFA, Quality of Service is a categorical method of classifying services provided by delivery infrastructure.

Quality of Service provides the ability for applications to access data with different qualities of service according to application requirements, user permissions and service availability.

An application requesting interest in OMM Data may specify a desired Quality of and obtain the actual Quality of Service in response. Additionally, an application satisfying an item request may specify the Quality of Service.

The following table lists the Quality of Service classes defined in common package.

INTERFACE	DESCRIPTION
QualityOfService	A class defining data Quality.
QualityOfServiceRequest	A class defining desired QualityOfService properties.

**Table 28: Quality of Service Interfaces**

### 7.8.2 QualityOfService

The Quality of Service is defined by two attributes: **Data Timeliness** and **Data Rate**. The data timeliness defines the quality/age associated with the data content. The data rate describes the delivery style of updates on an open stream.

The table below summarizes possible timeliness values:

VALUES	DESCRIPTION
REALTIME	No delay is applied to the data; all events are delivered in real time (when the event happens).
UNSPECIFIED_TIMELINESS	Timeliness has not been defined.
UNSPECIFIED_DELAYED_TIMELINESS	A fixed delay (such as 60 seconds) is applied to the data.
A positive number	The actual delay in seconds within range 0 – 0x7ffffff.

**Table 29: Timeliness Definitions**

The table below summarizes possible rate values.

VALUES	DESCRIPTION
TICK_BY_TICK	The application receives every update. The period (i.e., time interval between successive events) varies based on the market activity.
JUST_IN_TIME_FILTERED	The period (i.e., time interval between successive changes in data) varies during different time intervals. RFA may deliver data with a fixed period at some time intervals while at other time intervals RFA may deliver data tick-by-tick. Updates are delivered at a rate based on bandwidth, congestion and flow control. These updates will represent an aggregate of the updates that happened within that interval.
UNSPECIFIED_RATE	Rate has not been defined.
A positive number (Time Filtered)	Updates are delivered at a set interval period (e.g. 1 or 5 milliseconds). These updates will represent an aggregate of the updates that happened within that interval. The actual rate in milliseconds within range 0 – 0x7ffffff.

**Table 30: Rate Definitions**

Combining the above attributes yield the following Quality of Service definitions:

- Realtime: TickByTick
- Realtime: TimeFiltered
- Realtime: JustInTimeFilteredRate
- Delayed: TickByTick
- Delayed: TimeFiltered (many different filtering rates)
- Delayed: JustInTimeFilteredRate

The Quality of Service definitions above are ordered from the best (smallest value) to the worst (greatest value) QualityOfService. The QualityOfService class implements the Comparable interface and thus supports compare functionality through `compareTo()`, `equals()`, `greaterThan()`, `greaterThanOrEqualTo()`, `lessThan()`, `lessThanOrEqualTo()` methods.

The TimeFiltered data rate can have different values, ranging from 1 to 0x7fffff-2. Therefore two instances of QualityOfService with Realtime: TimeFiltered or Delayed: TimeFiltered values are not equal if the TimeFiltered values are different.

The QualityOfService class defines the frequently used values for Quality of Service as follows:

DEFINITION	VALUE
QOS_UNSPECIFIED	(UNSPECIFIED_TIMELINESS, UNSPECIFIED_RATE)
QOS_REALTIME_TICK_BY_TICK	(REALTIME, TICK_BY_TICK)
QOS_REALTIME_JIT	(REALTIME, JUST_IN_TIMEFILTERED)

**Table 31: Quality of Service Definitions**

### 7.8.3 QualityOfServiceRequest

An application specifying interest in an item may use the `QualityOfServiceRequest` attribute to define a range of Quality of Service that is acceptable. The range is specified by the worst timeliness and rate and the best timeliness and rate. If the application wants a specific Quality of Service, it sets the Best and Worst attributes to the same values. If the application does not specify the `QualityOfServiceRequest` attribute, the system will use default values (see the table below for details).

The `QualityOfServiceRequest` attribute also specifies whether the data quality of service can change following the initial image. This request is specified by the StreamProperty. The StreamProperty may have one of two values: **Static** or **Dynamic**. **Static** implies the Quality of Service will not change following the initial image. **Dynamic** implies the Quality of Service may change following the initial item image.

The following table lists properties, supported values, and default values of a QualityOfServiceRequest.

PROPERTY	SUPPORTED VALUES	DEFAULT VALUE
BestTimeliness	REALTIME DELAYED	REALTIME
WorstTimeliness	REALTIME DELAYED	DELAYED
BestRate	TICK_BY_TICK JUST_IN_TIME_FILTERED_RATE FASTEST_FILTERED_RATE SLOWEST_RATE (A positive number representing the actual rate in milliseconds)	TICK_BY_TICK
WorstRate	TICK_BY_TICK JUST_IN_TIME_FILTERED_RATE FASTEST_FILTERED_RATE SLOWEST_RATE (A positive number representing the actual rate in milliseconds)	SLOWEST_RATE
StreamProperty	STATIC_STREAM DYNAMIC_STREAM	STATIC_STREAM

**Table 32: QualityOfServiceRequest Interface: Properties**

Each of the properties above can be set or obtained using accessor methods.

The default values provide applications that are unaware of Quality of Service the ability to receive different level of Quality of Service based on available infrastructure and configuration. The above default values prefer the best Quality of Service, but allow for a lesser Quality of Service, which may vary depending on availability, entitlements, and network congestion. An application may change these parameters to specify a different desired Quality of Service.

## 7.9 Common Package Usage Guidelines

### 7.9.1 Event Distribution Usage Guidelines

#### 7.9.1.1 Event Queues and Event Queue Groups

Once an Event Queue has been added to an Event Queue Group, the ownership of the Queue is transferred to the Group. This means that the application should not attempt to call the `dispatch()` method on an individual Event Queue that is a member of a Group, nor should the application attempt to `destroy()` a Queue that has been added to a Group. Instead it should either remove the Queue from the Group first and then `destroy()` the Queue, or call the `destroy()` method on the Group. The latter will destroy both the Group and all the Queues previously added to the Group, once they are not needed.

The application must not destroy an Event Queue, or an Event Queue Group, while it has Active Event Streams. In a multi-threaded application, the only reliable way to ensure that a Queue or a Group does not have an Active Event Stream, is to call the `deactivate()` method of the `EventQueue` or the `EventQueueGroup` interface. In a single-threaded application, it is acceptable to rely on getting the “No Active Event Streams” return value from the `dispatch()` call.

#### 7.9.1.2 Using Event Queues and Event Queue Groups from Multiple Threads

An application must not use multiple threads at the same time for dispatching events from a single Event Queue or a single Event Queue Group.

On the other hand, an application may use different threads to dispatch Events from different Event Queues or Event Queue Groups. One reason for doing this might be to have multiple threads simultaneously working on different Event Queues in order to take advantage of multiprocessor hardware.

An application can use the same thread that makes requests to also dispatch Events, or it can have one or more threads making requests while having a separate thread dispatching Events.

#### 7.9.1.3 Event Processing Client Restrictions

Event objects are only valid within the Event Processing Client function context. Once the application returns from the `processEvent()` call, the Event object is no longer valid. If the application wishes to asynchronously process the Event, it must make its own copy for later processing.

The application does not control the life cycle of an Event object received via the `processEvent()` call. The application should never invoke the `destroy()` method of any object that can be obtained using the `Event` interface methods such as `getEventQueue()`, `getEventSource()`, etc.

The following methods cannot be called from an Event Processing Client `processEvent()` call:

- Any of the methods of an `EventQueue` or an `EventQueueGroup`, except `getName()`.
- The `destroy()` method of an Event Source associated with the current Event.

It is permissible to make a new request or requests from the Event Processing Client function context.

Prior to deleting an instance of an Event Processing Client, the application must make sure that the Client instance will not be used in a `dispatch()` call. The rules for ensuring this are different depending on whether the application uses a single thread or multiple threads to dispatch Events, make and cancel requests, and to destroy the Client instance.

The single-threaded case is simpler: The application needs to cancel all currently active requests that were made with the instance of the Client specified. Then it can delete the Client instance.

In a multi-threaded case, the only reliable way of ensuring that the Client instance will not be used in a `dispatch()` call is to use Completion Events. The application needs to cancel all currently active requests that were made with this instance of the Client specified and wait for a Completion Event for each of these requests. Then it can delete the Client instance.

Alternatively (both in single-threaded cases and multi-threaded cases), the application can deactivate all Event Queue(s) or Event Queue Group(s) with which this instance of a client is associated, and then delete the Client instance.

### 7.9.1.4 Notification Client Restrictions

Typically, an application would use a Notification Client to integrate the Event Distribution mechanisms with its own (application specific) event mechanism. Because the Event Distribution mechanisms call the Notification Client in its own (application) thread context, there are severe limitations on what can be done from within the Notification Client call. The only method that is safe to call is the `getName()` method on an instance of an Event Queue or an Event Queue Group that is passed to the Notification Client as a parameter. The application can use this information to control its own event processing logic (e.g., by signaling a platform-specific event object, or sending a message to the application's message processing loop). No other methods on any of the RFA interfaces should be called, directly or indirectly. The application should not perform any CPU-intensive processing or make any calls that can block for any significant period of time.

### 7.9.1.5 State and Status Code in Status Interfaces

Implementors of the Status interface typically provide State and Status Codes. State typically conveys the health of data. For example, the Session Layer's `OMMErrorStatus` class conveys the errors related to OMMProvider. Typical applications are expected to programmatically react to State but not Status Code. However, there are no requirements on reacting to State, Status Code, or either.

## 7.9.2 Using Event Distribution Model in a Single Thread Context

Using the Event Distribution Model in a single thread context implies the same application thread initializes RFA interfaces, obtains interest, relinquishes interest, and shuts down RFA interfaces. Using the Event Distribution Model in a single thread context involves no additional activities than those described in previous sections.

As indicated above, either the application or RFA may close the Event Stream. In the case when the application closes the Event Stream, RFA dispatches no more Events following the return to close the Event Stream. Hence, the return to close the Event Stream defines the point at which RFA designates the close of the Event Stream. In the case when RFA closes the Event Stream, RFA dispatches no more Events following indication that the Event Stream has been closed via the `isEventStreamClosed()` method, as described in Section 7.3.2.

## 7.9.3 Using Event Distribution Model in Multiple Thread Contexts

Using the Event Distribution Model in multiple thread contexts implies different application threads initialize RFA interfaces, obtain interest, relinquish interest, and shut down RFA interfaces. Using the Event Distribution Model in multiple thread contexts possibly requires dispatching Completion Events.

In the case when the application closes the Event Stream, RFA may dispatch additional Events due to context switching. To determine the point at which RFA designates the close of the Event Stream, an application should use Completion Events. A Completion Event is a specific type of Event guaranteed to be the last Event. Unless RFA is about to initiate the close of the Event Stream, RFA dispatches a Completion Event following the application's initiation to the close of the Event Stream. To use Completion Events, an application should specify this option to be `true` when initializing an Event Source using the `registerClient()` method..

Other than specifying the option to use Completion Events, an application typically need not perform any special processing for a Completion Event. A Completion Event merely always returns `true` when the application calls the `isEventStreamClosed()` method. Once the application receives `true` from `isEventStreamClosed()`, it is guaranteed to receive no more Events for this Event Stream. Hence, RFA designates the close of the Event Stream.

Some applications may desire to discern a Completion Events from other Events that may close an Event Stream. In this case, the application may add an additional case statement in the `processEvent()` method. The Completion Event is called `Event.COMPLETION_EVENT`, available in the Common package.

## 7.9.4 Thread Safety

All interfaces are thread-safe at the class level<sup>1</sup>. In addition to that, some interfaces are thread-safe at the object level<sup>2</sup>. The decision not to support object-level thread-safety for all interfaces was made for the following reasons:

- Implementing object-level thread-safety worsens the performance because of the synchronization overhead.
- Many methods exposed by RFA interfaces are not atomic, and thus making each method thread-safe would not necessarily make it possible to have multiple threads operating on the same object. In most of the cases, there is no practical need to have every single interface thread-safe at the object level.

If an application developer finds out that they need access to the same object from multiple threads, then they need to protect this object with an explicit locking mechanism.

---

<sup>1</sup> Class-level thread-safety means that static methods (if any) can be called from multiple threads at the same time, and that if there are any class-wide resources (i.e., static data members) then access to these resources from class instances is properly synchronized.

<sup>2</sup> Object-level thread-safety means that any non-static methods implemented by the class can be called on the same object (class instance) from multiple threads at the same time.

# Chapter 8 Session Package

## 8.1 Overview

The Session layer is the central component of RFA. It allocates sessions, connections and manages them to provide the data transport (refer to 8.1.1). The Session layer keeps track of application clients and controls data exchange between the clients and connection layer of the RFA utilizing the Event Distribution Model. The session layer supports many features of RFA. For details, refer to Chapter 13.

The Session package provides an interface to the application that allows access to the Session layer. Using this interface, an application can allocate a session instance within the Session layer, and use this session instance for sending and receiving data.

The Session package provides the following interfaces:

- The Session interface provides access to a session
- Event interfaces defining events handled by the Session layer, and related Status definitions

An event can contain status data. The session package defines several final status classes.

- EventSource interfaces
- Interest Specification definitions
- Command interface and definitions

For additional Session Package overview details, refer to Section 5.10.3.

### 8.1.1 Session and Connection

#### 8.1.1.1 Session

A Session is an event source factory that encapsulates one or more connections. An application uses a Session to create one or more Event Sources using the `session` interface.

Internally, the Session performs the core business logic of RFA (e.g. Quality of Service) and operates in a single thread of control. An application having multiple application components may choose to use a single Session instance or multiple Session instances. Use of a single Session instance allows the application to funnel all information into a single internal RFA thread and through a single (or set of select) Event Queues. Use of multiple Session instances allows segmenting information, perhaps to separate different types of market information. The ability for multiple application components to optionally share a single session is known as the **session sharing** feature.

The majority of requests that an application can make via the Session Layer are handled asynchronously. Some of these requests result in a single response that is delivered to the application asynchronously, while other requests result in a single response followed by a continuous stream of updates.

### 8.1.1.2 Connection

A connection encapsulates connectivity to a back-end system. Connections are not used directly by an RFA application; rather, an application deals with Sessions. Each Session encapsulates one or more connections. The difference between a Session and a connection is that the functionality of the connection is limited to a single back-end system or, in a limited number of cases, to the functionality provided by a predefined combination of back-end systems. On the other hand, a Session can be composed of multiple connections thereby bringing together the functionality that can only be provided by a combination of back-end systems.

Multiple *Sessions* can share a single *connection* (e.g., two different application components using two different Sessions but routing to the same ADS connection). This feature is known as **connection sharing**.

## 8.1.2 Configuring a Session

The configurable components contained in the Session Layer are identified by a Component Name, and follow the configuration rules defined for RFA packages in Section 9.2.2, Naming Scheme. Each configurable Session Layer component configures itself at startup (or when it is instantiated) by retrieving all the necessary configuration information from the RFA Configuration Database. Effectively, each component is aware which sub-tree it must use to configure itself.

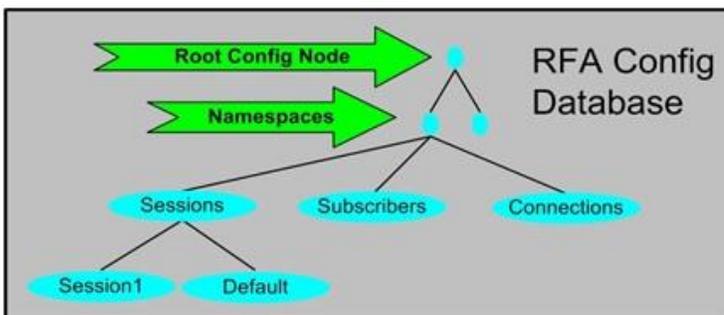


Figure 31: Session Layer Configuration Database Layout

For example, the Session component, (accessible from an application via the [Session](#) interface), requires configuration. Because an application can create more than one [Session](#), each particular instance of a Session is identified by its Component Name; this name is supplied as a parameter for the [acquire\(\)](#) method supported by the [Session](#) interface. When an instance of an implementation class that supports the [Session](#) interface is created, the implementation goes to the Configuration Database and retrieves the configuration parameters from the `com\reuters\rfa\namespace\Sessions\<SessionName>` node.

The Connection component, which handles connections to back-end systems, is exposed through configuration, but is not accessible to RFA applications programmatically. This configuration comes from the Connections Component's Relative Config Path.

The layout of the configuration database needed to configure the Session Layer is provided in *3 RFA - Java Edition – Configuration and Logging Guide*.

## 8.2 Session Interface

### 8.2.1 Session Interface Methods

The following table presents the methods supported by Session interface:

FUNCTION NAME	DESCRIPTION
acquire()	This static method takes the session name and returns a <a href="#">Session</a> object. The session name must match a session name defined in the RFA Configuration Database. As described in Section 8.1.2, the semantics of the <a href="#">acquire()</a> method supports instance sharing. The implementation of the <a href="#">acquire()</a> method uses the session name passed as a parameter to find an existing instance that has this name. If such an instance does not exist, then the implementation will instantiate a new object that supports the <a href="#">Session</a> interface. For every <a href="#">acquire()</a> call made in the application, it must call the <a href="#">release()</a> method.

FUNCTION NAME	DESCRIPTION
createEventSource()	Creates an object that implements the <a href="#">EventSource</a> interface. This method takes the type of the Event Source and the name, as arguments. If an Event Source with this name already exists, then a new instance is created with the same name. The EventSource interface defines the following types: <ul style="list-style-type: none"> <li>• EventSource.OMM_CONSUMER</li> <li>• EventSource.OMM_PROVIDER</li> <li>• Other legacy types</li> </ul> For details, refer to Section 8.3.
getName()	Returns the name of this session.
release()	This method is used by application to notify session layer that it will not use this Session instance anymore. Note that this Session can still be used by another application.
legacy	Other methods that supporting legacy protocol.

Table 33: Session Interface Methods

## 8.2.2 Session Interface Examples

The following code example gives two illustrations of how the consumer and provider applications initialize sessions, generate Consumer and Provider Event Sources, and release the sessions. The example assumes that sessions with names “[consSession](#)” and “[provSession](#)” are configured in RFA.

```
// create the consumer session
Session _session = Session.acquire("consSession");

// Create an OMMConsumer event source
_ommConsumer = (OMMConsumer) _session.createEventSource(EventSource.OMM_CONSUMER, "OMMConsumer", true);
...
// release session
_session.release();

// create the provider session
Session _session = Session.acquire("provSession");

// Create an OMMProvider event source
_ommProvider = (OMMProvider) _session.createEventSource(EventSource.OMM_PROVIDER, "OMMProvider", true);
...
// release session
_session.release();
```

Example 15: Using Session Interface

## 8.3 OMM Event Sources

### 8.3.1 OMMConsumer

The *OMM Consumer* interface is an Event Source used for creating consumer applications. It provides the capability to make requests, receive responses, send / receive generic messages, and send post messages. The Session Layer implements *OMMConsumer*.

An OMM Consumer is associated with a specific Session, and uses the back-end systems that the Session represents to satisfy the requests. The OMM Consumer implementation understands market information and other information (e.g., the concepts of an Item Image or an Item Update, receiving and sending generic messages, and the concept of an Item State). Multiple OMM Consumer event sources can be associated with any given Session.

The application typically registers an *OMMConsumer* Event Source to the following Interest Specifications:

- `omm.OMMItemIntSpec`
- `omm.OMMErrorIntSpec`
- `omm.OMMConnectionIntSpec`
- `omm.OMMConnectionStatsIntSpec`

The consumer application can register, modify or close multiple event streams using batch feature. For details, refer to Section 13.2. The *OMMConsumer* EventSource defines methods supporting the modifying and closing batch of event streams.

The *OMMConsumer* Event Source supports consumer functionality by providing additional methods shown in the following table:

FUNCTION NAME	DESCRIPTION
<code>getSession()</code>	This method returns the Session instance that owns this consumer.
<code>registerClient()</code>	<p>An application invokes <code>registerClient()</code> to open an event stream. <code>registerClient()</code> takes the following arguments: EventQueue, InterestSpec, Client, Object (closure).</p> <p>The registering Client will receive Events of the specified InterestSpec type from the specified EventQueue. InterestSpec can be an <i>OMMItemIntSpec</i>, <i>OMMErrorIntSpec</i>, <i>OMMConnectionIntSpec</i>, or <i>OMMConnectionStatsIntSpec</i> instance type:</p> <ul style="list-style-type: none"> <li>• <i>OMMItemIntSpec</i> encapsulates a streaming request or nonstreaming request message of any Domain Model (i.e., Login, Directory, Dictionary, Market Price, etc). This operation results in opening an event stream for receiving events of <i>OMMItemEvent</i> type. When <i>OMMItemIntSpec</i> is used, this method returns a Handle instance representing the open event stream. This interface can send both streaming requests and snapshot requests. In the case of snapshot requests, the stream is closed when the final refresh is received.</li> <li>• <i>OMMErrorIntSpec</i> is associated with a client. If a client registers with this Interest Specification, it can receive <i>OMMCmdErrorEvent</i> types events related to all open event streams registered by the Client. When <i>OMMErrorIntSpec</i> is used this method returns a Handle instance, representing an error handle for this Client.</li> <li>• <i>OMMConnectionIntSpec</i> is associated with a client. If a client registers with this Interest Specification, it may receive events of <i>OMMConnectionEvent</i> type. Optionally, an application can use a closure Object when registering. This Object is application-defined and provided by the interface for the user-defined purpose. The closure object specified in the request will be included in the associated response messages. The application can use it as a reference.</li> <li>• The <i>OMMConnectionStatsIntSpec</i> is associated with a client. If a client registers with this Interest Specification, it can receive events of <i>OMMConnectionStatsEvent</i> type which allows the client to receive connection statistics (i.e., the number of bytes on the wire).</li> </ul>
<code>reissueClient()</code>	<p>This method provides applications the capability to modify existing event streams opened using the <code>registerClient()</code> method. Refer to Section 6.3.2 for details on changing event stream parameters. <code>reissueClient()</code> takes the following arguments: Handle and InterestSpec. The Handle specifies which event stream is modified. The InterestSpec is an instance of <i>OMMItemIntSpec</i>. It encapsulates a streaming request message.</p> <p>When <code>reissueClient()</code> method is used to modify a batch of streams, it takes a list of Handles, instead of a single Handle. All the streams, represented by the Handles, will be modified. For details, refer to Section 13.2.</p>
<code>submit()</code>	This interface affords provider applications the ability to send refresh, update, or generic messages on an open stream and takes the arguments: <i>OMMCmd</i> and <i>Object</i> (closure). The <i>OMMCmd</i> parameter is an

FUNCTION NAME	DESCRIPTION
	<p>instance of <code>OMMItemCmd</code>, which encapsulates the refresh, update, or generic message and the token associated with the open event stream.</p> <p>An application may optionally specify a closure Object on the <code>submit()</code> call. This Object is application-defined and provided for the purpose specified by the user. The submit closure object will be included in <code>OMMCmdErrorEvent</code> events and can be retrieved by the <code>getSubmitClosure()</code> method.</p> <p>An application should specify an open event stream when calling the <code>OMMProvider submit()</code> method. If the stream is not open, the message will be dropped and the application will be notified through the <code>OMMCmdErrorEvent</code> (if the consumer had registered for an <code>OMMErrorIntSpec</code>).</p> <hr/> <p><b>NOTE:</b> <code>OMMCmdErrorEvents</code> are error notifications delivered to an application if RFA has issues processing the <code>submit()</code> call.</p> <hr/> <p>To get error notifications, an application has to register an <code>OMMErrorIntSpec</code> Interest Specification using the <code>registerClient()</code> method. Optionally, the application can specify a closure object in the <code>registerClient()</code>. This closure object is returned in the <code>OMMCmdErrorEvent</code> and can be retrieved using the <code>getClosure()</code> method.</p>
unregisterClient()	<p>This interface affords applications the ability to close existing event streams. For both Streaming and Snapshot requests, it is not necessary to invoke <code>unregisterClient()</code> on a stream after a closed status is received. For Snapshot requests, it is not necessary to invoke <code>unregisterClient()</code> after receiving the final refresh of an item.</p> <p>This method takes a handle instance as an argument. The Handle identifies the event stream to be closed.</p> <p>The <code>unregisterClient()</code> method can be used to close a login stream, a single item stream or all open item streams.</p> <p>The behavior of <code>unregisterClient()</code> is based on the type of the handle passed as an argument:</p> <ul style="list-style-type: none"> <li>• Item Handle: the item event stream is closed</li> <li>• Login Handle: the login event stream and all the open item event streams are closed, including service directory handles, dictionary handles, market price item handles, etc.</li> <li>• Null: all the item event streams except for the login stream are closed</li> </ul> <p>When <code>unregisterClient()</code> method is used to close a batch of streams, it takes a list of Handles, instead of a single Handle. All the streams, represented by the Handles, will be closed. Refer to Section 13.2 for details.</p>

**Table 34: OMMConsumer Event Source Interface Methods**

The following code example shows how a consumer application uses an OMMConsumer Event Source and OMMItemIntSpec (refer to Section 8.5) to register and unregister a client. The Event Source is associated with an event queue.

```
// EventSource _ommConsumer;

Client client = new ConcreteClient(); // application defined

// create the event queue to accept requests
EventQueue _eventQueue = EventQueue.create("Consumer Application EventQueue");

// generate the message to send
OMMMsg ommMsg = encodeOMMMsg(); // application method

// create a new item interest specification
OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();

// bind the interest specification with the message
ommItemIntSpec.setMsg(ommMsg);

// register the client
Handle handle = _ommConsumer.registerClient(_eventQueue, ommItemIntSpec, client, null);
```

```
// unregister the client
_ommConsumer.unregisterClient(handle);
```

### Example 16: Using OMMConsumer Interface

## 8.3.2 OMMProvider

The **OMM Provider** interface is an Event Source used for creating provider applications. It affords the capability to service requests and send response and generic messages. The Session package implements an **OMMProvider** Event Source. An OMM Provider also implements the capability to publish data using a broadcast-publishing style. Similar to an **OMMConsumer**, the **OMMProvider** is associated with a specific Session, and uses the back-end systems e.g. an ADH. The **OMMProvider** implementation understands market information and other information (e.g., the concepts of an Item Image or an Item Update, receiving and sending generic messages). The OMM Provider Event Source is used by Interactive Provider applications and Non-Interactive Provider applications. More than one OMM Provider can be associated with any given Session.

A provider application typically registers an **OMMProvider** Event Source to the following Interest Specifications:

- `omm.OMMClientSessionIntSpec`
- `omm.OMMListenerIntSpec`
- `omm.OMMItemIntSpec` (non-interactive provider)
- `omm.OMMErrorIntSpec`

**TimerIntSpec**

- `omm.OMMConnectionStatsIntSpec`

The **OMMProvider** Event Source supports provider functionality by providing the methods shown in the table below:

FUNCTION NAME	DESCRIPTION
<code>generateToken()</code>	This method is used by a non-interactive provider to generate a Token object. The Token is used to open an event stream. The non-Interactive provider opens an event stream by submitting an <b>OMMItemCmd</b> type command. This command encapsulates an <b>unsolicited refresh message</b> and also must have a Token. The subsequent update response messages submitted on this event stream must include this token.  The Token is valid until the provider is destroyed, the session is released, the Login stream for this provider is closed, or the application submits a final message using this Token.
<code>getSession()</code>	This method returns the session instance that owns this provider.
<code>registerClient()</code>	A provider application uses this method to register a client to receive specific events. This method takes the following arguments: <b>EventQueue</b> , <b>InterestSpec</b> , <b>Client</b> , <b>Object</b> (closure). A registering client will receive events of the specified <b>InterestSpec</b> type from the specified <b>EventQueue</b> . The <b>InterestSpec</b> can be any of the following, based on the type of provider application: <ul style="list-style-type: none"> <li>• Interactive providers can register to <b>OMMClientSessionIntSpec</b> to receive <b>OMMActiveClientSessionEvent</b>, <b>OMMSolicitedItemEvent</b>, and at the end of the client session, <b>OMMInactiveClientSessionEvent</b> events. <b>OMMSolicitedItemEvents</b> encapsulate a response or generic type message.</li> <li>• Interactive providers can register to <b>OMMListenerIntSpec</b> to receive <b>OMMListenerEvents</b>.</li> <li>• Non-interactive providers can register to <b>OMMItemIntSpec</b> to receive the <b>OMMItemEvent</b>, which encapsulates a response type message.</li> <li>• A provider application can register to <b>TimerIntSpec</b> to receive timer-specific events.</li> <li>• A provider application can register to <b>OMMErrorIntSpec</b> to receive the <b>OMMCmdErrorEvent</b>.</li> <li>• <b>OMMConnectionStatsIntSpec</b> is associated with a client. If a client registers with this Interest Specification, it may receive events of <b>OMMConnectionStatsEvent</b> type which allows the client to receive connection statistics (i.e., bytes on the wire).</li> </ul> This method returns a Handle instance identifying the client registration.  When registering, an application may optionally use a closure object. This object is application-defined and is provided by the interface for the user-defined purpose. The closure object specified in the request will be included in the associated response messages. The application can use it as a reference.
<code>submit()</code>	This interface provides an application the ability to send refresh messages, update messages, and generic messages on an open stream and takes the arguments: <b>OMMCmd</b> and <b>Object</b> (closure). The <b>OMMCmd</b>

FUNCTION NAME	DESCRIPTION
	<p>parameter is an instance of <code>OMMItemCmd</code>. The <code>OMMItemCmd</code> encapsulates the refresh messages, update messages, and generic messages and the token associated with the open event stream.</p> <p>An application may optionally specify a closure Object on the <code>submit()</code> call. This Object is application-defined and provided for the purpose specified by the user. The submit closure object will be included in <code>OMMCmdErrorEvent</code> events and can be retrieved by the <code>getSubmitClosure()</code> method.</p> <p>An application should specify an open event stream when calling the <code>OMMProvider submit()</code> method. If the stream is not open, the message will be dropped and the application will be notified through the <code>OMMCmdErrorEvent</code> (if the consumer had registered for an <code>OMMErrorIntSpec</code>).</p> <hr/> <p><b>NOTE:</b> <code>OMMCmdErrorEvents</code> are error notifications delivered to an application if RFA has issues processing the <code>submit()</code> call.</p> <hr/> <p>To get error notifications, an application has to register an <code>OMMErrorIntSpec</code> Interest Specification using the <code>registerClient()</code> method. Optionally, the application can specify a closure object in the <code>registerClient()</code>. This closure object is returned in the <code>OMMCmdErrorEvent</code> and can be retrieved using the <code>getClosure()</code> method.</p>
unregisterClient()	<p>Applications use this method to close open streams.</p> <p>This method takes a handle instance as an argument. The Handle identifies the event stream to be closed.</p>

**Table 35: OMMProvider Event Source Interface Methods**

The following example shows how the consumer application uses an OMMProvider Event Source and OMMListenerIntSpec (refer to Section 8.5) to register and unregister a client. The Event Source is associated with an event queue.

```
// EventSource _ommProvider;

Client client = new ConcreteClient();           // application defined

// create the event queue
EventQueue _eventQueue = EventQueue.create("Provider Application EventQueue");

// create a new listener interest specification
OMMListenerIntSpec ommListenerIntSpec = new OMMListenerIntSpec();
String connectionName = "provConnection";      // a listening connection name
ommListenerIntSpec.setListenerName(connectionName); //

// register the listening client
Handle handle = _ommProvider.registerClient(_eventQueue, ommListenerIntSpec, client, null);

// unregister the client
_ommProvider.unregisterClient(handle);
```

**Example 17: Using OMMProvider Interface**

## 8.4 OMM Command

The `OMMcmd` interface supports an application sending messages on an open event stream. This interface is used by an `OMMConsumer` and an `OMMProvider` Event Source. The Event Source uses the `OMMcmd` as an argument to the `submit()` method. Each command instance that implements `OMMcmd` interface provides the command's specific data. The `OMMcmd` interface provides one method `getCmdType()` that returns the type of the command. The interface also defines the following command types, which are defined in the sections that follow:

- `OMM_HANDLE_ITEM_CMD`
- `OMM_INACTIVE_CLIENT_SESSION_CMD`
- `OMM_SOLICITED_ITEM_CMD`

For each of the types above, the Session package provides a class implementation.

### 8.4.1 OMMHandleItemCmd

The `OMMHandleItemCmd` command encapsulates a message and a Handle. The handle indicates the open event stream on which the message will be sent. The message is sent via the `OMMConsumer` `submit()` method.

This Command is used by an `OMMConsumer` Event Source to send generic or post-type messages.

#### 8.4.1.1 OMMHandleItemCmd Methods

The following table presents the additional methods supported by `OMMHandleItemCmd` Command class:

FUNCTION NAME	DESCRIPTION
<code>getHandle()</code>	This method returns the Handle associated with this Command.
<code>getMsg()</code>	This method returns the OMM message associated with this Command.
<code>setHandle()</code>	This method sets the Handle associated with this Command. The handle represents an open event stream.
<code>setMsg()</code>	This method sets the message associated with this Command.

**Table 36: OMMHandleItemCmd Methods**

#### 8.4.1.2 OMMHandleItemCmd Example of Use

The following code example shows how the application uses `OMMHandleItemCmd` and Event Source to submit a message on an open stream:

```
// EventSource _ommConsumerOrProvider;
Handle _openStreamHandle;

OMMMsg msg = generateStreamChangeMsg(); // application method

// Create a new OMMHandleItemCmd
OMMHandleItemCmd cmd = new OMMHandleItemCmd();
cmd.setMsg(msg);
cmd.setHandle(_openStreamHandle);

_ommConsumerOrProvider.submit(cmd, null);
```

**Example 18: Using OMMHandleItemCmd**

## 8.4.2 OMMinactiveClientSessionCmd

The `OMMinactiveClientSessionCmd` command encapsulates a client session Handle. The handle indicates the client session to which inactive session message will be sent. The message is sent via OMMPProvider `submit()` method.

This command is used by a interactive provider application. When the provider application does not accept an OMM Consumer, it uses the `OMMinactiveClientSessionCmd` command to convey the rejection message. The command encapsulates the client session handle. The application submits this command via OMMPProvider Event Source `submit()` method, which sends a message, indicating inactive session, to the rejected OMM Consumer.

The following table presents the additional methods supported by `OMMinactiveClientSessionCmd` Command class:

FUNCTION NAME	DESCRIPTION
<code>getClientSessionHandle()</code>	This method returns a client session Handle associated with this Command.
<code>setClientSessionHandle()</code>	This method sets the client session Handle associated with this Command. The handle represents an open event stream.

**Table 37: OMMinactiveClientSessionCmd Methods**

The following example shows how a provider application uses `OMMinactiveClientSessionCmd` and OMMPProvider Event Source to reject a client session:

```
// EventSource _ommProvider;
Handle handle; // the handle is included in the incoming event

// Create a new OMMinactiveClientSessionCmd
OMMinactiveClientSessionCmd cmd = new OMMinactiveClientSessionCmd();
cmd.setClientSessionHandle(handle);

_ommProvider.submit(cmd, null);
```

**Example 19: Using OMMinactiveClientSessionCmd**

## 8.4.3 OMMItemCmd

The `OMMItemCmd` command encapsulates a message and a Token. The Token indicates the open event stream on which the message will be sent. The message is sent via the OMMPProvider `submit()` method.

This command is used by provider applications to send refresh responses, update responses, or generic type messages.

The following table presents the additional methods supported by `OMMItemCmd` Command class:

FUNCTION NAME	DESCRIPTION
<code>getMsg()</code>	This method returns an OMM message associated with this Command.
<code>getToken()</code>	This method returns a Token associated with this Command.
<code>setMsg()</code>	This method sets the message associated with this Command.
<code>setToken()</code>	This method sets the Token associated with this Command. The Token represents an open event stream.

**Table 38: OMMItemCmd Methods**

The following code example shows how a provider application uses `OMMItemCmd` and `OMMProvider` Event Source to submit a message on an open event stream:

```
// EventSource _ommProvider;
Token token; // token associated with the open event stream

OMMMsg msg = generateMsg(); // application method

// Create a new OMMItemCmd
OMMItemCmd cmd = new OMMItemCmd();
cmd.setMsg(msg);
cmd.setToken(token);

_ommProvider.submit(cmd, null);
```

### Example 20: Using OMMItemCmd

## 8.5 Interest Specifications

All of the Interest Specifications implement the `InterestSpec` interface from the Common package, and thus provide an implementation of the following methods: `clone()` and `getInterestSpecType()`.

The following table presents the Interest Specification classes supported by the Session package. The second column shows additional methods that are implemented by the specific interest specification:

INTEREST SPECIFICATION	METHODS	DESCRIPTION
omm.OMMClientSessionIntSpec	getClientSessionHandle() setClientSessionHandle()	The provider application uses this interest specification to register its interest in client sessions. When registered to this interest specification, the provider application can receive the following events: <ul style="list-style-type: none"> <li><b>OMMActiveClientSessionEvent</b>: received when the consumer application requests a login. The provider can receive multiple such events and register multiple client sessions.</li> <li><b>OMMSolicitedItemEvent</b>: received if the provider application accepted the session.</li> <li><b>OMMInactiveClientSessionEvent</b>: indicates client session.inactivity</li> </ul>
omm.OMMErrorIntSpec		Used by consumer and provider applications - this Interest Specification is used to receive the <b>OMMCmdErrorEvent</b> event (generated when a command submission error occurs) The consumer application can register an <b>OMMErrorIntSpec</b> after sending a Login request message. The provider application can register this interest specification after registering an <b>OMMListenerIntSpec</b> .
omm.OMMConnectionIntSpec		Used by consumer and non-interactive applications, this Interest Specification is used to register to receive <b>OMMConnectionEvent</b> events.
omm.OMMConnectionStatsIntSpec		Used by RFA applications, this interest specification registers to receive <b>OMMConnectionStatsEvent</b> events.
omm.OMMItemIntSpec	getMsg() setMsg()	Used by consumer and non-interactive applications, this Interest Specification is used to register to receive <b>OMMItemEvent</b> events. After the application registers a request message with the <b>OMMItemIntSpec</b> , and opens event stream, it can receive

INTEREST SPECIFICATION	METHODS	DESCRIPTION
		<b>OMMItemEvent</b> events encapsulating response type messages or generic type messages.
omm.OMMListenerIntSpec	getListenerName() setListenerName()	A provider application uses this interest specification to register with the OMM Provider to open a listening port for consuming client sessions to connect. The provider application is notified of the listening port setup status via an <b>OMMListenerEvent</b> .
TimerIntSpec	getDelay() setDelay() isRepeating() setRepeating()	Typically used by a provider applications using timers to provide periodic update responses to the clients that issued streaming requests.
legacy		Interest Specifications used by legacy implementations.

Table 39: Session Interest Specifications

## 8.6 Event Interfaces and Statuses

The event interfaces, such as `ConnectionEvent`, `OMMCmdErrorEvent`, etc., implement the Event interface from the Common package. In addition to supporting the basic Event functions, each of the interfaces implements specific methods related to the type of the event, and contains event specific definitions.

Some events contain a specific status. A status may contain different information for different types of events. The package defines several status-specific classes that inherit from the Status class defined in the Common package. For example, the connection event will contain a `ConnectionStatus` instance.

The following table summarizes the Event interfaces and related Status definition classes supported by the Session layer:

EVENT INTERFACE	STATUS	DESCRIPTION
event.ConnectionEvent	event.ConnectionStatus	Delivers a connection status change. For details, refer to Section 8.6.1.
omm.OMMCmdErrorEvent	omm.OMMErrorStatus	Delivers status when errors occur during command submission. For details, refer to Section 8.6.2.
omm.OMMActiveClientSessionEvent		Indicates a client session connection event. For details, refer to Section 8.6.3.
omm.OMMInactiveClientSessionEvent()		Indicates a client session disconnection event. For details, refer to Section 8.6.4.
omm.OMMItemEvent		Wrapper for response or generic message. For details, refer to Section 8.6.4.
omm.OMMListenerEvent	Internally defined ListenerStatus	Indicates status on opening of a listening port. For details, refer to Section 8.6.6.
omm.OMMSolicitedItemEvent		Wrapper for request or generic messages. For details, refer to Section 8.6.7.
omm.OMMConnectionEvent	event.ConnectionStatus	Delivers OMM connection information (i.e., connection status, active host name, port, version information, and any multicast connection information for the connected component). For details, refer to Section 8.6.8.

EVENT INTERFACE	STATUS	DESCRIPTION
omm.OMMConnectionStatsEvent		Contains information on connection statistics (bytes on the wire).

**Table 40: Session Events**

## 8.6.1 Connection Event

The ConnectionEvent interface provides users with access to the connection name, type and status of a connection. The name and type of the connection does not change during its lifetime. The change of connection status is delivered to the interested clients as an event. An application can register its interest in notifications of changes in connection statuses. As soon as it registers its interest, it will receive information about existing connections. Then every time a connection status changes, the application will be notified.

The ConnectionEvent interface defines the following connectionTypes:

- RSSL: Used by consumer-type applications
- RSSL\_PROV: Used by interactive provider-type applications
- RSSL\_PROV\_CLIENT: Used by non-interactive provider-type applications
- Other legacy types

A connection name is a String representing the name as it was configured.

A connection status is defined by ConnectionStatus final class. This class specifies the following connection specific states and status codes:

States:

- UP
- DOWN

Status Codes:

- ACCESS\_DENIED
- CLOSED
- INTERMITTENT\_PROBLEMS

An instance of the ConnectionEvent contains the ConnectionStatus with a state, and if the state is DOWN, it will also contain status code. The status may contain text further describing the event.

## 8.6.2 OMM Command Error Event

The provider application can receive error notifications related to a `submit()` call via an `OMMCmdErrorEvent` event. This event occurs when a `submit()` call fails.

This event gives access to the command, command ID, error closure, submit closure, and status for the command that failed. You can access this data via `getCmd()`, `getCmdID()`, `getClosure()`, `getSubmitClosure()`, and `getStatus()` respectively.

A command status is defined by `OMMErrorStatus` final class, which specifies the following states and status codes:

States:

- FAILURE

Status Codes:

- NO\_RESOURCES

The status may contain text further describing the event.

## 8.6.3 OMM Active Client Session Event

The `OMMActiveClientSessionEvent` interface supports activation of client session events.

The provider application receives `OMMActiveClientSessionEvent` events when the client session connects. The client session can be accepted or rejected by the provider application.

This interface provides access to the host name, IP address, component version information, and session handle of the client requesting the connection. These can be accessed via `getClientHostName()`, `getClientIPAddress()`, `getClientComponentVersion()`, and `getClientSessionHandle()` respectively. This interface also provides access to the listener name, as the configured connection name via the `getListenerName()` method.

## 8.6.4 OMM Inactive Client Session Event

The `OMMActiveClientSessionEvent` interface supports deactivation of client session events.

The provider application receives client session disconnect notifications via the `OMMInactiveClientSessionEvent` event. This event indicates that the client session is inactive. All tokens previously received for this client become inactive.

Use this interface to access the client host name, client IP address, and client component version information via `getClientHostName()`, `getClientIPAddress()` and `getClientComponentVersion()` respectively. This interface also provides access to the listener name as the configured connection name via the `getListenerName()` method.

## 8.6.5 OMM Item Event

The `OMMItemEvent` interface encapsulates the response and generic types of messages. This type of event is received by a consumer-type application when a response is available for the request the application registered. It can be also received by consumer or provider applications when a generic message is available on the event stream opened by the application.

This interface implements the `getMessage()` method, which returns the encapsulated OMM message.

## 8.6.6 OMM Listener Event

The `OMMListenerEvent` interface supports listening-port events. This event is delivered to a provider-type application when RFA attempts to open a listening port. It can indicate either successful or unsuccessful port openings. This interface provides access to the listener name and status via the `getListenerName()` and `getStatus()` methods respectively.

The status is defined within this interface. `OMMListenerEvent` specifies the following states and status codes:

- States: SUCCESS or FAILED
- Status Code: WILL\_RETRY (The status may contain additional text further describing the event).

## 8.6.7 OMM Solicited Item Event

The `OMMSolicitedItemEvent` interface encapsulates request, generic, or post types of messages. This type of event is received by a provider application when a request message or post message is received. It can be also received by consumer or provider applications when a generic message is available on the event stream opened for a previous request.

This interface implements the `getMessage()` method, which returns the encapsulated OMM message, and the `getRequestToken()` method, which returns the token associated with the message.

## 8.6.8 OMM Connection Event

The `OMMConnectionEvent` interface provides information about the consumer's or non-interactive provider's hostname, port, and component version information. `OMMConnectionEvent` implements the following methods:

METHOD	DESCRIPTION
<code>getConnectionStatus()</code>	Returns the status of the active connection.
<code>getConnectedHostName()</code>	Returns the connected host name for the active connection.
<code>getConnectedPort()</code>	Returns the connected port for the active connection.
<code>getConnectedComponentVersion()</code>	Returns the connected client component version for all connected connections.
<code>getConnectedRecvAddress()</code>	Returns the connected address for all receiving connections.
<code>getConnectedRecvPort()</code>	Returns the connected port for all receiving connections.
<code>getConnectedSendAddress()</code>	Returns the connected address for all sending connections.
<code>getConnectedSendPort()</code>	Returns the connected port for all sending connections.
<code>getConnectedUnicastPort()</code>	Returns the connected port for all unicast connections.
<code>getConnectionName()</code>	Returns the name of the connection on which this event occurred.
<code>getBytesWritten()</code>	Returns the number of actual bytes written to the wire.
<code>getBytesRead()</code>	Returns the number of actual bytes read from the wire.

**Table 41: OMMConnectionEvent Methods**

## 8.6.9 OMM Connection Statistic Event

The `OMMConnectionStatsEvent` interface provides information about the number of bytes read and written on the wire for a connection. `OMMConnectionStatsEvent` implements the following methods:

METHOD	DESCRIPTION
<code>getConnectionName()</code>	Returns the name of the connection on which the event occurred.
<code>getBytesWritten()</code>	Returns the actual number of bytes written on the wire.
<code>getBytesRead()</code>	Returns the actual number of bytes read from the wire.

**Table 42: OMMConnectionStatsEvent Methods**

## Chapter 9 Config Package

### 9.1 Config Package Overview

RFA configuration provides access to and management of a hierarchical, in-memory configuration data storage. RFA can be configured utilizing either Java Preferences API or utilizing RFA Config package.

The RFA Java Edition kit provides configuration tools that can be utilized to create xml files containing configuration data. The tool generates the xml file in the predefined location. The configuration data is accessed utilizing the Java Preferences API. The xml file can be also created manually. For details on the Preferences API, refer to 4.

The configuration database can be populated programmatically by utilizing the Config package. For details on the Config package, refer to Section 9.33.

For the Configuration Package overview, refer to Section 5.10.4.

### 9.2 Configuration Database

#### 9.2.1 Configuration Concepts

Configuration of RFA Java Edition introduces several concepts related to populating and querying configuration information. The following table defines these concepts.

CONCEPT	DESCRIPTION
Config Database	In-memory hierarchal configuration storage that is set up by the Preferences API or by RFA Config package interface and is populated by the user.
Config Node	A node is a hierarchical collection of configuration data. Each node is named in a similar fashion to directories in a hierarchical system. It follows the Java Preferences node naming scheme (see reference 4).
Config Subtree	A specific concrete type of Config Node that typically contains a set of child Config Nodes. RFA Java Edition first searches the user tree and then the system tree (see Section 9.2.2.1).
Namespace	A name chosen to correspond to a Config Subtree (called Namespace Tree) under the Config Root. Namespaces are used to logically group sets of configuration hierarchies in a Config Database (see Section 9.2.2.3).
Config Root	The uppermost Config Node in a Config Database in which RFA Java stores all its configuration data. It must be: <b>"/com/reuters/rfa"</b> (see Section 9.2.2.2).
Component Path	The Config Node(s) under the Namespace node and above the Instance Name node (see Section 9.2.2.5).
Instance Name	The Config Node under the Component Path (see Section 9.2.2.6).
Softlink	A specific type of Config Node used to point to another Config Node for configuration sharing using a relative path. The notation is <b>[Namespace:]InstanceName</b> (see Section 9.2.3.2).

**Table 43: Configuration Concepts**

## 9.2.2 Naming Scheme

### 9.2.2.1 Overview

The RFA Java Config Database follows the syntax and naming scheme detailed in the Java Preferences API, in which there are two configuration trees (i.e., the user tree and the system tree). The RFA Java API shields programmers from this differentiation. When retrieving configuration data, the RFA API will always search the user tree first. If the requested information does not exist in the user tree, it will then search the system tree. Since there is no difference with respect to the data hierarchy between the user tree and the system tree, we discuss the RFA Java configuration within a single configuration tree.

The format of a full configuration path to a configuration variable is the following (see following corresponding sections for further details):

```
<ConfigurationRoot><Namespace><ComponentPath><InstanceName><variableKey=variableValue>
```

Also see Figure 32 for examples.

### 9.2.2.2 Configuration Root

The uppermost Config Subtree in the Config Database is the Config Root, which is the root of the RFA Java Config Database and must be `/com/reuters/rfa`.

### 9.2.2.3 Namespace

To allow for component-style application development and prevent name conflicts, RFA Java configuration supports logical grouping of the configuration into separate namespaces. A namespace is an arbitrary string chosen by the development team to relate, for example, to the company name or the product name. Each namespace corresponds to a Config Subtree under the Config Root. This subtree is called a Namespace Tree and has the same name as the namespace. For examples see Figure 32: An Example RFA Configuration Tree.

The RFA software packages fully support user applications built using the component paradigm. The RFA addresses the issues that usually arise when doing component development:

- Unique component identification of disparate component instances and
- Component instance sharing when using RFA components from user-developed components.

Using Namespaces allows for unique component names, reducing the probability of name conflicts. The RFA uses component names to configure them and to control instance sharing.

### 9.2.2.4 Namespace Constraints

RFA Java Edition reserves namespaces with the prefix `_`. The two reserved namespaces are `_Default` and `_System`. The `_Default` namespace stores configuration data that will be returned if RFA cannot locate the information it needs under the user specified namespace. The `_System` namespace stores configuration data that cannot be customized by a user specified (arbitrary) namespace (see Figure 32).

### 9.2.2.5 Component Path

The Component Path is the Config Node under the Namespace node and above the Instance Name node. The value of the Component Path is one from a set of RFA predefined paths and is documented in the 3 *Configuration and Logging Guide*. Examples are Sessions, Connections, etc.

The Component Path may consist of multiple Config Nodes.

### 9.2.2.6 Instance Name

The Instance Name is a Config Node that is under the Component Path. Usually there will be multiple Instance Names under a single Component Path, e.g. “Sessions/Session\_1”, “Sessions/Session\_2”, etc.

There is also a `_Default` instance under the Sessions node, i.e., “Sessions/\_Default”.

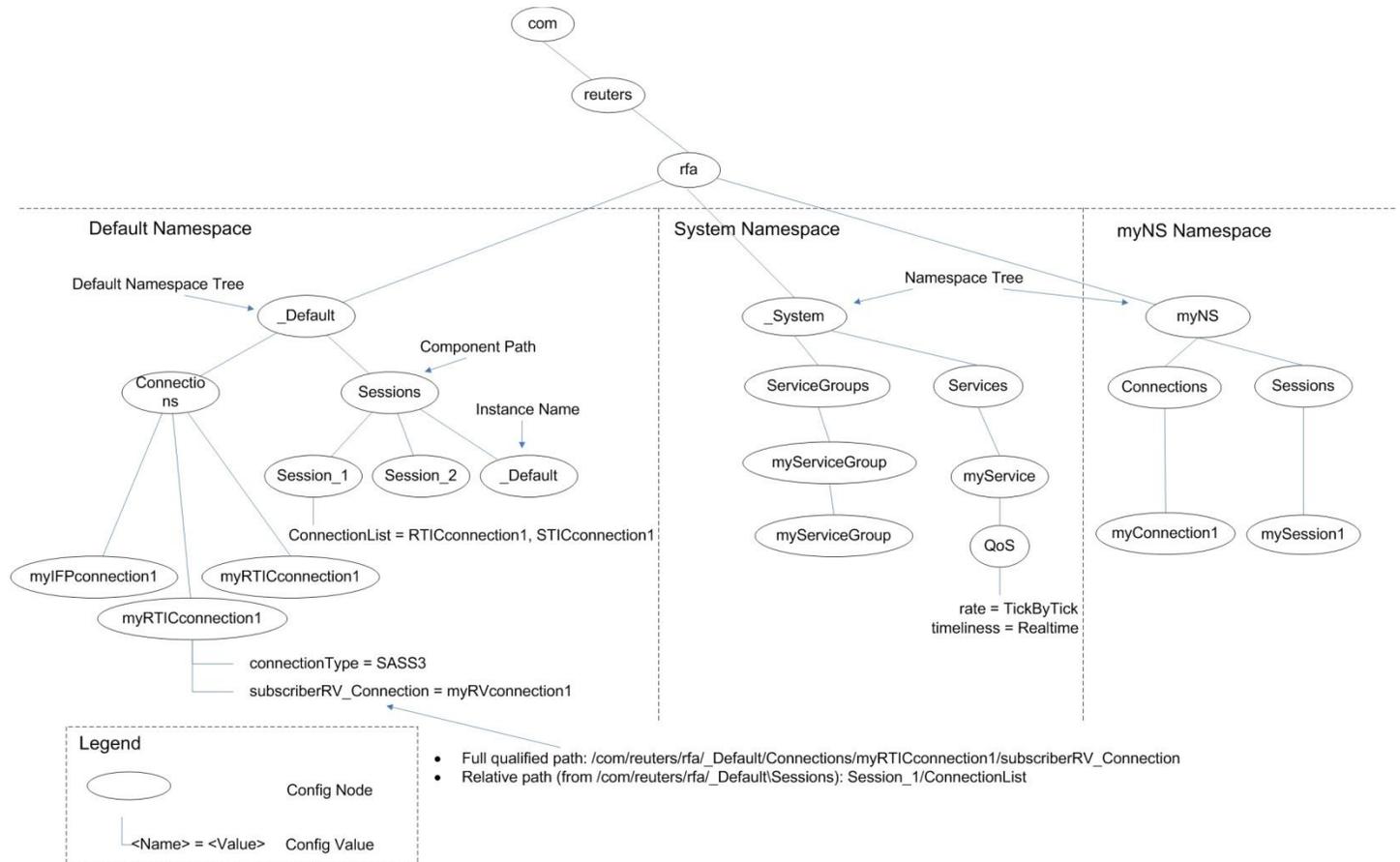


Figure 32: An Example RFA Configuration Tree

## 9.2.3 Configuration Sharing

RFA Java Edition enables components to share their configuration data via a node link mechanism. For example, we may have two instances of sessions created, e.g. “mySession1” and “mySession2”.

### 9.2.3.1 Soft Links

A Soft Link is a link-node that has the following keys (i.e., variable names with their values):

- `type = “SoftLink”`
- `path = [namespace::]<the target node name>`
- `softLinkInstanceSharing = “true”` or “false”

See Figure 33.

### 9.2.3.2 Node Links

A link is a Config Node (link-node) with the reserved node name “**\_Link**” that points to another Config Node (the target node). The link-node must contain two attributes, “**type**” and “**path**”. The value of the “**path**” attribute provides information for reaching the target-node. The value of the “**type**” attribute specifies the syntax and semantic of the “**path**”. There is also an additional optional attribute, “**softLinkInstanceSharing**”, which enables instance sharing of a session or a connection.

There is one type of link, i.e., Softlink. A Soft Link specifies the namespace and instance name of the target node.

A link-node exists under an Instance Name node. If an instance node contains a link, all configuration variables defined under this node will be ignored. A link can point to a target node that contains another link.

### 9.2.3.3 Constraints

If a Config Node has a link-node, it must not contain any Config Subtree or any key.

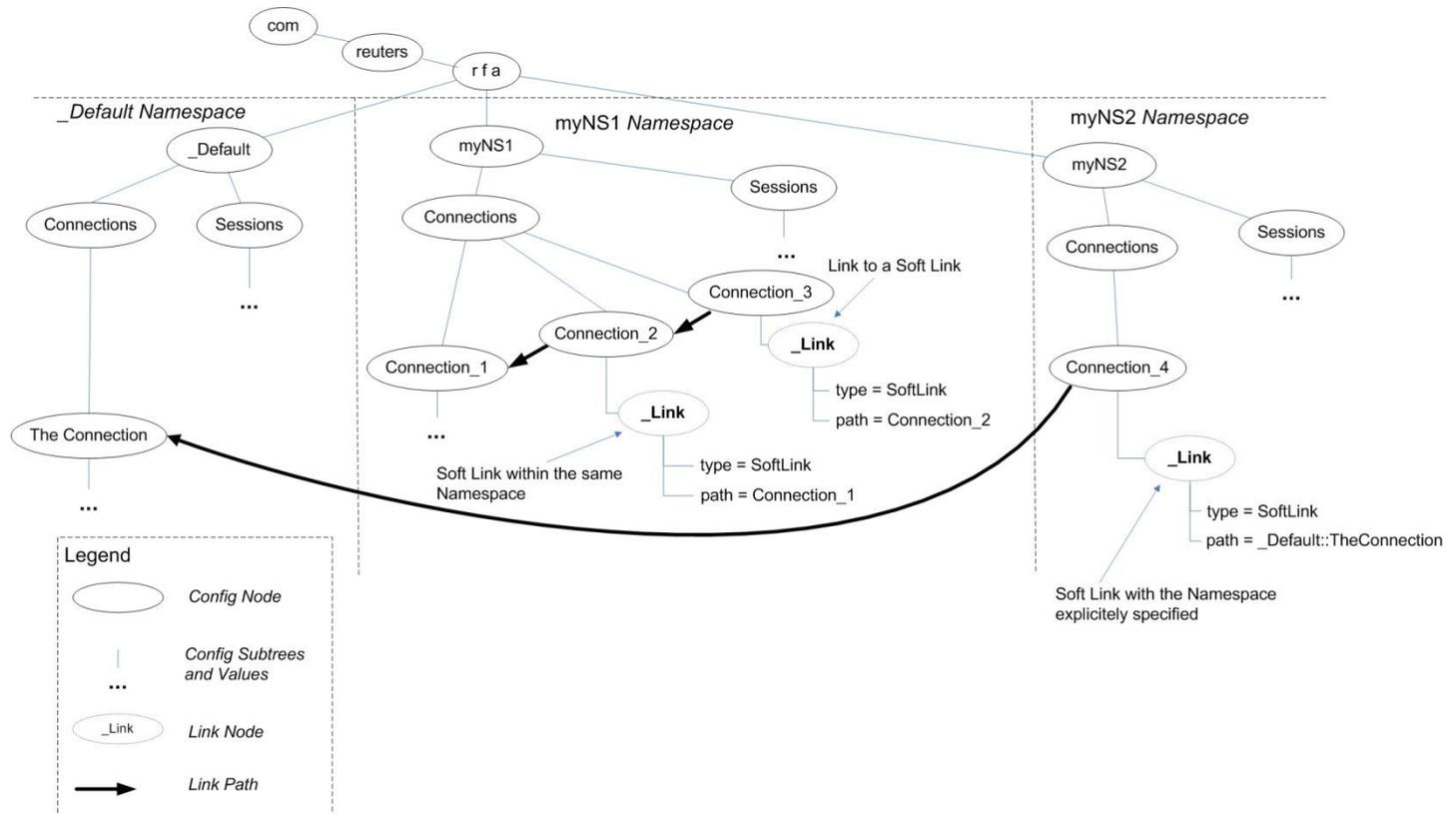


Figure 33: Soft Link Examples

## 9.3 Config Package

The Config package enables developer to configure the RFA programmatically by an application. The configured nodes should follow the convention of the configuration tree.

The Config package provides the `ConfigProvider` interface and the `ConfigDb` class which implements it. The `ConfigDb` class can be instantiated by an application. This class represents the configuration data base.

The table below summarizes methods supported by the `ConfigDb` class.

METHOD NAME	DESCRIPTION
<code>addVariable()</code>	This method adds a parameter into the config data base. It takes three Strings as attributes. The first String defines the path parameter, related to the configuration tree. The next arguments are parameter name and parameter value respectively. There is an overloaded method. This method does not take the second argument.
<code>childrenName()</code>	This method takes a String that defines configuration path. It returns an array of nodes names that are configured in this path. This method is inherited from the <code>ConfigProvider</code> interface.
<code>getValue()</code>	This method returns String representation of the configured value.
<code>toString()</code>	Interface to return the contents of the ConfigDb instance in a String format.
<code>Variable()</code>	Several overloaded methods providing access to the ConfigDb parameters.

**Table 44: ConfigDb Methods**

When configuring utilizing the config package, an application needs to create the `ConfigDb` class and initialize context with this instance. The example below shows the initialization.

```
ConfigDb configDb = new ConfigDb();
initializeConfigDb(configDb);
Context.initialize(configDb);
```

### Example 21: Context Initialized with ConfigDb

The method `initializeConfigDb()` is application implemented and it sets the configuration database. A code example, showing setting of several configuration variables by a consumer application is shown below.

```
void initializeConfigDb(ConfigDb configDb)
{
    configDb.addVariable("aNamespace.Sessions.consSession.connectionList", "consConnection");
    configDb.addVariable("aNamespace.Connections.consConnection.connectionType", "RSSSL");
    configDb.addVariable("aNamespace.Connections.consConnection.serverList", "localhost");
    configDb.addVariable("aNamespace.Connections.consConnection.portNumber", "14002");
}
```

### Example 22: Programming ConfigDb

## 9.4 Configuration Using Java Preferences API

### 9.4.1 Populate Config Database

The user must populate a Config Database in an application by using the Java Preferences API. As shown in the following two examples, Java's `importPreferences` method is used. Similarly Java outputs the Config Database to an XML file by using the `exportSubtree` method.

#### 9.4.1.1 Populating from File

```
try
{
    FileInputStream filestream = new FileInputStream(configFile);
    Preferences.importPreferences(filestream);
    return true;
}
```

#### Example 23: Populating RFA Config Database from File Path

In this example the Config Database is populated from an XML file (using the input file path) via the `importPreferences()` method from the Java Preferences API.

#### 9.4.1.2 Populating from URL

In the following example, the Config Database is populated from an XML file (using the input URL path) via the `importPreferences()` method from the Java Preferences API.

```
try
{
    URL u = new URL(urlString);
    URLConnection uc = u.openConnection();
    uc.setUseCaches(true);
    int len = uc.getContentLength();
    DataInputStream stream = new DataInputStream(uc.getInputStream());
    byte [] data = new byte[len];
    stream.readFully(data);
    ByteArrayInputStream bytestream = new ByteArrayInputStream (data);
    try
    {
        Preferences.importPreferences(bytestream);
        return true;
    }
}
...
```

#### Example 24: Populating RFA Config Database from URL Path

## 9.4.2 Remove RFA Configuration Nodes

The following example shows how we may remove a Config Subtree within the Config Database in the User Preference tree. The example uses the Preferences API `removeNode()` method, which will remove the Config Node corresponding to `pathToRemove` and all its child nodes.

```
Preferences prefs = Preferences.userRoot().node(pathToRemove);
try
{
    prefs.removeNode();
}
...
```

### Example 25: Removing RFA Config Subtree

## 9.4.3 Query RFA Configuration Information: Retrieving and Iterating RFA Configuration Nodes

RFA uses the following rules to query Configuration information.

To retrieve and iterate through Config Nodes, the user will use corresponding methods from the Java Preferences API.

Configuration searches information in the following order:

- `/<myNamespace>/<componentConfigPath>/<myInstanceName>`
- `/<myNamespace>/<componentConfigPath>/_Default`
- `/_Default/<componentConfigPath>/<myInstanceName>`
- `/_Default/<componentConfigPath>/_Default`

SESSION SPECIFIED	CONFIG TREE PATH CHOSEN IN FIGURE 32
myNS::mySession1	/com/reuters/rfa/myNS/Sessions/mySession1
myNS::Session_1	/com/reuters/rfa/_Default/Sessions/Session_1
myNS::Session_3	/com/reuters/rfa/_Default/Sessions/_Default
myNS::	/com/reuters/rfa/_Default/Sessions/_Default
Session_1	/com/reuters/rfa/_Default/Sessions/Session_1
MySession1	/com/reuters/rfa/_Default/Sessions/_Default

**Table 45: Session search examples from Figure 32**

In the above searching order, Configuration will first look for a link, before moving onto the next search step. If a link does exist, it will be followed and thus all configuration defined in this node will be ignored. At the node pointed to by the link, the search order will be repeated again until reaching an Instance Name node that does not have a link.

Examples with links:

CONNECTION SPECIFIED	CONFIG TREE PATH CHOSEN IN FIGURE 33
myNS1/Connections/Connection_3	/com/reuters/rfa/myNS1/Connections/Connection_1
myNS2/Connections/Connection_4	/com/reuters/rfa/_Default/Connections/TheConnection

**Table 46: Session search examples from Figure 33**

## 9.5 Configuration Tools

Tools are provided to help with the management of the Configuration Database:

- **config\_editor**: A GUI editor tool for editing the Config Database and importing or exporting it to an XML file.
- **config\_loader**: A command line utility for populating the Config Database from an XML file, through its full path or through a URL.
- **config\_remover**: A command line utility for removing any Config Subtree.

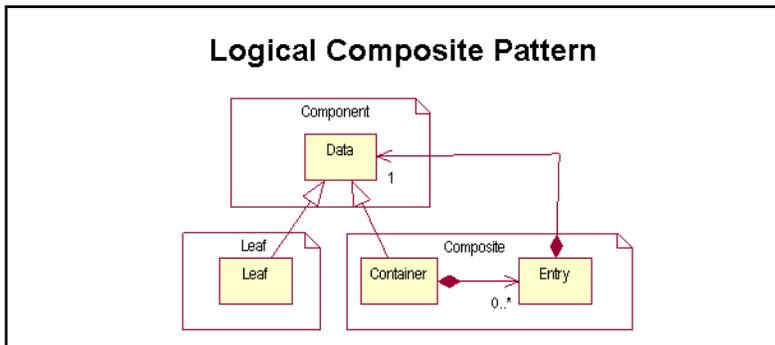
# Chapter 10 OMM Data

## 10.1 Overview

For details on the OMM Data constructs concept, refer to Section 5.7.

OMM Data is classified as *primitive data* or *dataformat*. Primitive data is typically a simple data type, such as an integer or a date, but also includes the simple container Array. Dataformats are typically more complex data types, such as Map and ANSI Page. OMM data is supported through the `OMMData` interface and several interfaces supporting the OMM containers that implement the `OMMData` interface.

RFA uses a Composite Pattern as shown below.



**Figure 34: Logical Composite System**

The container is a descendant of Data. Each container houses zero or more of the same type of entry. In turn, an entry contains data to form the composite. Similar to a composite pattern, RFA's logical composite can contain data whose descendants are either a composite or leaf. This ability to contain data enables the formation of a comprehensive nesting hierarchy. The leaf component represents a primitive data, where composite represents a dataformat.

The OMM package also defines several complex data types that are not primitive or dataformats. These types do not implement `OMMData` interface.

The raw data content needs to be encoded into dataformats that can be embedded in messages flowing between the consumer and provider applications. The OMM package provides encoding/decoding interfaces to create/parse all containers and entries. **Encoding** is the process of converting raw data into dataformats/entries. The OMM package provides the `OMMEncoder` object to encode dataformats and `set.xxx` methods to encode the leaf. **Decoding** is the inverse of encoding and is the process of obtaining the raw information from an encoded object. Support for decoding is provided through `get.xxx()` methods and read iterators for the containers. A few data types require alternative encoders (e.g. ANSI page, XML, 3rd party proprietary content).

## 10.2 OMM Data Interfaces

In Figure 34, the Leaf represents a primitive data and the Composite represents a dataformat. The primitive and complex OMM data implement the `OMMData` interface. Complex data is organized into a tree-like structure with parent-child relationships, and allows repeating parent-child relationships. Each parent can have many children but, each child has only one parent. The child can be a parent for a sub-tree and it may contain children. The parent is known as the *container*, and the contained child is known as an *entry*. These data types may contain 0...n entries.

The OMM package supports the following containers and entries:

- ElementList / Element Entries
- FieldList / Field Entries
- Vector / Vector Entries
- Map / Map Entries
- Series / Series Entries
- FilterList / Filter Entries
- Array / Array Entries

Each of the dataformats listed above is supported by a unique interface (e.g. the ElementList dataformat is supported through the `OMMElementList` interface). The entries extend the `OMMEntry` interface, except for the Series Entry and Array Entry. The Series Entry and Array Entry are represented by the `OMMEntry` itself. An array can contain only primitives, so the nested hierarchy is limited to one level.

The table below describes methods defined by the `OMMData` interface:

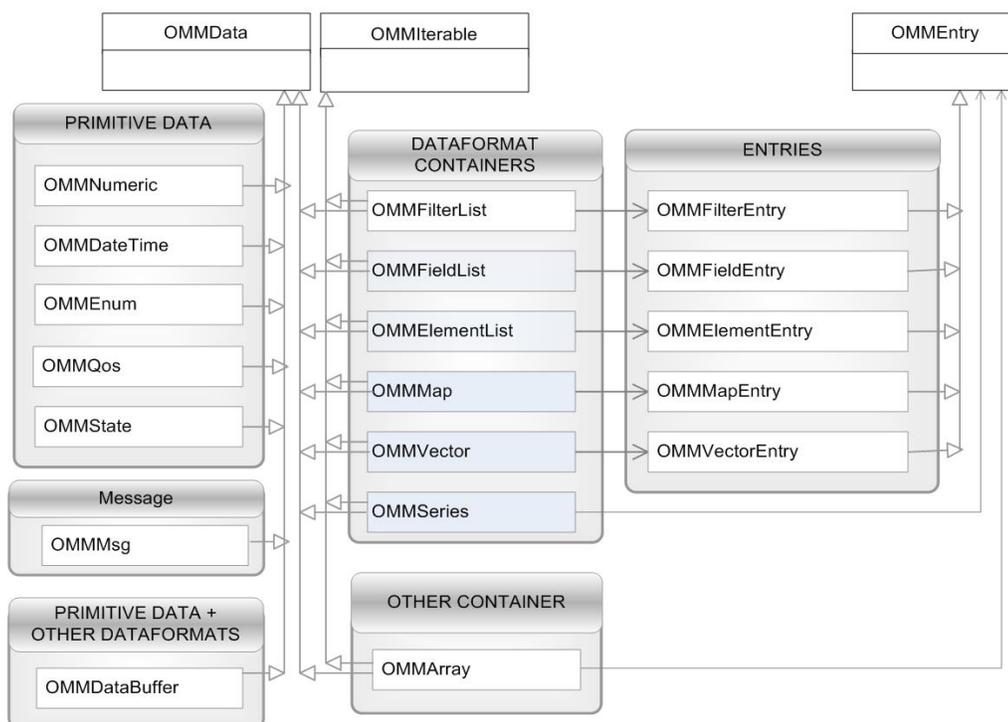
METHOD	DESCRIPTION
<code>getBytes()</code>	This method returns a copy of bytes used to encode this data. It can be only used when decoding the <code>OMMData</code> . If the data is just created, it returns an empty array of bytes. The overloaded method takes a byte array as a parameter and fills it with the encoded data. The second parameter, passed to this method is an int specifying a starting position where the data should be added to the buffer. This method returns the number of bytes copied into the array.
<code>getEncodedLenght()</code>	Returns the length of encoded data without requiring the data to be copied into a buffer. Can be used only for decoding.
<code>getMajorVersion()</code>	Returns the major version that this data is linked to through metada information.
<code>getMinorVersion()</code>	Returns the minor version that this data is linked to through metada information.
<code>getType()</code>	Returns the OMM type of the data, as a short value.
<code>isBlank()</code>	Returns true if the data has no value, i.e., encoded data has an empty buffer.
<code>isPrimitive()</code>	Returns true if the data is primitive.
<code>setAssociatedMetalInfo()</code>	This method sets the realated versions on this object.

**Table 47: OMMData Interface**

## 10.2.1 Interfaces Hierarchy

The classes and interfaces are designed solely for encoding and decoding purposes. They are not optimized for caching with respect to object size or data lifecycle. Objects retrieved from an event typically last only as long as the event. The values stored by these objects should be extracted and copied, if necessary, before storing in a cache (e.g. Java Collection Classes).

The following diagram presents a high level view of the data interfaces supported by the OMM package. Data is supported through the generic `OMMData` interface and is an abstraction for both leaf (primitive data) and containers (dataformats) described in Sections 10.3 and 10.4.



**Figure 35: Data Interfaces in OMM Package**

The primitive data is represented by primitive interfaces. Thus numerics are represented by `OMMNumeric`, date by `OMMDateTime`, etc. The primitive interfaces provides methods to obtain the data content, e.g. the long value represented by `OMMNumeric` is obtained using the `toLong()` method.

The `OMMDataBuffer` interface can be used for primitive strings as well as dataformats requiring external parsers, e.g. ANSI page and XML.

The dataformats inherit from both `OMMData` and `OMMIterable`. The `OMMIterable` object provides support for traversing the entries. The dataformats contain entries. The entries inherit from `OMMEntry`. The dataformats Series and Array contain the generic entry `OMMEntry`. The rest of the dataformats contain specific kinds of entries, e.g. the dataformat `OMMFilterList` contains entries of kind `OMMFilterEntry`.

The container `OMMArray` is a primitive since it can contain only primitives and not nested dataformats.

## 10.2.2 Data Dictionary Utilities

The OMM package provides utilities to parse, cache and access field, enumeration and data definition dictionaries. They are available in the `com.reuters.rfa.dictionary` package. See the *Java Reference Manual* for details.

## 10.3 Primitive Data

The primitive data interfaces inherit from `OMMData` and are used to represent the primitive data types. The primitive data types are defined in `OMMTypes`. The following table lists the primitive types and the corresponding interfaces.

TYPE	INTERFACE	DESCRIPTION
Length Specified Numeric Types	<code>OMMNumeric</code>	Primitive data type used as an <code>OMMData</code> adapter for numeric values. Examples include: <code>INT</code> , <code>INT_1</code> , <code>UINT</code> <code>OMMTypes</code> . For details, refer to Section 10.3.1.
String Types Buffer Types Dataformat requiring external parsers, e.g. ANSI page, XML	<code>OMMDataBuffer</code>	Primitive data type used as an <code>OMMData</code> adapter for strings and buffers. For details, refer to Section 10.3.2.2.
Length Specified Date Type Length Specified Time Type	<code>OMMDateTime</code>	Primitive data type used as an <code>OMMData</code> adapter for <code>java.util.Calendar</code> data. For details, refer to Section 10.3.4.

TYPE	INTERFACE	DESCRIPTION
Length Specified DateTime Type Defined Data Date Type Defined Data Time Type Defined Data DateTime Type		Represents the date (day, month, year), time (hours, minutes, seconds, milliseconds) or a combination of thereof.
Quality of Service	OMMQos	Primitive data type used as an OMMData adapter for QualityofService data. For details, refer to Section 10.3.5.
State	OMMState	State Information used for message, service and item group. For details, refer to Section 10.3.6.
Enumeration	OMMEnum	Numeric values that can be used to look up an expanded string value in an enumeration dictionary. For details, refer to Section 10.3.7.

Table 48: Primitive Data Interfaces

### 10.3.1 Standard and Defined Data Types

**Standard data** is data that unites the data content and its type in an entry. It is represented as one entity. Standard data is self-describing data, such as INT, REAL, etc.

**Defined data** is the data content without its type and requires data definitions. The defined primitive data types are similar to the non-defined primitive types, with some differences. While the non-defined primitive types data can be encoded as a variable number of bytes, most of the defined primitive types leverage a fixed length encoding. This fixed length encoding can help to reduce the total number of bytes required to contain the encoded primitive type. `OMMDataTypes` interface defines values between **64** and **127** as defined primitive types. These define fixed length encodings for many of the primitive types (e.g. `INT_1` is a one byte fixed length encoding of `INT`). The defined primitive data is used in data definitions.

### 10.3.2 OMMNumeric

Several OMM types implement the `OMMNumeric` interface. This interface supports conversion methods (to/from various numeric types). It also provides methods that allow hit values to be specified as-necessary (for specific types of numeric conversions). An `OMMNumeric` value is encoded or decoded utilizing this interface. The interface allows a Java type numeric value to be converted into `OMMNumeric` object of a specific type, and also convert in reverse order.

The code below demonstrates the usage of the `OMMNumeric` interface for encoding and decoding numeric values. Later chapters will describe more details about the encoding and decoding. This example intends to illustrate how the Java numeric value combined with `OMMTypes` is encoded into `OMMData` and how the `OMMData` is converted into Java numeric value, using `OMMNumeric` interface.

```
// encoding a numeric value
Short type = OMMType.UINT;
Long value = 10L;
_encoder.encodeUInt(value); // encoder converts to OMMData
OMMData convertedData = (OMMData)_encoder.acquireEncodedObject();

// decoding numeric value
OMMData data = acquireEncodedData() // method that returns decoded data
short type = data.getType();
switch (type){
    case OMMTypes.UINT:
        long value = (OMMNumeric)data.toLong();
        System.out.println("Decoded value: " + value); // prints: Decoded value: 10
```

#### Example 26: Using OMMNumeric Interface

The `OMMNumeric` interface defines hint flags that apply to the real values only. The flags specify how the value contained in the `OMMNumeric` should be recalculated to obtain the real value during decoding. For example if the numeric value is encoded with the `OMMNumeric.EXPONENT_NEG4` hint flag, the concluded numeric value should be multiplied by 10 to the power 4. Refer to Sections 10.3.2.9 and 10.3.2.10 for encoding and decoding examples of real numbers using hint flags.

### 10.3.2.1 OMMNumeric Types

The following table lists the OMMNumeric types and the encoding methods supported by OMMEncoder. The string representation of the primitive data, if applicable, is encoded using the `encodeString()` method.

OMMTYPES	DESCRIPTION	ENCODING METHODS
INT,	An integer value.	<code>encodeInt ()</code> <code>encodeString ()</code>
INT_1, INT_2, INT_4, INT_8	Respectively, a one byte integer value, 2-bytes integer, 4-bytes integer, 8-bytes integer. These are defined data types. Refer to Section 10.3.1 for details on defined data.	<code>encodeInt ()</code> <code>encodeString ()</code>
UINT	An unsigned integer value.	<code>encodeUInt ()</code> <code>encodeString ()</code>
UINT_1, UINT_2, UINT_4, UINT_8	Respectively, a one byte unsigned integer value, 2-bytes unsigned integer, 4-bytes unsigned integer, 8-byte unsigned integer. These are defined data types. Refer to Section 10.3.1 for details on defined data.	<code>encodeUInt ()</code> <code>encodeString ()</code>
FLOAT	A float value.	<code>encodeFloat ()</code> <code>encodeString ()</code>
FLOAT_4	A 8-bytes float. This is a defined data type. Refer to Section 10.3.1 for details on defined data.	<code>encodeFloat ()</code> <code>encodeString ()</code>
DOUBLE	A double value.	<code>encodeDouble ()</code> <code>encodeString ()</code>
DOUBLE_8	A 8-bytes double. This is a defined data type. Refer to Section 10.3.1 for details on defined data.	<code>encodeDouble ()</code> <code>encodeString ()</code>
REAL	A real value.	<code>encodeReal ()</code> <code>encodeString ()</code>
REAL_4RB, REAL_8RB	Respectively, 4-bytes real, 8-bytes real. These are defined data types. Refer to Section 10.3.1 for details on defined data.	<code>encodeReal ()</code> <code>encodeString ()</code>

**Table 49: OMMNumeric OMMDataTypes**

### 10.3.2.2 OMMNumeric Hint Flags

The following table lists the hint flags supported by the OMMNumeric interface. The hint flags apply to the REAL, REAL\_4RB, and REAL\_8RB OMMNumeric types only.

HINT FLAG	DESCRIPTION
BLANK_REAL	
DIVISOR_1	Fractional denominator operation, equivalent to 1/1. The value does not change.
DIVISOR_2	Fractional denominator operation, equivalent to 1/2. Depending on the type of conversion, this adds or removes a denominator of two.
DIVISOR_4	Fractional denominator operation, equivalent to 1/4. Depending on the type of conversion, this adds or removes a denominator of four.

HINT FLAG	DESCRIPTION
DIVISOR_8	Fractional denominator operation, equivalent to 1/8. Depending on the type of conversion, this adds or removes a denominator of eight.
DIVISOR_16	Fractional denominator operation, equivalent to 1/16. Depending on the type of conversion, this adds or removes a denominator of 16.
DIVISOR_32	Fractional denominator operation, equivalent to 1/32. Depending on the type of conversion, this adds or removes a denominator of 32.
DIVISOR_64	Fractional denominator operation, equivalent to 1/64. Depending on the type of conversion, this adds or removes a denominator of 64.
DIVISOR_128	Fractional denominator operation, equivalent to 1/128. Depending on the type of conversion, this adds or removes a denominator of 128.
DIVISOR_256	Fractional denominator operation, equivalent to 1/256. Depending on the type of conversion, this adds or removes a denominator of 256.
EXPONENT_0	Exponent operation, equivalent to $10^0$ . The value does not change.
EXPONENT_NEG1	Negative exponent operation, equivalent to $10^{-1}$ . Shifts decimal by one position, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG2	Negative exponent operation, equivalent to $10^{-2}$ . Shifts decimal by two positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG3	Negative exponent operation, equivalent to $10^{-3}$ . Shifts decimal by three positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG4	Negative exponent operation, equivalent to $10^{-4}$ . Shifts decimal by four positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG5	Negative exponent operation, equivalent to $10^{-5}$ . Shifts decimal by five positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG6	Negative exponent operation, equivalent to $10^{-6}$ . Shifts decimal by six positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG7	Negative exponent operation, equivalent to $10^{-7}$ . Shifts decimal by seven positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG8	Negative exponent operation, equivalent to $10^{-8}$ . Shifts decimal by eight positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG9	Negative exponent operation, equivalent to $10^{-9}$ . Shifts decimal by nine positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG10	Negative exponent operation, equivalent to $10^{-10}$ . Shifts decimal by ten positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG11	Negative exponent operation, equivalent to $10^{-11}$ . Shifts decimal by 11 positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG12	Negative exponent operation, equivalent to $10^{-12}$ . Shifts decimal by 12 positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG13	Negative exponent operation, equivalent to $10^{-13}$ . Shifts decimal by 13 positions, depending on the type of conversion, to the right or to the left.
EXPONENT_NEG14	Negative exponent operation, equivalent to $10^{-14}$ . Shifts decimal by 14 positions, depending on the type of conversion, to the right or to the left.
EXPONENT_POS1	Positive exponent operation, equivalent to $10^1$ . Depending on the type of conversion, this adds or removes one trailing zero.

HINT FLAG	DESCRIPTION
EXPONENT_POS2	Positive exponent operation, equivalent to $10^2$ . Depending on the type of conversion, this adds or removes two trailing zeros.
EXPONENT_POS3	Positive exponent operation, equivalent to $10^3$ . Depending on the type of conversion, this adds or removes three trailing zeros.
EXPONENT_POS4	Positive exponent operation, equivalent to $10^4$ . Depending on the type of conversion, this adds or removes four trailing zeros.
EXPONENT_POS5	Positive exponent operation, equivalent to $10^5$ . Depending on the type of conversion, this adds or removes five trailing zeros.
EXPONENT_POS6	Positive exponent operation, equivalent to $10^6$ . Depending on the type of conversion, this adds or removes six trailing zeros.
EXPONENT_POS7	Positive exponent operation, equivalent to $10^7$ . Depending on the type of conversion, this adds or removes seven trailing zeros.

Table 50: OMMNumeric Hint Flags

### 10.3.2.3 OMMNumeric Interface Methods

OMM preserves the precision of encoded numeric values. Developers should note that some conversions from OMM numeric types to primitive types (e.g. REAL to Java's long) may result in a loss of precision; this is an example of a **narrowing precision conversion** (as defined in *The Java™ Language Specification*). Also note that because the IEEE Standard for Floating-Point Arithmetic (IEEE 754) (implemented as the float and double types in Java) cannot represent some values exactly, rounding (per the IEEE 754 spec) may occur when converting from the OMM representation of numeric values to other data types manually, or via the provided utility methods. The following table lists methods that are supported by the OMMNumeric interface. As mentioned before, the hint flags apply to the REAL, REAL\_4RB, and REAL\_8RB OMMNumeric types only.

METHOD	DESCRIPTION
compareTo()	Compares two OMMNumeric values of any type. A blank value is considered as 0.
equals()	Returns true when two OMMNumeric of the same type are equal. Blank value is equal only to blank.
getHint()	Returns a byte, indicating the hint flag, set on this OMMNumeric data.
getLongValue()	Returns long part of real value, which is used in conjunction with hint.
hintString()	Returns a string representation of the hint flag, set on this OMMNumeric data.
toBigDecimal()	Returns an OMMNumeric converted to a BigDecimal. Any hint flag is applied.
toBigInteger()	Returns an OMMNumeric converted to a BigInteger. Any hint flag is applied.
toDouble()	Returns an OMMNumeric converted to a double. Any hint flag is applied.
toFloat()	Returns an OMMNumeric converted to a float. Any hint flag is applied.
toLong()	Returns an OMMNumeric converted to an int. Any hint flag is applied.
toString()	Returns a string representation of an OMMNumeric.

Table 51: OMMNumeric Interface Methods

### 10.3.2.4 OMMNumeric Utility Methods Applicable to OMMDData.REAL

The following table lists the OMMNumeric utility methods applicable to the **OMMDData.REAL** type that may introduce imprecision:

METHOD	DESCRIPTION
toBigInteger()	Converts the REAL to a BigInteger. This conversion is analogous to the <i>narrowing primitive conversion</i> from double to long as defined in <i>The Java™ Language Specification</i> : any fractional part of the REAL is discarded. Note that this conversion can lose information about the precision of the REAL value.
toDouble()	Converts the REAL to a double by multiplying the unscaled value by the value specified by the associated hint. Note that some values resulting from this multiplication cannot be represented exactly (per the IEEE 754 specification), thus the result of the multiplication may be rounded. Also note that even when the return value is finite, this conversion can lose information about the precision of the REAL value.
toFloat()	Converts the REAL to a float by multiplying the unscaled value by the value specified by the associated hint. This conversion is similar to the narrowing primitive conversion from double to float as defined in <i>The Java™ Language Specification</i> . Note that some values resulting from this multiplication cannot be represented exactly (per the IEEE 754 specification), thus the result of the multiplication may be rounded. Also note that even when the return value is finite, this conversion can lose information about the precision of the REAL value.
toLong()	Converts the REAL to a long by multiplying the unscaled value by the value specified by the associated hint, and any fractional part of the result will be discarded. This conversion is similar to the narrowing primitive conversion from double to short as defined in <i>The Java™ Language Specification</i> .  <b>NOTE:</b> This conversion can lose information about the overall magnitude and precision of this REAL value as well as return a result with the opposite sign.

**Table 52: REAL OMMNumeric Utility Methods that Can Introduce Imprecision**

### 10.3.2.5 OMMNumeric Utility Methods Applicable to OMMDData.DOUBLE

The following table lists the OMMNumeric utility methods applicable to the **OMMDData.DOUBLE** type that may introduce imprecision:

METHOD	DESCRIPTION
toBigDecimal()	Translates a DOUBLE into a BigDecimal which is the exact decimal representation of the double's binary floating-point value.  <b>NOTE:</b> The BigDecimal is constructed via its double constructor. The results of this constructor can be somewhat unpredictable. One might assume that writing new BigDecimal(0.1) in Java creates a BigDecimal which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257827021181583404541015625. This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed in to the constructor is not exactly equal to 0.1. Consider manually constructing a BigDecimal via the <a href="#">BigDecimal(String)</a> constructor (by calling <a href="#">toString()</a> on the DOUBLE) for less performant, but more predictable results.
toBigInteger()	This conversion is analogous to a narrowing primitive conversion as defined in <i>The Java™ Language Specification</i> .
toFloat()	This conversion is analogous to a narrowing primitive conversion as defined in <i>The Java™ Language Specification</i> .
toLong()	This conversion is analogous to a narrowing primitive conversion as defined in <i>The Java™ Language Specification</i> .

**Table 53: DOUBLE OMMNumeric Utility Methods that Can Introduce Imprecision**

### 10.3.2.6 OMMNumeric Utility Methods Applicable to OMMDData.Float

The following table lists the OMMNumeric utility methods applicable to the `OMMDData.FLOAT` type that may introduce imprecision:

METHOD	DESCRIPTION
<code>toBigDecimal()</code>	<p>Translates a <code>FLOAT</code> into a <code>BigDecimal</code> which is the exact decimal representation of the float's binary floating-point value.</p> <p><b>NOTE:</b> The <code>BigDecimal</code> is constructed via its double constructor. The results of this constructor can be somewhat unpredictable. One might assume that writing <code>new BigDecimal(0.1)</code> in Java creates a <code>BigDecimal</code> which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257827021181583404541015625. This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed in to the constructor is not exactly equal to 0.1. Consider manually constructing a <code>BigDecimal</code> via the <code>BigDecimal(String)</code> constructor (by calling <code>toString()</code> on the <code>FLOAT</code>) for less performant, but more predictable results.</p>
<code>toBigInteger()</code>	This conversion is analogous to a narrowing primitive conversion as defined in <i>The Java™ Language Specification</i> .
<code>toLong()</code>	This conversion is analogous to a narrowing primitive conversion as defined in <i>The Java™ Language Specification</i> .

**Table 54: FLOAT OMMNumeric Utility Methods that Can Introduce Imprecision**

### 10.3.2.7 OMMNumeric Utility Methods Applicable to OMMDData.INT

The following table lists the OMMNumeric utility methods applicable to the `OMMDData.INT` type that may introduce imprecision:

METHOD	DESCRIPTION
<code>toDouble()</code>	This conversion is similar to the <i>widening primitive conversion</i> from long to double as defined in <i>The Java™ Language Specification</i> . Conversion of a <code>INT</code> value to double may result in loss of precision -- that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.
<code>toFloat()</code>	This conversion is similar to the widening primitive conversion from long to float as defined in <i>The Java™ Language Specification</i> . Conversion of a <code>INT</code> value to float may result in loss of precision -- that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.

**Table 55: INT OMMNumeric Utility Methods that Can Introduce Imprecision**

### 10.3.2.8 OMMNumeric Utility Methods Applicable to OMMDData.UINT

The following table lists the OMMNumeric utility methods applicable to the `OMMDData.UINT` type that may introduce imprecision:

METHOD	DESCRIPTION
<code>toDouble()</code>	This conversion is similar to the widening primitive conversion from long to double as defined in <i>The Java™ Language Specification</i> . Conversion of a <code>UINT</code> value to double may result in loss of precision -- that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.
<code>toFloat()</code>	This conversion is similar to the widening primitive conversion from long to float as defined in <i>The Java™ Language Specification</i> . Conversion of a <code>UINT</code> value to float may result in loss of precision -- that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.

**Table 56: UINT OMMNumeric Utility Methods that Can Introduce Imprecision**

### 10.3.2.9 Encoding OMMNumerics

You can encode numerics using the appropriate `encodeXXX()` method specific to the OMMNumeric type. The string representation can be encoded using the `encodeString()` method and passing the type as an argument.

For a list of modifiers on OMMNumeric data, refer to Table 50.

The following example illustrates how to encode a numeric using the appropriate `encodeXXX()` method:

```
long longValue = 10;

switch(dataType) {
    case OMMTypes.INT:
    case OMMTypes.INT_1:
        ommEncoder.encodeInt(longValue);
        break;
    case OMMTypes.UINT:
    case OMMTypes.UINT_1:
    case OMMTypes.UINT_2:
    case OMMTypes.UINT_4:
    case OMMTypes.UINT:
        ommEncoder.encodeUInt(longValue);
        break;
}
```

#### Example 27: Encoding Numeric Primitives

The encoding of real numeric values typically utilizes the hint flags that imply operations on the data. The hint flags imply operation on the data during decoding. Thus, while encoding the operation on the initial value is reverse. For example a value of **34.27** can be encoded with **EXPONENT\_NEG2** as follows:

```
// encode real value 34.27, using EXPONENT_NEG2 hint flag
// to encode, apply operation reverse to EXPONENT_NEG2, i.e 34.27 * 102 = 3427
long longValue = 3427;
ommEncoder.encodeReal(longValue, OMMNumeric.EXPONENT_NEG2);
```

#### Example 28: Encoding Real Primitives Using Calculations

The string representation of a numeric is encoded using the `encodeString()` method. The following example illustrates passing the data type as an argument to the `encodeString()` method.

```
String stringData = "10.12";

switch(dataType) {
    case OMMTypes.REAL_8RB:    {
        ommEncoder.encodeString(stringData,OMMTypes.REAL_8RB);
        break;
    }
    case OMMTypes.REAL:       {
        ommEncoder.encodeString(stringData,OMMTypes.REAL);
        break;
    }
    ...
}
```

### Example 29: Encoding String Representations of Numeric Primitives

#### 10.3.2.10 Decoding OMMNumerics

The numeric types consist of a value and optionally, a hint flag. The presence of a hint flag applies to `OMMTypes.REAL` and indicates that the data needs to be modified by the manner indicated by the hint flag.

When the `OMMData` type is one of the numeric category, the object should be casted to the `OMMNumeric`. The application then can use the `toXXX()` methods to decode the value of the data. The following example shows decoding of the numeric data.

```
short dataType = data.getDataType();    // data is OMMData object to be decoded
switch(dataType)
{
    case OMMTypes.INT:
    case OMMTypes.UINT_1:    {
        long l = ((OMMNumeric)data).toLong();
        break;
    }
    case OMMTypes.FLOAT:
    case OMMTypes.FLOAT_4:    {
        float f = ((OMMNumeric)data).toFloat();
        ...    // other numeric OMMTypes
    }
}
```

### Example 30: Decoding Numeric Primitives

Decimals that have strict requirements on decimal precision, fractional values, and exponentials are contained in the `OMMNumeric` object and represented using `OMMTypes.REAL`.

Hint values are enumeration values representing decimals, fractions, and exponents, which help interpret the value contained in the `OMMNumeric` object.

If hint is less than `OMMNumeric.DIVISIOR_1` (a constant value of 22), it is an exponent:

- `exponent = hint - OMMNumeric.EXPONENT_0;` (hint - 14)
- The exponent is used during decoding, as follows:
- `doubleValue = intValue * 10exponent.`

If hint is greater or equal to `OMMNumeric.DIVISIOR_1`, it is a fraction divisor:

- `exponent = hint - OMMNumeric.DIVISIOR_1;` // hint - 22
- `divisor = 2exponent`
- The divisor is used during decoding, as follows:
- `doubleValue = longValue / divisor.`

## Examples

- `longValue` of 215 and a hint of `OMMNumeric.EXPONENT_NEG2` (12) results in  $215 \cdot (10^{-2}) = 2.15$ .
- `longValue` of 11 and `OMMNumeric.DIVISOR_4` (24) results in  $11 / (2^{24-22}) = 11/4 = 2 \frac{3}{4}$ .

The value obtained by `getLongValue()` is the raw value. The client application must apply the hint before interpreting or performing any calculations.

When performing floating point calculations, precision may be lost (for details, refer to Table 54, Table 55, and Table 56).

```
// decimals
if( hint < OMMNumeric.DIVISOR_1 )
{
  /* insert the decimal point back into a decimal value */
  int e = hint - OMMNumeric.EXPONENT_0;
  double y = Math.pow(10,e);
  double dv = value * y;

  System.out.println( dv );
}
else
{
  /* apply the denominator to the value to convert back to fraction */
  int e = hint - OMMNumeric.DIVISOR_1;
  double y = Math.pow(2, e);
  double dv = value / y;
  System.out.println( dv );
}
```

### Example 31: Decoding Decimals, Fractions, and Exponentials

## 10.3.3 OMMDDataBuffer

Several OMM types are `OMMDDataBuffer` objects. The `OMMDDataBuffer` interface supports strings and buffers. An `OMMDDataBuffer` object is encoded or decoded utilizing the interface. The interface allows Java strings and buffers to be converted to/from `OMMDDataBuffer` objects of a specific type. The `OMMDDataBuffer` is `OMMData`, thus, the interfaces supports conversion between Java strings and buffers types and `OMMData`.

The code below demonstrates the usage of the `OMMDDataBuffer` interface for encoding and decoding Java buffer.

```
// encoding a string value
Short type = OMMType.ASCII_STRING;
String value = "example";
_encoder.encodeString(value, type); // encoder converts to OMMData
OMMData convertedData = (OMMData)_encoder.getEncodedObject();

// decoding string value
OMMData data = getEncodedData() // method that returns decoded data
short type = data.getType();
switch (type){
  case OMMTypes.ASCII_STRING:
    String s = (OMMDDataBuffer)data.toString();
}
```

### Example 32: Using OMMDDataBuffer Interface

### 10.3.3.1 OMMDataBuffer Types

The following table lists the OMMDataBuffer types and the encoding methods supported by OMMEncoder. The string representation of the primitive data, if applicable, is encoded using the `encodeString()` method.

OMMTYPES	DESCRIPTION	ENCODING METHODS
ASCII_STRING	Buffer containing a string of 8-bit characters encoded using the Refinitiv Basic Character Set (RBCS).	<code>encodeString()</code> <code>encodeBytes()</code>
BUFFER	Byte buffer	<code>encodeString()</code> <code>encodeBytes()</code>
UTF8_STRING	Buffer containing a string encoded using the UTF-8 encoding of ISO 10646.	<code>encodeString()</code> <code>encodeBytes()</code>
RMTES_STRING	Buffer containing a string encoded using the Refinitiv Multilingual Text Encoding Standard.	<code>encodeString()</code> <code>encodeBytes()</code>
OPAQUE_BUFFER	This type of data is used by data formats.	Create byte buffer using application specific external encoder <code>encodeBytes()</code>
XML	XML format data	Create byte buffer using XML encoder (e.g. <code>javax.xml.stream.XMLEventWriter</code> ) <code>encodeBytes()</code>
ANSI_PAGE	ANSI Page data	Create byte buffer using <code>com.reuters.rfa.ansipage</code> package <code>encodeBytes()</code>

**Table 57: OMMDataBuffer OMMDataTypes**

### 10.3.3.2 OMMDataBuffer Interface Methods

The following table lists methods that are supported by the OMMDataBuffer interface.

METHOD	DESCRIPTION
<code>getBytes()</code>	Returns a copy of bytes contained by this buffer object. If the data is a String, no character conversion is performed.  An overloaded method takes a byte array parameter and an integer value. The buffer content is copied to the array parameter, starting at the position defined by the second integer parameter. The method returns the number of copied bytes.
<code>getString()</code>	Returns a string, decoded from the content of the buffer according to its type (e.g. RMTES, ASCII).
<code>hasPartialUpdates()</code>	Returns true if the buffer contains partial update sequences. This is only applicable to RMTES.
<code>horizontalPosition()</code>	Returns an offset of this partial sequence. This is only applicable to RMTES.
<code>partialUpdateIterator()</code>	Returns Iterator of OMMDataBuffer of partial field substrings. This is only applicable to RMTES.
<code>toString()</code>	Returns a string representation of the buffer.

**Table 58: OMMDataBuffer Interface Methods**

### 10.3.3.3 Encoding Strings

The strings can be encoded using the `encodeString()` method. In general, the `encodeString()` method can be used to encode any type of data represented as a string. The following examples illustrates passing the type as an argument to the `encodeString()` method.

Raw data can be represented either as a byte buffer or a string. The encoding methodology is based on the representation.

```
String stringValue = "hello";

if((dataType==OMMTypes.ASCII_STRING)||
    (dataType==OMMTypes.UTF8_STRING)||
    (dataType==OMMTypes.RMTES_STRING))
{
    ommEncoder.encodeString(stringValue, dataType);
}
```

#### Example 33: Encoding Strings

### 10.3.3.4 Encoding Raw Data Available as a Byte Buffer

The byte buffer representation can be encoded using the `encodeBytes()` method as shown below

```
byte [] groupIdBuffer;
ommEncoder.encodeBytes(groupIdBuffer);
```

#### Example 34: Encoding Raw Data Available in a Byte Buffer

### 10.3.3.5 Encoding Raw Buffer Represented as a String

This can be encoded using the `encodeString()` method and passing `OMMTypes.BUFFER` as the input argument. The string can also be converted to a byte buffer and encoded as shown in the previous section.

```
String _stringValue = "hello";
ommEncoder.encodeString(_stringValue ,OMMTypes.BUFFER);
```

#### Example 35: Encoding Raw Data available as a String

### 10.3.3.6 Encoding Blank Primitives

An application can encode a blank value by using the `encodeBlank()` method.

```
String stringData = "";
if(stringData.equals("")) {
    ommEncoder.encodeBlank();
}
```

#### Example 36: Encoding Blank Primitives

### 10.3.3.7 Decoding Strings

When the OMMData type is one of this category, the object should be casted to the OMMDataBuffer. The application then can use the `toString()` method to decode the data. The following example shows decoding of the objects.

```
short dataType = data.getDataType(); // data is OMMData object to be decoded
switch(dataType)
{
    case OMMTypes.BUFFER:
    case OMMTypes.UTF_STRING:
    case OMMTypes.ASCII_STRING:
        String s = ((OMMDataBuffer)data).toString();
}
```

#### Example 37: Decoding String

### 10.3.4 OMMDateTime

Several OMM types are OMMDateTime objects. The OMMDateTime interface supports calendar type data. An OMMDateTime object is encoded or decoded utilizing the interface. The interface allows Java Calendar data to be converted into OMMDateTime object of a specific type, and also convert in reverse order. The OMMDateTime is OMMData, thus, the interfaces supports conversion between Java Calendar data types and OMMData.

The code below demonstrates the usage of the OMMDateTime interface for encoding and decoding Java Calendar data.

```
// encoding a date value
Calendar date = new GregorianCalendar(TimeZone.getTimeZone("GMT"));
type = OMMTypes.DATE;
_encoder.encodeDate(date); // encoder converts to OMMData
OMMData convertedData = (OMMData)_encoder.getEncodedObject();

// decoding date value
OMMData data = getEncodedData() // method that returns decoded data
short type = data.getType();
switch (type){
    case OMMTypes.DATE:
        int date = (OMMDateTime)data.getDate();
}
```

#### Example 38: Using OMMDateTime Interface

#### 10.3.4.1 OMMDateTime Types

The following table lists the OMMDateTime types and the encoding methods supported by OMMEncoder. The string representation of the primitive data, if applicable, is encoded using the `encodeString()` method.

OMMTYPES	DESCRIPTION	ENCODING METHODS
DATE	Object containing date information, no time is specified.	encodeDate ( ) encodeString ( )
DATE_4	Object containing defined date information, no time is specified. This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeDate ( ) encodeString ( )
TIME	Object containing time information, no date is specified.	encodeTime ( ) encodeString ( )
TIME_3	Object containing time information, no date is specified. The time data contains seconds and no milliseconds. This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeTime ( ) encodeString ( )
TIME_5	Object containing time information, no date is specified. The time data contains seconds and milliseconds. This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeTime ( ) encodeString ( )

OMMTYPES	DESCRIPTION	ENCODING METHODS
TIME_7	Object containing time information, no date is specified. The time data contains seconds, milliseconds, and microseconds. This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeTime ( ) encodeString ( )
TIME_8	Object containing time information, no date is specified. The time data contains seconds, millisecond, microseconds, and nanosecond. This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeTime ( ) encodeString ( )
DATETIME	Object containing date and time information.	encodeDateTime ( ) encodeString ( )
DATETIME_7	Object containing date and time data (seconds and no milliseconds). This is a defined data type. Refer to Section 10.3.1 for details on defined data.	encodeDateTime ( ) encodeString ( )
DATETIME_9	Object containing date and time data (seconds and milliseconds). This is a defined data type. Refer to Section 10.3.1 for details on defined data.	encodeDateTime ( ) encodeString ( )
DATETIME_11	Object containing date and time data (seconds, milliseconds, and microseconds). This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeDateTime ( ) encodeString ( )
DATETIME_12	Object containing date and time data (seconds, millisecond, microseconds, and nanoseconds). This is a defined data type. For details on defined data, refer to Section 10.3.1.	encodeDateTime ( ) encodeString ( )

**Table 59: OMMDatetime OMMDataTypes**

### 10.3.4.2 OMMDateTime Interface Methods

The following table lists methods that are supported by the OMMDateTime interface.

METHOD	DESCRIPTION
compareTo()	Compares two OMMDateTme objects. The objects have to be of the same type, otherwise a <b>ClassCastException</b> is thrown.
getDate()	Returns the day of the month that the object represents.
getHour()	Returns the hour of the day that the object represents.
getMicrosecond()	Returns time in microseconds that the object represents.
getMillisecond()	Returns time in milliseconds that the object represents.
getMinute()	Returns time in minutes that the object represents.
getMonth()	Returns the month that the object represents.
getNanosecond()	Returns time in nanoseconds that the object represents.
getSecond()	Returns time in seconds that the object represents.
getYear()	Returns the year that the object represents.
isInactive()	Returns true if the data are invalid for the time or data.
toCalendar()	Return Calendar Java type value that the object represents.
toDate()	Return Date Java type value that the object represents.
toSeconds()	Returns the time as UTC milliseconds from the epoch.
toString()	Returns a string representation of the date, time, or both.

**Table 60: OMMDateTime Interface Methods**

### 10.3.4.3 Encoding Date, Time and DateTime

The encoding of Date and Time objects is done through `encodeXXX()` methods. The following example shows the usage of the `encodeXXX()` methods.

```
int year = 2008;
int month = 10;
int day = 15;

int hour = 2;
int minutes = 30;
int seconds = 45;
int milliseconds = 23;

switch(dataType)
{
    case OMMTypes.DATE:    {
        ommEncoder.encodeDate(year,month,day);
        break;
    }
    case OMMTypes.TIME:    {
        ommEncoder.encodeTime(hour,minutes,seconds,milliseconds);
        break;
    }
    case OMMTypes.DATETIME: {
        ommEncoder.encodeDateTime(year,month,day,hour,minutes,seconds,milliseconds);
        break;
    }
    :
}
```

#### Example 39: Encoding Date, Time and DateTime

### 10.3.4.4 Decoding Date, Time, and DateTime

When the OMMData type is one of this category, the object should be casted to the OMMDatetime. The application then can use the `getXXX()` accessors methods to decode the data. The following example shows decoding of the objects.

```
short dataType = data.getDataType(); // data is OMMData object to be decoded
switch(dataType)
{
    case OMMTypes.DATE:
    case OMMTypes.DATE_4: {
        OMMDatetime dateTime = (OMMDatetime)data;
        int date = dateTime.getDate();
        int month = dateTime.getMonth();
        int year = dateTime.getYear();
        break;
    }
    case OMMTypes.TIME: {
        OMMDatetime dateTime = (OMMDatetime)data;
        int hour = dateTime.getHour();
        int min = dateTime.getMinute();
        int sec = dateTime.getSecond();
        int msec = dateTime.getMillisecond();
        break;
    }
    case OMMTypes.DATETIME: {
        OMMDatetime dateTime = (OMMDatetime)data;
        int date = dateTime.getDate();
        int month = dateTime.getMonth();
        int year = dateTime.getYear();
        int hour = dateTime.getHour();
        int min = dateTime.getMinute();
        int sec = dateTime.getSecond();
        int msec = dateTime.getMillisecond();
        break;
    }
}
:
```

#### Example 40: Decoding Date, Time, and DateTime

### 10.3.5 OMMQos

The OMMQos is a primitive data type used as an OMM adapter for the QualityOfService data, information about service quality, described in Section 7.8.2. The OMMQos data is an optional element on OMM message and can be included in refresh response and status response types messages.

The OMMQos supports `toQos()` method that provides the adapter functionality for the QualityOfService. This method returns QualityofService type. The interface also supports `toString()` method that returns string representation of the QualityOfService value.

### 10.3.5.1 OMMQoS Definitions and Related Quality of Service Definitions

The following table shows definitions of OMMQoS and how they translate to the QualityOfService definitions. Refer to Table 31 for the QualityOfService definitions.

OMMQOS DEFINITIONS	QUALITYOFSERVICE DEFINITIONS
QOS_REALTIME_JIT	QualityOfService.QOS_REALTIME_JIT
QOS_REALTIME_TICK_BY_TICK	QualityOfService.QOS_REALTIME_TICK_BY_TICK
QOS_UNSPECIFIED	QualityOfService.QOS_REALTIME_UNSPECIFIED

**Table 61: OMMQoS Translation to QualityOfService**

### 10.3.5.2 Encoding QualityOfService

The Quality of Service details can be encoded directly or indirectly using the `encodeQos()` method.

In the direct method, the Timeliness and Rate parameters are passed as input arguments.

In the indirect method, a populated `OMMQoS` object is passed as the input argument. As an added convenience, the `OMMQoS` object provides constant objects, e.g. `QOS_REALTIME_TICK_BY_TICK`. These constant objects can also be passed as input arguments.

The following shows the different ways of encoding Quality of Service:

```
// direct method
ommEncoder.encodeQos(QualityOfService.REALTIME, QualityOfService.TICK_BY_TICK);

OR

// indirect method using a predefined QoS object;
ommEncoder.encodeQos(OMMQoS.QOS_REALTIME_TICK_BY_TICK);

OR

// indirect method using a populated QoS object;
OMMQoS qos = (OMMQoS)data;
ommEncoder.encodeQos(qos);
```

**Example 41: Encoding Quality of Service**

### 10.3.5.3 Decoding QualityOfService

When the OMMData type is OMMTypes.QOS, the object should be casted to the OMMQos. The application then can use the `getXXX()` accessors methods to decode the data. The following example shows decoding of the objects.

```
short dataType = data.getDataType(); // data is OMMData object to be decoded
switch(dataType)
{
    case OMMTypes.QOS:
        {
            long qosRate = ((OMMQos)data).toQos().getRate();
            long qosTimeliness = ((OMMQos)data).toQos().getTimeliness();
            break;
        }
    :
}
```

#### Example 42: Decoding QualityOfService

## 10.3.6 OMMState

OMMState is an element of OMM message that carries state information. Depending on the type and domain of the message, OMMState represents the state of the item stream, service, item group, etc. Refer to 1 OMM Usage Guide for details about the state codes related to different domains. OMMState objects can be included in response and acknowledgement message types.

### 10.3.6.1 OMMState Elements

The OMMState elements are described in the table below.

ELEMENT	DESCRIPTION
Data	Conveys data about the health of data flowing within a stream. Data State values are described in Table 63.
Stream	Conveys data about the stream's health. Stream State values are described in Table 64.
Code	An enumerated code value that conveys additional information about the current state. Typically indicates more specific information, possibly pertaining to a condition occurring upstream resulting in the current data and stream states. <code>Code</code> is typically used for informational purposes, and an application should not trigger specific behavior based on this content. State Code values are described in Table 65.
NackCode	An enumerated code value indicating reason for negative acknowledgement. Used in acknowledgment response type messages. If used, the other elements: Stream, Data and Code are irrelevant.
text	Specific text pertaining to the current data and stream state. Typically used for informational purposes and an application should not trigger specific behavior based on this content.

**Table 62: OMMState Elements**

### 10.3.6.2 OMMState Data State Definitions

The data state definitions are shown in the table below:

DEFINITION	DESCRIPTION
Data.NO_CHANGE	State of the item data did not change.
Data.OK	This definition is used to deliver item data status and indicates that data is good and up to date.
Data.SUSPECT	This definition is used to deliver item data status and indicates that some or all of the item's data is stale or unknown.

**Table 63: OMMState.Data Definitions**

### 10.3.6.3 OMMState Stream State Definitions

The stream state definitions are shown in the table below:

DEFINITION	DESCRIPTION
Stream.CLOSED	The stream for this item is closed.
Stream.CLOSED_RECOVER	The stream for this item is closed. The item may be available later and the application can attempt to re-open this item on another stream.
Stream.NONSTREAMING	This stream definition is included in the refresh response message to a nonstreaming request.
Stream.OPEN	Indicates an open stream
Stream.REDIRECT	This stream state indicates that the requested item is available with another name, which is included in the attribute information. The stream is closed. Consumers may opt to request the item using said name on another stream.
Stream.UNSPECIFIED	The stream state defaults to this value, if not initialized.

**Table 64: OMMState.Stream Definitions**

### 10.3.6.4 OMMState Code State Definitions

The code state definitions are shown in the table below:

DEFINITION	DESCRIPTION
Code.ALREADY_OPEN	Indicates that the stream item is already open.
Code.DACS_DOWN	Indicates that the connection to Refinitiv Data Access Control System is down, and users are not allowed to connect.
Code.DACS_MAX_LOGINS_REACHED	Indicates that the maximum number of Refinitiv Data Access Control System logins has been reached.
Code.DACS_USER_ACCESS_TO_APP_DENIED	Indicates that the user is not allowed to use the application.
Code.ERROR	Indicates an internal error from the sender. Specific information should be available in the text.
Code.EXCEEDED_MAX_MOUNTS_PER_USER	Indicates that the maximum number of mounts per user has been exceeded.
Code.FAILOVER_STARTED	Indicates that a component is recovering due to a failover condition.
Code.FAILOVER_COMPLETED	Indicates that the recovery from a failover condition has finished.
Code.FULL_VIEW_PROVIDED	Indicates that the full view (e.g. all available fields) is being provided, even though a specific view was requested.
Code.GAP_DETECTED	Indicates a gap was detected between messages. A gap might be detected via an external reliability mechanism, or when using the <a href="#">seqNum</a> element on message.
Code.GAP_FILL	Reserved for future use.
Code.INVALID_ARGUMENT	Indicates that a parameter on the request is invalid or unrecognized. Specific information should be available in the text.
Code.INVALID_VIEW	Indicates that the requested view is invalid, possibly due to bad formatting. Additional information should be available in the text.
Code.JIT_CONFLATION_STARTED	Indicates that Just-In-Time conflation has begun on the stream.
Code.NO_BATCH_VIEW_SUPPORT_IN_REQ	Indicates that the request does not support batch or view.
Code.NO_RESOURCES	Indicates that no resources are available to accommodate the stream.
Code.NON_UPDATING_ITEM	Indicates that a streaming request was made for non-updating data.

DEFINITION	DESCRIPTION
Code.NONE	No additional state code information is required or present.
Code.NOT_ENTITLED	Indicates that the request was denied due to permissioning. This typically indicates the requesting user is not permitted to request on the service, or to receive data at the requested Quality of Service.
Code.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.
Code.NOT_OPEN	Indicates that the stream was not open. Additional information should be available in the text.
Code.PREEMPTED	Indicates that the stream was preempted, possibly by a caching device. Typically indicates the user has exceeded an item limit, whether specific to the user or specific to a component in the system. Specific information should be available in the text.
Code.REALTIME_RESUMED	Indicates that just-in-time conflation on the stream has finished.
Code.SOURCE_UNKNOWN	Indicates that the requested service is not known, though the service might be available later.
Code.TIMEOUT	Indicates that a timeout occurred somewhere in the system while processing requested data.
Code.TOO_MANY_ITEMS	Indicates that a request cannot be processed because too many streams are already open.
Code.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.
Code.UNSUPPORTED_VIEW_TYPE	Indicates that the domain on which a request is made does not support the requested <code>viewType</code> .
Code.USAGE_ERROR	Indicates invalid usage of code within the system. Specific information should be available in the text.
Code.USER_UNKNOWN_TO_PERM_SYS	Indicates that the user is unknown to the permissioning system, which could be DACS, AAA, or RTED.

**Table 65: OMMState.Code Definitions**

### 10.3.6.5 OMMState NACK State Definitions

The NACK state definitions are shown in the table below:

DEFINITION	DESCRIPTION
NackCode.NACK_ACCESS_DENIED	User is not properly permissioned for posting on the item or service.
NackCode.NACK_DENIED_BY_SRC	The source being posted to has denied accepting this post message.
NackCode.NACK_GATEWAY_DOWN	A gateway device for handling posted or contributed information is down or not available.
NackCode.NACK_INVALID_CONTENT	The content of the post message is invalid and cannot be posted. This does not match the expected formatting for this post.
NackCode.NACK_NO_RESOURCES	Some component along the path of the post message does not have appropriate resources available to continue processing the post.
NackCode.NACK_NO_RESPONSE	There is no response from the source being posted to. This may mean that the source is unavailable or there is a delay in processing the posted information.
NackCode.NACK_NOT_OPEN	The item being posted to does not have an available stream open.
NackCode.NACK_SOURCE_DOWN	The source being posted to is down or not available.

DEFINITION	DESCRIPTION
NackCode.NACK_SOURCE_UNKNOWN	The source being posted to is unknown and is unreachable.
NackCode.NACK_SYMBOL_UNKNOWN	The item information provided with the post message is not recognized by the system, this may be an invalid item.

**Table 66: OMMState.NackCode Definitions**

### 10.3.6.6 OMMState Methods

OMMState supports accessor methods and utility functions. The table below summarizes OMMState methods:

METHOD	DESCRIPTION
clear()	Clears the instance of the OMMState. Must be called before reusing OMMPool instance.
getCode()	Returns the <code>Code</code> value. It should be one defined in Table 65.
getDataState()	Returns the <code>Data</code> value. It should be one defined in Table 63.
getStreamState()	Returns the <code>Stream</code> value. It should be one defined in Table 64.
getText()	Returns the <code>Text</code> value.
isFinal()	This method evaluates the <code>OMMState.Stream</code> element. If the stream state indicates the stream is closed (i.e., <code>OMMState.Stream</code> is equal <code>Stream.CLOSED</code> , <code>Stream.CLOSED_RECOVER</code> , <code>Stream.REDIRECT</code> , or <code>Stream.NONSTREAMING</code> and the indication flag <code>REFRESH_COMPLETE</code> is set), this method returns true.
isNackCode()	Indicates if there is a Nack code associated with the acknowledgement message.
setCode()	Sets the <code>Code</code> value. It should be one defined in Table 65.
setDataState()	Sets the <code>Data</code> value. It should be one defined in Table 63.
setStreamState()	Sets the <code>Stream</code> value. It should be one defined in Table 64.
setText()	Sets the <code>Text</code> value.
toString()	Returns a string representation of the OMMState object.

**Table 67: OMMState Methods**

### 10.3.6.7 Encoding State

State information such as stream state, data state, state code, and text can be encoded using the `encodeState()` interface. The example below shows the encoding:

```
byte _streamState = 4;    // "CLOSED_RECOVER"
byte _dataState = 1;    // "DATA_OK"
short _code = 0;        // "NONE"
String _text = "Status closed recover";

ommEncoder.encodeState(_streamState, _dataState, _code, _text);
```

#### Example 43: Encoding State

### 10.3.6.8 Decoding State

When the OMMData type is OMMTypes.STATE, the object should be casted to the OMMState. The application then can use the `getXXX()` accessors methods to decode the data. The following example shows decoding of the OMMState.

```
short dataType = data.getDataType();    // data is OMMData object to be decoded
switch(dataType)
{
    case OMMTypes.STATE:                {
        byte streamState = ((OMMState)data).getStreamState();
        byte dataState = ((OMMState)data).getDataState();
        short code = ((OMMState)data).getCode();
        String text = ((OMMState)data).getText();
        break;
    }
    :
}
```

#### Example 44: Decoding State

## 10.3.7 OMMEnum

The OMMEnum type is used for a numeric value representing an expanded string. Enumeration values are unsigned integer values from 0 to 65535. This value combined with `dictionary.FieldDictionaryexpandedValueFor` or `dictionary.EnumTable.expandedValueFor` will determine the string. This class has a natural ordering and it implements the Comparable interface.

### 10.3.7.1 Encoding Enumeration

The `encodeEnum()` method can be used to encode enumerations as shown below:

```
int enumValue = 10;
ommEncoder.encodeEnum (enumValue);
```

#### Example 45: Encoding Enumeration

### 10.3.7.2 Decoding Enumeration

The `getValue()` method can be used to decode enumerations as shown below:

```
int enumValue = ((OMMEnum)data).getValue() ;
```

#### Example 46: Decoding Enumeration

## 10.3.8 OMMArray

OMMArray is a container used for sequential data. Although it is a container, it is also considered to be a primitive type; it cannot contain nested elements. An OMMArray object has a header and entries. The header contains general information that is related to all the entries, such as data type, number of elements, and the width of the elements. The container can hold up to 65535 fixed width or variable width entries. The entries have to be of the same primitive type, and entries can't be another OMMArray type. The entries are accessed through an iterator, as the OMMArray implements OMMIterable.

### 10.3.8.1 OMMArray Methods

The following table summarizes general OMMArray information and accessor methods.

ELEMENT	METHOD	DESCRIPTION
data type	getDataType()	The type of the entries. It can only be a primitive type.
elements width	getWidth()	A width of 0 indicates entries with variable width. Otherwise all entries must have the same width. The value is related to the data type (i.e., the entries of float can have width 0 or 4).
count	getCount()	Number of elements the OMMArray contains.

**Table 68: OMMArray Elements**

### 10.3.8.2 Encoding Array

Array consists of a header and array entries. An Array can contain only primitive data type entries, and all the entries must have the same type.

The encoding steps are as follows:

- Initializing the Array using `encodeArrayInit()`.
- Encoding the entries.
  - Encoding the entry header using `encodeArrayEntryInit()`.
  - Encoding the entry data.
  - Completing the encoding using `encodeAggregateComplete()`.

### 10.3.8.3 Initializing Array

An Array is initialized using the `encodeArrayInit()` interface, and expects the data type and the element length as input arguments. The element length is an optional data and is set when the size of all elements is uniform. The size of 0 indicates to that the length needs to be calculated by encoder.

Array encoding initialization is shown in the example below. In this example, the elements are Strings, and since the item length is variable, the second parameter is 0.

```
// Array encoding initialization.
ommEncoder.encodeArrayInit(OMMTypes.ASCII_STRING, 0);
```

#### Example 47: Array Initialization

### 10.3.8.4 Encoding Array Entry

The `encodeArrayEntryInit()` method is used to initialize an element entry. This method does not take any parameters. The data encoding depends on the data type.

### 10.3.8.5 Completing Array Encoding

The `encodeAggregateComplete()` is used to indicate completion of the array encoding.

### 10.3.8.6 Array Encoding Example

The example code below shows encoding array that contains two ASCII String elements:

```
// Initialize Array
ommEncoder.encodeArrayInit(OMMTypes.ASCII_STRING,0);

// encode two entries
ommEncoder.encodeArrayEntryInit();
ommEncoder.encodeString("abc", OMMTypes.ASCII_STRING);

ommEncoder.encodeArrayEntryInit();
ommEncoder.encodeString("asdfg", OMMTypes.ASCII_STRING);

// Complete Array encoding
ommEncoder.encodeAggregateComplete();
```

#### Example 48: Encoding Array

### 10.3.8.7 Decoding Array with Example

The array properties, such as data type, number of elements, and the width of the elements are accessed using `getXXX()` methods. An iterator is set up using `iterator()` to traverse the entries. The iterator returns the entries as `OMMEntry`. The data in OMM form is obtained by calling `getData()` method. The application should retrieve the data type of the entries and decode accordingly. The array entry type is primitive type, thus the entry is decoded by casting the data to its type and using an appropriate method, as described throughout this chapter.

In the following example, an entry object is contained in the `OMMEntry` object. The `OMMData` object is extracted using `getData()`. Data content is decoded using the application-implemented `decodeData()` method.

```
int numberOfElements = anArray.getCount();
int itemLength = anArray.getWidth();
short dataType = anArray.getDataType();

// entries
for (Iterator iter = anArray.iterator(); iter.hasNext(); )
{
    OMMEntry entry = (OMMEntry) iter.next();
    OMMData data = entry.getData();

    decodeData(data);    // application implemented
}
```

#### Example 49: Decoding Array

## 10.4 Dataformats

A dataformat is made up of the header and a list of entries. The dataformat header contains the properties of the dataformat.

The entry itself is comprised of a header and the data content. The entry header contains entry properties. The data content along with the type represents the raw data. The data content could be another dataformat known as **nested dataformat** or simple data known as **primitive data**. The entry containing primitive data is known as a **leaf** and is the last in the hierarchy. An entry can also be blank with no data.

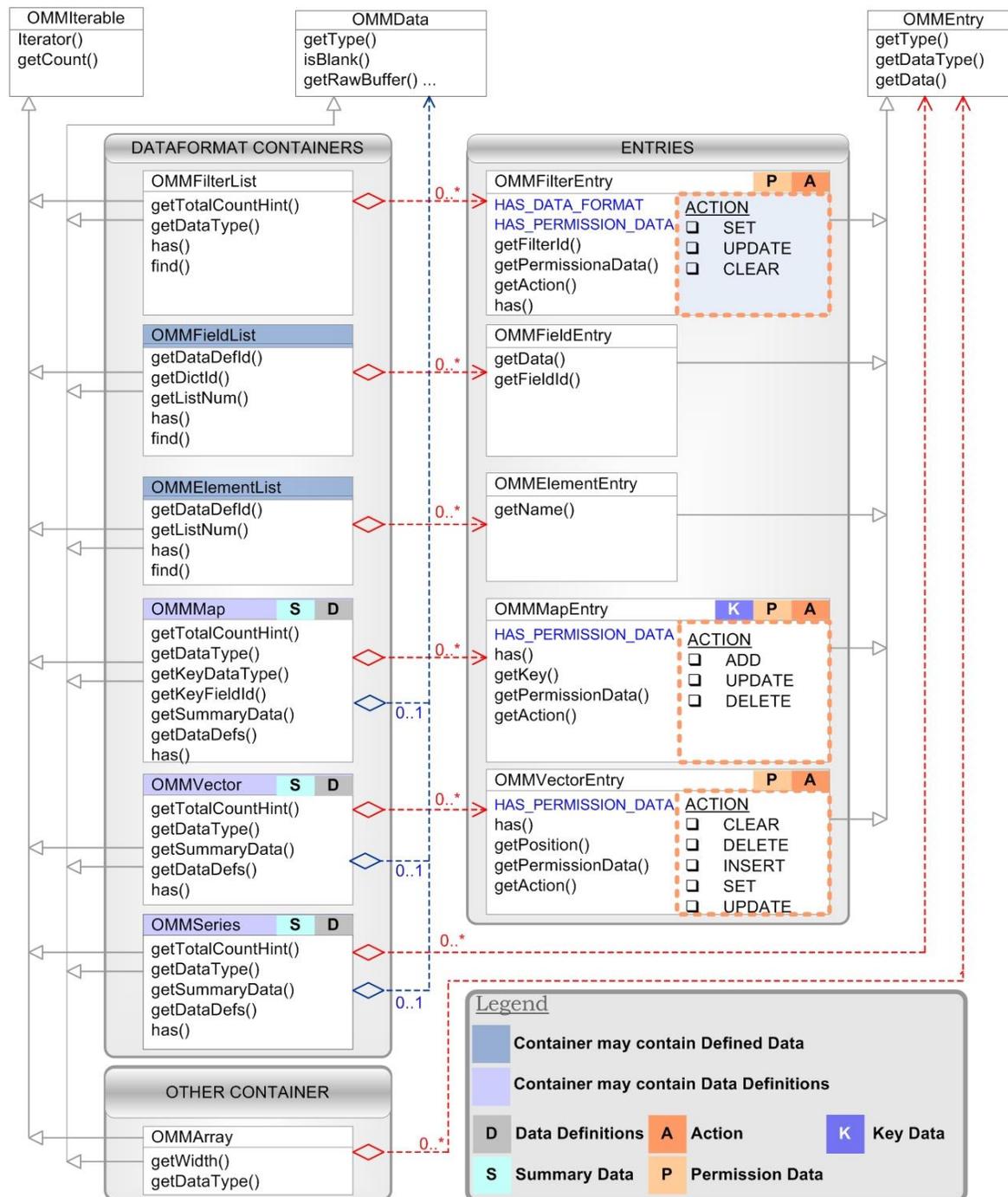
The entry data content is encapsulated in an object corresponding to the data type. Together, a dataformat and its contained entries, jointly resemble that of a composite pattern's composite. Jointly a container and its housed entries realize constructs such as field identifier value pairs, associative key value pairs or self-describing named value pairs (refer to Figure 34).

Based on the interface type, the presence of a property contained in the interface may be indicated with an indicator known as a **flag**. A single flag or a combination of flags is used by messages, dataformats and entries to convey the presence of the properties. There may be properties that do not have an accompanying flag.

Some interfaces require flags to be specified during initialization. For some interfaces there is no need to specify the flags as they would be implicitly set by RFA when the properties are specified.

It may be required to change the entries within a dataformat. This can be achieved through **actions**. Actions are operations on entries within a dataformat to manage change processing. Actions are available in `OMMFilterEntry`, `OMMMapEntry` and `OMMVectorEntry` objects. The supported actions differ depending on the object type.

The following diagram illustrates dataformat interfaces:



**Figure 36: Dataformat Interfaces in OMM Package**

The dataformats `OMMFieldList` and `OMMElementList` may contain defined data. The dataformats `OMMMap`, `OMMVector` and `OMMSeries` may contain data definitions and summary data. The entries `OMMFilterEntry`, `OMMMapEntry` and `OMMVectorEntry` may contain permission data and they support actions. The map entry contains key data.

## 10.4.1 Encoding Nested Dataformats

An application must use the `OMMEncoder` interface for encoding. The encoding process depends on the dataformat, and thus varies for the different dataformats.

### 10.4.1.1 Generic Steps

The process of encoding dataformat follows the sequence outlined below. Based on the scenario and dataformat, some of the steps may be omitted; however, the relative order of the remaining steps must be preserved:

- Initialize the encoder object with the data type and estimated size.
- Initialize the dataformat header with flags and other input information where appropriate, using `encodexxxInit()` method, where `xxx` is the name of the dataformat.
- Encode data definitions.
- Encode summary data.
- Encode the entries. This involves encoding the entry header and entry data. Encoding the entry header is not applicable for defined data and must be skipped. When both defined data and standard data is available, defined data must be encoded first followed by standard data.
- Encode the entry header; this is not applicable for defined data.
- If applicable, encode the data content within the entry. The data content can be one of the following:
  - Primitive data
  - Nested dataformat
  - Pre-encoded data contained in an `OMMData` object
  - Pre-encoded data in a raw byte buffer
- Repeat the preceding sequence for each entry.
- Complete the dataformat encoding by calling `encodeAggregateComplete()`. This is not applicable if the entries contain only defined data.

### 10.4.1.2 Recursive Encoding

The nested dataformat is encoded recursively, until the last Leaf is encoded. The following diagram shows an example of a nested dataformat, followed by an explanation of encoding process. This example is intended to be simple, and the dataformats presented here do not include data definitions, summary data, or actions.

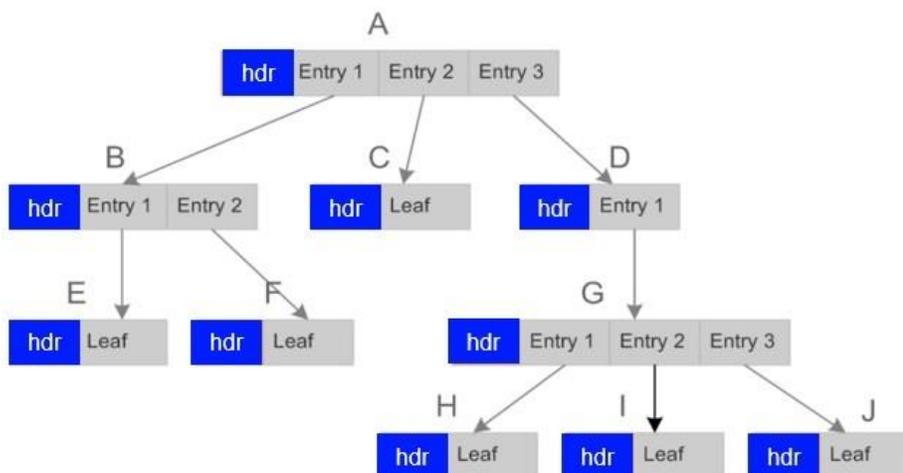


Figure 37: Nested Dataformats

In the preceding example, data for encoding is represented by A (containing a header and three entries).

Encoding a leaf means initializing the leaf header and encoding its value. The encoding steps are as follows:

- The dataformat A header is initialized.
- The first entry, dataformat B is encoded.
- The dataformat B header is initialized.
- The first entry, leaf E is encoded.
- The second entry, leaf F is encoded.
- The `encodeAggregateComplete()` is called to complete encoding dataformat B.
- The second entry, leaf C is encoded.
- The third entry, dataformat D is encoded.
- The dataformat D header is initialized.
  - The dataformat G is encoded.
  - The dataformat G header is initialized.
  - The first entry, leaf H is encoded.
  - The second entry, leaf I is encoded.
  - The third entry, leaf J is encoded.
  - The `encodeAggregateComplete()` is called to complete encoding dataformat G.
- The `encodeAggregateComplete()` is called to complete encoding dataformat D.
- The `encodeAggregateComplete()` is called to complete encoding dataformat A.

## 10.4.2 Decoding Nested Dataformats

The decoding process depends on the dataformat and thus varies for the different dataformats. The application uses iterator to retrieve entries from a container.

### 10.4.2.1 Generic Steps

The decoding process of dataformat follows the sequence as below. Based on the scenario and dataformat, some of the steps may be omitted; however, the relative order of the remaining steps must be preserved:

- Identify the type of the dataformat and convert the `OMMData` to the corresponding interface type
- Decode dataformat header
- Decode data definitions
- Decode summary data
- Retrieve the entries using the iterator and decode the entries

The dataformats implement the `OMMIterable` interface. Using this interface, the application can obtain iterator and number of entries, using the following methods `Iterator()` and `getCount()`, respectively.

### 10.4.2.2 Recursive Decoding

Similar to encoding, the nested dataformat is decoded recursively, until the last Leaf is decoded. Refer to the Figure 37 for an example of the nested dataformats. Using the same example, the decoding sequence is described below. As highlighted before, this example is intended to be simple, and the dataformats presented here do not include data definitions, or summary data, or actions.

- The dataformat A header is decoded.
- The iterator of dataformat A is retrieved.
- The first entry, dataformat B is decoded.
- The dataformat B header is decoded.
- The iterator of dataformat B is retrieved.
- The first entry, Leaf E is decoded.
- The second entry, Leaf F is decoded.
- The second entry, Leaf C is decoded.
- The third entry, dataformat D is decoded.
- The dataformat D header is decoded.
- The iterator of dataformat D is retrieved.
- The dataformat G is decoded.
  - The dataformat G header is decoded.
  - The iterator of dataformat G is retrieved.
  - The first entry, Leaf H is decoded.
  - The second entry, Leaf I is decoded.
  - The third entry, Leaf J is decoded.

### 10.4.3 ElementList

An *ElementList* is a container of flexible self-describing named entries known as *element entries*.

Element entries are identified with a string based tag and self-describe the type of data along with the actual data content. An optional *ElementList number* can exist to optimize any caching logic.

Element lists are easy to use but can be bandwidth intensive.

ELEMENT LIST DATA FORMAT			
- Element List Number			
- Count			
Element Entry 1	Tag - "Bid"	Type – signed integer	Data Content - 2589
Element Entry 2	Tag - "Ask"	Type – unsigned integer	Data Content - 987687

**Figure 38: ElementList Dataformat**

#### 10.4.3.1 Encoding ElementList

The ElementList contain header and element list entries. The ElementList can contain the following categories of entries:

- Standard data entries
- Defined data entries
- Defined data entries followed by standard data entries

### 10.4.3.1.1 Generic Steps

If the `ElementList` contains both defined data entries and standard data entries, the defined data entries are encoded prior to standard data entries. If the `ElementList` contains defined data entries, one of the parent dataformats (`Map`, `Vector` or `Series`) in the hierarchy should have encoded the corresponding data definitions. The encoding steps are as follows:

- Initializing the `ElementList` using `encodeElementListInit()`.
- Encoding the entries.
- If a standard type entry, encoding the entry header using `encodeElementEntryInit()`.
- Encoding the entry data.
- Completing the encoding using `encodeAggregateComplete()`, if the list contains any standard entries.

### 10.4.3.1.2 Initializing ElementList

An `ElementList` is initialized using the `encodeElementListInit()` method, and expects the flags, element list number, and definition ID as input arguments. Multiple flags can be set by combining them with the Java `OR` operator.

`ElementList` supports the following flags:

- `HAS_INFO`: indicates the presence of the element list number. The element list number specifies a template which can be used to pre-allocate and optimize the element list's cache.
- `HAS_DEFINED_DATA`: indicates the presence of defined data entries.
- `HAS_DATA_DEF_ID`: indicates the presence of the data definition ID; this is valid only if the `HAS_DEFINED_DATA` flag is also set, i.e., the `ElementList` must contain defined data.
- If this flag is not set, and the `HAS_DEFINED_DATA` flag is set, the data definitions contain one entry with ID of 0.
- `HAS_STANDARD_DATA`: indicates presence of standard data entries.

The `ElementList` encoding initialization is shown in the example below. In this example, the list uses template indicated by `elementListNumber` of 10, and all of the entries have standard data. The data definition is not used.

```
// ElementList encoding initialization.
int flags = OMMElementList.HAS_INFO | OMMElementList.HAS_STANDARD_DATA;
short elementListNumber = 10;
ommEncoder.encodeElementListInit(flags, elementListNumber, (short)0);
```

#### Example 50: ElementList Initialization

### 10.4.3.1.3 Encoding Element Entry

The `encodeElementEntryInit()` method is used to initialize an element entry with the element name and the data type. This is applicable only for standard data, and is not used for defined data.

```
ommEncoder.encodeElementEntryInit("MinOpenInterest", OMMTypes.UINT); // initialize the element entry
```

#### Example 51: Encoding Element Entry Header

The data may be standard data or defined data. The encoding process of a standard type entry depends on the type of the data.

An entry that is encoded using `DataDef` does not include header. Only the value is encoded. The entry name and data type is defined by the `DataDef`.

### 10.4.3.1.4 Completing ElementList Encoding

The `encodeAggregateComplete()` is used to indicate completion of the element list encoding. This is required if the element list contains standard data.

### 10.4.3.1.5 Element List Encoding Example

The presented example of an Element List contains defined data elements and standard data elements.

The data definition that is used by an element within an element list must be encoded in some parent in the hierarchy. The parent could be a `Map`, `Vector` or `Series`. The definition used for encoding defined data in the examples is shown below. The definition ID is 1 and it defines three entries. For details on encoding data definitions, refer to 10.8.2.

```
Definition Id = 1, Dataformat = ElementList, no. of entries = 3
  Definition Entry 1: name="s1" dataType= OMMTypes.REAL_4RB
  Definition Entry 2: name="s2" dataType= OMMTypes.ASCII_STRING
  Definition Entry 3: name="s3" dataType= OMMTypes.INT_4
```

#### Example 52: A Sample ElementList Definition

The example code shown below utilizes this data definition:

```
// Initialize ElementList that has defined and standard data, the element list number is not specified,
// and the data definition ID is 1
int flags = OMMElementList.HAS_DEFINED_DATA | OMMElementList.HAS_DATA_DEF_ID |
           OMMElementList.HAS_STANDARD_DATA;
ommEncoder.encodeElementListInit(flags, (short)0, (short)1);

// encode three entries according to the DataDef with ID=1
ommEncoder.encodeReal(13.23);           // entry name: "s1", dataType: OMMTypes.REAL_4RB, value: 13.23
ommEncoder.encodeString("rfa");        // entry name: "s2", dataType: OMMTypes.ASCII_STRING, value: "rfa"
ommEncoder.encodeInt(345);              // entry name: "s3", dataType: OMMTypes.INT_4, value: 345

// encode two standard data entries
ommEncoder.encodeElementEntryInit("e1", OMMTypes.INT); // entry name: "e1", dataType: OMMTypes.INT
ommEncoder.encodeInt(439);                // entry value: 439
ommEncoder.encodeElementEntryInit("e2", OMMTypes.REAL); // entry name: "e2", dataType: OMMTypes.REAL
ommEncoder.encodeInt(1.234);              // entry value: 1.234

// Complete element list encoding, since the list contained standard entries
ommEncoder.encodeAggregateComplete();
```

#### Example 53: Encoding Element List

### 10.4.3.2 Decoding ElementList

The OMMElementList extends OMMIterable, thus it provides `iterator()` method that returns Iterator object. Using the Iterator, the application can traverse through the element list elements.

#### 10.4.3.2.1 Decoding Defined Entries

If any of the elements has been encoded using Data Definitions, the flag `HAS_DEFINED_DATA` is set in Element List. If the parent dataformat containing the Data Definitions has more than one data definition, or one definition with ID other than 0, the flag `HAS_DATA_DEF_ID` is also set. The decoding application should test the two flags. If only `HAS_DEFINED_DATA` flag is set, and the other is not, the Data Definition ID of 0 is assumed. If both flags are set, the application should get the Data Definition ID via `getDataDefId()` method. Section 10.8.3 describes how to retrieve a Data Definition with the specified ID from the parent dataformat.

The application uses an Iterator from the DataDef object to decode names and types of element entries. The values of the entries are found by using an Iterator from the Element List.

#### 10.4.3.2.2 Decoding Standard Entries

If the Element List contains standard entries, the flag `HAS_STANDARD_DATA` is set in Element List. The application can find a number of standard entries by calling `getStandardCount()` method. The standard entries should be decoded after defined entries.

The entries are traversed using an Iterator from the Element List. The properties of the entries: name and type are obtained using `getXXX()` methods. The data obtained via accessor method is returned as OMMDData. An application can implement a method to decode the entries. The code below shows decoding Element List entries:

```
for (Iterator iter = in.iterator(); iter.hasNext(); )    // in is Element List
{
    OMMElementEntry ommEntry = (OMMElementEntry) iter.next()
    String entryName = ommEntry.getName();
    short entryType = ommEntry.getDataType();
    OMMDData data = ommEntry.getData();
    decodeData(data);                                // application implemented method
}
```

#### Example 54: Decoding Element Entry

## 10.4.4 FieldList

A **FieldList** represents a sequential container of identifier/value pair entries known as a **field entry**.

The field entry identifier is a numeric and known as **field identifier** or **Field Identifier**. Unlike the element entry, the data content in the field entry is not self describing, i.e., the type of the data is available in the field entry. A separate **data dictionary** that describes the Field Identifiers in detail is required to obtain the Field Identifier's tag (name) and data type. Dictionaries have versions and may be optionally identified through the **dictionary identifier**. An optional **field list number** can exist to optimize caching logic.

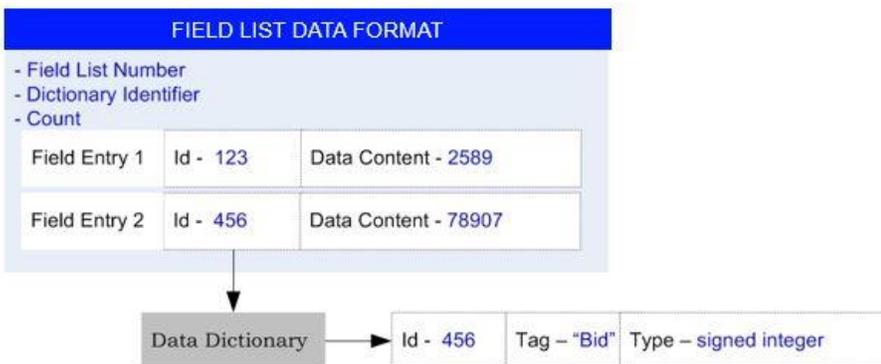


Figure 39: FieldList Dataformat

### 10.4.4.1 Encoding FieldList

The FieldList contains header and field list entries. The FieldList can contain the following categories of entries:

- Standard data entries
- Defined data entries
- Defined data entries followed by standard data entries

#### 10.4.4.1.1 Generic Steps

If the FieldList contains both defined data entries and standard data entries, the defined data entries are encoded prior to standard data entries. If the FieldList contains defined data entries, one of the parent dataformats (Map, Vector or Series) in the hierarchy should have encoded the corresponding data definitions. The encoding steps are as follows:

- Initializing the FieldList using `encodeFieldListInit()`.
- Encoding the entries.
- If a standard type entry, encoding the entry header using `encodeFieldEntryInit()`.
- Encoding the entry data.
- Completing the encoding using `encodeAggregateComplete()`, if the list contains any standard entries.

#### 10.4.4.1.2 Initializing FieldList

The FieldList is initialized using the `encodeFieldListInit()` interface and expects the flags, dictionary ID, field list number, and data definition ID as input arguments. Multiple flags can be set by combining them with the Java `OR` operator.

The FieldList supports the following flags:

- **HAS\_INFO**: indicates presence of a dictionary ID and a field list number. The field list number indicates a field list template that contains information about all possible fields contained in this field list, and is typically used by caching implementations to pre-allocate storage. The dictionary ID refers to an external dictionary, which contains specific name and type information that correlates to fielded values present in each field entry. The default dictionary, `RDMFieldDictionary`, has an ID of 1.
- **HAS\_DEFINED\_DATA**: indicates the presence of defined data entries. The definition ID must be passed as an input argument if defined data needs to be encoded.
- **HAS\_DATA\_DEF\_ID**: indicates the presence of the data definition ID; this is valid only if the **HAS\_DEFINED\_DATA** is also set, i.e., the FieldList must contain defined data. If this is not set, a definition ID of 0 is assumed.
- **HAS\_STANDARD\_DATA**: indicates presence of standard data entries.

FieldList encoding initialization is shown in the example below. In this example, the dictionary ID is 1, field list number is not specified, and the list uses defined data with ID of 2:

```
// FieldList encoding initialization.
int flags = OMMElementList.HAS_DATA_DEF_ID | OMMElementList.HAS_DEFINED_DATA;
short dictionaryId = (short)1;    // dictionary ID
short dataDefId = (short)2;      // DataDef ID (from dictionary)
ommEncoder.encodeFieldListInit(flags, dictionaryId, (short)0, dataDefId);
```

#### Example 55: FieldList Initialization

#### 10.4.4.1.3 Encoding Field Entry

The `encodeFieldEntryInit()` method is used to initialize an field entry with the field ID and the data type. This is applicable only for standard data and is not used for defined data.

```
short fid = (short)2;
ommEncoder.encodeFieldEntryInit(fid, OMMTypes.UINT); // initialize the field entry
```

#### Example 56: Encoding Field Entry Header

The data may be standard data or defined data. The encoding process of standard type entry depends on the type of the data.

An entry that is encoded using `DataDef` does not include a header. Only the value is encoded. The entry ID and data type is defined by the `DataDef`.

#### 10.4.4.1.4 Completing FieldList Encoding

The `encodeAggregateComplete()` is used to indicate completion of the field list encoding. This is required if the field list contains standard data.

#### 10.4.4.1.5 Field List Encoding Example

The following example of a Field List contains only defined data elements.

The data definition that is used by an element within element list must be encoded in some parent in the hierarchy. The parent could be a `Map`, `Vector` or `Series`. The definition used for encoding defined data in the examples is shown below. The definition ID is 3 and it defines two entries. Refer to 10.8.2. for details on encoding data definitions.

```
Definition Id = 3; Dataformat = FieldList; no. of entries = 2;
  Definition Entry 1; ID=30 dataType= OMMTypes.UINT
  Definition Entry 2; ID=31 dataType= OMMTypes.INT
```

#### Example 57: A Sample FieldList Definition

The example code shown below utilizes `RDMFieldDictionary`, and to optimize the encoded information even further, uses the above `FieldList` data definition:

```
// Initialize a fieldList that uses RDMFieldDictionary (ID = 1), and has defined data. The data
// definition ID is 3, and the field list number is 7. The data definition is stored in the dictionary,
// thus a flag OMMFieldList.HAS_DATA_DEF_ID is set.
int flags = OMMFieldList.HAS_DEFINED_DATA | OMMFieldList.HAS_DATA_DEF_ID | OMMFieldList.HAS_INFO;
short dictionaryId = (short)1;
short fieldListNumber = 7;
short dataDefId = (short)3;          // DataDef ID (from defined data dictionary)
ommEncoder.encodeFieldListInit(flags, dictionaryId, fieldListNumber, dataDefId);

// encode two entries according to the DataDef with ID=3
ommEncoder.encodeUINT(13);          // entry ID: 30, dataType: OMMTypes.UINT, value: 13
ommEncoder.encodeINT(16);          // entry ID: 31, dataType: OMMTypes.INT, value: 16

// The field list does not contain standard entries, the complete list encoding is not called.
```

#### Example 58: Encoding Field List

#### 10.4.4.2 Decoding a Field List

The field list header includes a field dictionary ID, which is used in decoding field list entries. An application can use the `getDictId()` method to retrieve the ID.

The `OMMFieldList` extends `OMMIterable`, thus it provides the `iterator()` method that returns an `Iterator` object. Using the `Iterator`, the application can traverse through the field list elements.

#### 10.4.4.2.1 Decoding Defined Entries

If any of the elements have been encoded using Data Definitions, the flag `HAS_DEFINED_DATA` is set in Field List. If the parent `dataformat` containing the Data Definitions has multiple data definitions, or one definition with an ID other than 0, the flag `HAS_DATA_DEF_ID` is also set. The decoding application should test the two flags. If only the `HAS_DEFINED_DATA` flag is set, a Data Definition ID of 0 is assumed. If both flags are set, the application should get the Data Definition ID via the `getDataDefId()` method. Section 10.8.3 describes how to retrieve a Data Definition with the specified ID from the parent `dataformat`.

The application uses `Iterator` from the `DataDef` object to decode IDs and types of field entries. The values of the entries are found by using an `Iterator` from the field list.

### 10.4.4.2.2 Decoding Standard Entries

If a field list contains standard entries, the flag `HAS_STANDARD_DATA` is set in the field list. The application can find a number of standard entries by calling the `getStandardCount()` method. The standard entries should be decoded after defined entries.

The entries are traversed using an iterator from the field list. The properties of the entries, ID and type, are obtained using `getxxx()` methods. The application uses the dictionary identified in the header to look up the entry ID to derive the tag and type related to this field entry. The data obtained via an accessor method is returned as `OMMData`. An application can implement a method to decode the entries. The code below shows decoding field list entries:

```
short dictionaryId = in.getDictId();
FieldDictionary fieldDictionary = getFieldDictionary(dictionaryId);    // application method

for (Iterator iter = in.iterator(); iter.hasNext(); )    // in is Field List
{
    OMMFieldEntry ommEntry = (OMMFieldEntry) iter.next()
    short entryId = ommEntry.getFieldId();
    OMMData data = ommEntry.getData();
    decodeData(data, fieldDictionary);    // application implemented method
}
```

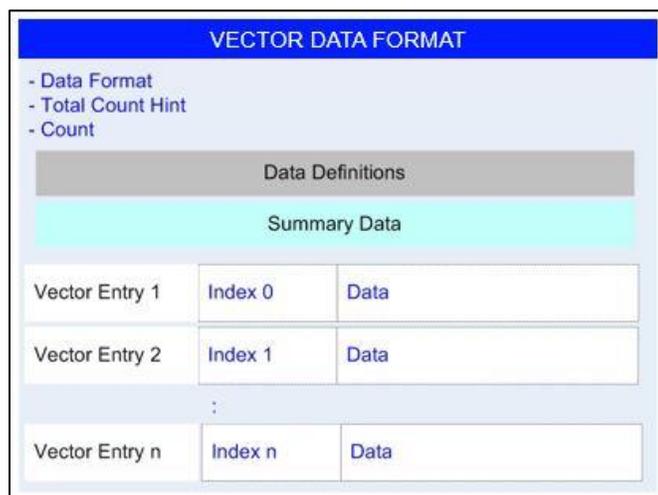
#### Example 59: Decoding Field Entry

## 10.4.5 Vector

A **Vector** is a container of position-oriented, highly manipulable entries each known as a **vector entry**.

The vector entry identifier is a position within the Vector and is identified by an integer index value. The index starts at 0 and can go as high as a 30-bit unsigned integer value.

Entries in the Vector can be set, updated or cleared and can optionally each have a separate permissions data for even finer control. A Vector can also optionally support sorting operations such as insert and delete.



**Figure 40: Vector Dataformat**

The dataformat for each vector entry data is identified once in the vector header (i.e., all vector entries must be the same dataformat). Vectors optionally contain summary data that applies to the entire structure. Data definitions can also optionally be defined when vector entries contain repetitive characteristics.

### 10.4.5.1 Encoding Vector

A Vector contains a header and vector entries. The entries are of the same type, and can be primitive data or dataformats. The header may contain summary data, data definitions, and number of vector entries.

#### 10.4.5.1.1 Generic Steps

The general process for encoding a Vector is shown below:

- Initialization of a Vector with flags, data type, and total count of vector entries. The data type applies to entries and summary data.
- Optionally, data definitions encoding. The data definitions must be encoded before summary data.
- Optionally, summary data encoding. The summary data may use data definitions encoded in the previous step. Summary data must be of the same type as entries.
- Encoding vector entries – the following is done for each entry:
  - Encoding the vector entry header.
  - Encoding the data contained inside the vector entry.
  - Completing vector encoding.

#### 10.4.5.1.2 Initializing Vector

A Vector is initialized using the `encodeVectorInit()` method, and requires the flags, vector entries data type and number of entries as input arguments. Multiple flags can be set by combining them with the Java **OR** operator.

The Vector supports the following flags:

- **HAS\_DATA\_DEFINITIONS**: indicates the presence of data definitions. If this is set, the data definitions must be encoded and the vector entries must be either `OMMTypes.ELEMENT_LIST` or `OMMTypes.FIELD_LIST`.
- **HAS\_PERMISSION\_DATA\_PER\_ENTRY**: indicates that some or all entries have permission data.
- **HAS\_SORT\_ACTIONS**: indicates that the vector is sortable. If the flag is set, the vector entries may encode `INSERT` or `DELETE` actions.
- **HAS\_SUMMARY\_DATA**: indicates the presence of summary data.
- **HAS\_TOTAL\_COUNT\_HINT**: indicates the presence of number of entries data.

The Vector encoding initialization is shown in the example below. In this example the vector has data definitions, and has 4 elements. The elements are of `OMMTypes.ELEMENT_LIST` type.

```
// Vector encoding initialization.
int flags = OMMVector.HAS_DATA_DEFINITIONS | OMMVector.HAS_TOTAL_COUNT_HINT;
short dataType = OMMTypes.ELEMENT_LIST;
short count = 4;
ommEncoder.encodeVectorInit(flags, dataType, count);
```

#### Example 60: Vector Initialization

#### 10.4.5.1.3 Encoding Data Definitions

Data definitions encoding is described in Section 10.8.2. The availability of data definitions must be indicated by setting the flag to **HAS\_DATA\_DEFINITIONS** when the Vector is initialized.

#### 10.4.5.1.4 Encoding Summary Data

Summary data encoding is described in Section 10.6.1. The presence of summary data must be indicated by setting the hint flag to `HAS_SUMMARY_DATA` when the Vector is initialized.

#### 10.4.5.1.5 Encoding Vector Entry

The `encodeVectorEntryInit()` method is used to initialize a vector entry. The vector entry contains a header and data. The header may contain action and permission data. Supported actions include CLEAR, DELETE, INSERT, SET, UPDATE. If the entry contains permission data, the `OMMVectorEntry.HAS_PERMISSION_DATA` flag is set. If any vector entry contains the permission data, the vector flag `OMMVector.HAS_PERMISSION_DATA_PER_ENTRY` is set. The code example below demonstrates initializing encoding of vector entry. In this example, the entry has permission data and has an action UPDATE. The entry index is 2.

Data encoding depends on the data type.

```
int flags = OMMVectorEntry.HAS_PERMISSION_DATA;
byte action = OMMVectorEntry.Action.UPDATE;
int index = 2;
byte [] permissionExpression = { 0x2A, 0x67, 0xBC, 0x3B };
ommEncoder.encodeVectorEntryInit(flags, action, index, permissionExpression); // initialize vector entry
```

#### Example 61: Encoding Vector Entry Header

#### 10.4.5.1.6 Completing Vector Encoding

The `encodeAggregateComplete()` is used to indicate completion of the vector encoding.

### 10.4.5.1.7 Vector Encoding Example

In the following example, the vector has summary data, does not contain defined data, and has two element list entries:

```
// Initialize vector that has summary data, the data type is element list and the vector contains two
// entries
int flags = OMMElementList.HAS_SUMMARY_DATA | OMMElementList.HAS_TOTAL_COUNT_HINT;
int numEntries = 2;
ommEncoder.encodeVectorInit(flags, OMMTypes.ELEMENT_LIST, numEntries);

// encode summary data
ommEncoder.encodeSummaryDataInit();
ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
ommEncoder.encodeElementEntryInit("NumStandbyServers", OMMTypes.UINT);
ommEncoder.encodeUInt(3);
ommEncoder.encodeAggregateComplete()    // complete element list

// encode first entry
// the flags are not set, action is SET, the index entry is 0, no permission expressions.
ommEncoder.encodeVectorEntryInit(0, OMMVectorEntry.Action.SET, 0, null);
ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
ommEncoder.encodeElementEntryInit("Hostname", OMMTypes.ASCII_STRING);
ommEncoder.encodeString("Galaxite");
ommEncoder.encodeElementEntryInit("Port", OMMTypes.UINT);
ommEncoder.encodeUInt(1423);
ommEncoder.encodeAggregateComplete();    // complete element list

// encode second entry
// the flags are not set, action is SET, the index entry is 1, no permission expressions.
ommEncoder.encodeVectorEntryInit(0, OMMVectorEntry.Action.SET, 1, null);
ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
ommEncoder.encodeElementEntryInit("Hostname", OMMTypes.ASCII_STRING);
ommEncoder.encodeString("Leonida");
ommEncoder.encodeElementEntryInit("Port", OMMTypes.UINT);
ommEncoder.encodeUInt(1424);
ommEncoder.encodeAggregateComplete();    // complete element list

// Complete vector encoding
ommEncoder.encodeAggregateComplete();
```

#### Example 62: Encoding Vector

### 10.4.5.2 Decoding Vector

Vector properties are extracted using `getXXX()` methods. In the example below, the method `getCount()` is used to retrieve the number of entries.

The presence of total count hint is verified using the `OMMVector.HAS_TOTAL_COUNT_HINT` flag. If available, the number is extracted using `getTotalCountHint()`.

A Vector may contain data definitions. The presence of data definitions is identified by the `OMMVector.HAS_DATA_DEFINITIONS` flag. The data definitions decoding is described in Section 10.8.3.

The presence of summary data is verified using `OMMVector.HAS_SUMMARY_DATA` flag. Summary data decoding is described in Section 10.6.2.

If the vector can be sorted, the `OMMVector.HAS_SORT_ACTION` flag is set. If set, the vector entries can apply INSERT and DELETE actions.

The vector entries are retrieved using the vector Iterator.

#### 10.4.5.2.1 Decoding Vector Example

```
// "in" is an OMMVector
int numberEntries = in.getCount(); // number of vector entries contained in the "in" vector
int totalCountHint;
boolean sortable = false; // initialize the sortable property of vector to false
if (in.has(OMMVector.HAS_TOTAL_COUNT_HINT)) // vector has total count hint
{
    totalCountHint = in.getTotalCountHint();
}
if (in.has(OMMVector.HAS_DATA_DEFINITIONS)) // vector has data definitions
{
    // decode data definitions
}
if (in.has(OMMVector.HAS_SUMMARY_DATA)) // vector has summary data
{
    // decode summary data
}
if (in.has(OMMVector.HAS_SORT_ACTIONS)) // vector is sortable
    sortable = true;
}

// decode entries
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMVectorEntry ommEntry = (OMMVectorEntry) iter.next();
    // if application needs the vector entry index, may use getPosition method
    int index = ommEntry.getPosition();
    decodeVectorEntry(ommEntry); // application method
}
}
```

#### Example 63: Decoding Vector

### 10.4.5.2.2 Decoding Vector Entry

The entries retrieved using a Vector Iterator are cast to an `OMMVectorEntry` object.

A vector entry may contain an action. If the `getAction()` method returns an action defined by `OMMVector`, the decoding application should apply the action to the vector entry. Section 10.7 summarizes dataformats actions. If the Vector is sortable, the cache may use INSERT and DELETE actions to sort the entries in the cache.

If any vector entry contains permission data, the `OMMVector.HAS_PERMISSION_DATA_PER_ENTRY` flag is set in the Vector. Decoding of permission data is described in Section 10.5.2.

The data object is retrieved from the vector entry using `getData()` after ensuring that the action is not a `DELETE` action.

The code example below shows decoding of Vector entries.

```
// "in" is OMMVector
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMVectorEntry ommEntry = (OMMVectorEntry) iter.next();
    byte action = 0;
    if (ommEntry.getAction() != 0)    // Vector entry has action
        action = ommEntry.getAction();
    if (in.has(OMMVector.HASPERMISSION_DATA_PER_ENTRY)    // Vector entry may have permission data
        if (ommEntry.has(OMMVectorEntry.HAS_PERMISSION_DATA))    // Vector entry has permission data
            decodePermissionData(ommEntry.getPermissionData());    // application implemented
    if (action != OMMMapEntry.Action.DELETE){
        OMMData data = ommEntry.getData();
        decodeData(data);    // application method
    }
}
```

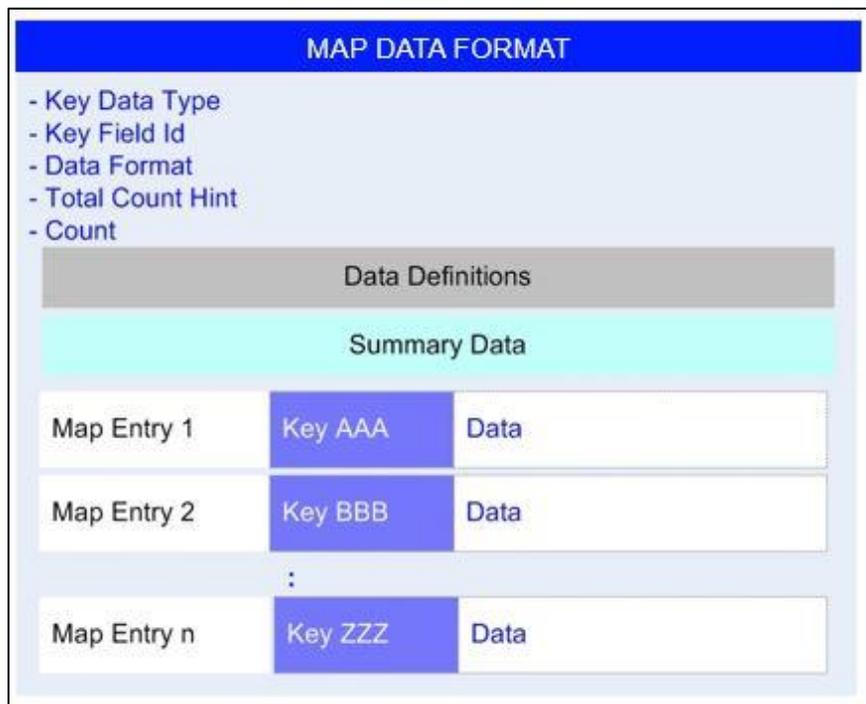
#### Example 64: Decoding Vector Entry

## 10.4.6 Map

A **Map** is a unique associative container that maintains a collection of key/value paired entries. Each entry is referred to as the **map entry**.

The identifier in the map entry is a **key** that can be of any primitive type (e.g. signed integer, ASCII string).

Entries in the Map can be added, updated or deleted, and can optionally each have a separate permissions expression for even finer control.



**Figure 41: Map Dataformat**

The dataformat for each map entry data is identified once in the map header (i.e., all map entries must be the same dataformat). Maps optionally contain summary data that applies to the entire structure. Data definitions can also optionally be defined when map entries contain repetitive characteristics.

### 10.4.6.1 Encoding Map

The Map contains header and map entries. The entries are of the same type, and can be primitive data or dataformats. The header may contain summary data, data definitions, key field ID, and number of vector entries.

#### 10.4.6.1.1 Generic Steps

The general process for encoding a Map is shown below:

- Initialization of Map with flags, key data type, map entries data type, total count hint and key field ID
- Optionally, data definitions encoding
- Optionally, summary data encoding
- Encoding map entries – the following is done for each entry
- Encoding the map entry header
- Encoding the map entry key
- Encoding the data contained inside the map entry
- Completing map encoding

### 10.4.6.1.2 Initializing Map

The Map is initialized using the `encodeMapInit()` interface and passing in the flags, key data type, data type of all the map entries, total count hint and key field ID as input arguments.

The flags supported by the Map are described below. Multiple flags can be set by using the Java `OR` operator.

- `HAS_DATA_DEFINITIONS`: Map contains data definitions
- `HAS_SUMMARY_DATA`: Map contains summary data
- `HAS_PERMISSION_DATA_PER_ENTRY`: permission data is available in some or all map entries
- `HAS_TOTAL_COUNT_HINT`: total count hint is available
- `HAS_KEY_FIELD_ID`: key field ID is available

The following example shows the Map initialization:

```
// Map encoding initialization.
// The flags indicate that the map has a total hint count and data definitions
int flags = OMMMap.HAS_TOTAL_COUNT_HINT | OMMMap.HAS_DATA_DEFINITIONS;

// The key type is ascii string
short keyDataType = OMMTypes.ASCII_STRING;

// The data type of all entries is filter list
short dataType = OMMTypes.FILTER_LIST;

// The map contains one entry
int totalHintCount = 1;

// The field key ID is 0
short keyFieldID = (short)0;

ommEncoder.encodeMapInit(flags, keyDataType, dataType, totalHintCount, keyFieldID);
```

#### Example 65: Map Initialization

### 10.4.6.1.3 Encoding Map Data Definitions

Data definitions encoding is described in Section 10.8.2. The availability of data definitions must be indicated by setting the flag to `HAS_DATA_DEFINITIONS` when the Map is initialized.

### 10.4.6.1.4 Encoding Map Summary Data

Summary data encoding is described in Section 10.6.1. The presence of summary data must be indicated by setting the hint flag to `HAS_SUMMARY_DATA` when the Map is initialized.

### 10.4.6.1.5 Encoding Map Entries

For each of the Map entries the encoding comprises of encoding header, key and data, as described in the three subsequent sections. The flag `HAS_TOTAL_COUNT_HINT` is set when the number of entries is specified during map initialization. The number of encoded entries should match the number passed during initialization.

### 10.4.6.1.6 Encoding Map Entry Header

The map entry is initialized using the `encodeMapEntryInit()` interface. This method takes an integer representing flags, a byte representing an action, and a byte array representing permission as input parameters.

- The only flag supported is the `HAS_PERMISSION_DATA`, which must be set if the map entry contains permission data.
- The action can be one of the following: `UPDATE`, `ADD` or `DELETE`.
- The availability of permission data is indicated by setting `HAS_PERMISSION_DATA` flag.

### 10.4.6.1.7 Encoding Map Entry Key

The map key can be any primitive data type except array, and the type is specified during the Map initialization. The encoding of the key corresponds to its type; i.e., if the key data type is a signed integer, `encodeInt()` is used, else if the key data type is a string, `encodeString()` is used.

The following example shows the map entry initialization and key data encoding:

```
// Map entry encoding initialization.
int flags = 0;
short action = OMMMapEntry.Action.ADD;
ommEncoder.encodeMapEntryInit(flags, action, null);
// encode the key
ommEncoder.encodeString("DIRECT_FEED",OMMTypes.ASCII_STRING);
```

#### Example 66: Map Entry Initialization and Key Encoding

### 10.4.6.1.8 Encoding Map Entry Data

Typically the map entry data is a nested data format. The encoding process depends on the data type.

### 10.4.6.1.9 Completing Map Encoding

The `encodeAggregateComplete()` method is used to indicate completion of the map encoding.

### 10.4.6.1.10 Map Encoding Example

The code below presents the example of encoding the `OMMMap`. The map entries contain `FieldLists` with defined data.

```
int flags =      OMMMap.HAS_DATA_DEFINITIONS |
                OMMMap.HAS_SUMMARY_DATA |
                OMMMap.HAS_PERMISSION_DATA_PER_ENTRY |
                OMMMap.HAS_TOTAL_COUNT_HINT;

////////////////////////////////////
///// Map Header
////////////////////////////////////
ommEncoder.encodeMapInit(flags, OMMTypes.ASCII_STRING, OMMTypes.FIELD_LIST , 2, (short)0);

////////////////////////////////////
///// Data Definition of FieldList
////////////////////////////////////
ommEncoder.encodeDataDefsInit();
ommEncoder.encodeFieldListDefInit((short)1);           // Data Definition 1
ommEncoder.encodeFieldEntryDef((short)3, OMMTypes.RMTES_STRING); //DSPLY_NAME
ommEncoder.encodeFieldEntryDef((short)22, OMMTypes.REAL); //BID
ommEncoder.encodeFieldEntryDef((short)30, OMMTypes.REAL); //BIDSIZE
```

```

ommEncoder.encodeListDefComplete();           // End of Definition 1

ommEncoder.encodeFieldListDefInit((short)2); // Data Definition 2
ommEncoder.encodeFieldEntryDef((short)22, OMMTypes.REAL); //BID
ommEncoder.encodeFieldEntryDef((short)25, OMMTypes.REAL); //ASK
ommEncoder.encodeListDefComplete();         // End of Definition 2
ommEncoder.encodeDataDefsComplete();        // End of Data Definitions

////////////////////////////////////
///// Summary Data
////////////////////////////////////
//data type of summary data must be the same as dataType
ommEncoder.encodeSummaryDataInit();
ommEncoder.encodeFieldListInit(OMMFieldList.HAS_STANDARD_DATA, (short)0, (short)0, (short)0);
ommEncoder.encodeFieldEntryInit((short)15, OMMTypes.ENUM);
ommEncoder.encodeEnum(840);
ommEncoder.encodeFieldEntryInit((short)16, OMMTypes.DATE);
ommEncoder.encodeDate(2007, 2, 27);
ommEncoder.encodeAggregateComplete();

////////////////////////////////////
///// Map Entries
////////////////////////////////////
flags = OMMMapEntry.HAS_PERMISSION_DATA;
ommEncoder.encodeMapEntryInit(flags, OMMMapEntry.Action.ADD, _permExp); // assume _permExp is set before
//Key 1
ommEncoder.encodeString("AA1.0", OMMTypes.ASCII_STRING);
//Value
ommEncoder.encodeFieldListInit(OMMFieldList.HAS_DEFINED_DATA | OMMFieldList.HAS_DATA_DEF_ID, (short)0,
    (short)0, (short)1);
// encode defined data for definition 1
ommEncoder.encodeString("AA.0", OMMTypes.RMTES_STRING);
ommEncoder.encodeReal(335, OMMNumeric.EXPONENT_NEG2);
ommEncoder.encodeReal(333, OMMNumeric.EXPONENT_NEG2);
//Has only define data, no need to call_ommEncoder.encodeAggregateComplete(); //FieldList

ommEncoder.encodeMapEntryInit(flags, OMMMapEntry.Action.ADD, _permExp);
//Key 2
ommEncoder.encodeString("AA2.0", OMMTypes.ASCII_STRING);
//Value
ommEncoder.encodeFieldListInit(OMMFieldList.HAS_DEFINED_DATA | OMMFieldList.HAS_DATA_DEF_ID, (short)0,
    (short)0, (short)2);
// encode defined data for definition 2
ommEncoder.encodeReal(433, OMMNumeric.EXPONENT_NEG2);
ommEncoder.encodeReal(443, OMMNumeric.EXPONENT_NEG2);
//Has only define data, no need to call _ommEncoder.encodeAggregateComplete(); //FieldList

ommEncoder.encodeAggregateComplete(); // MAP

```

### Example 67: Map Encoding

### 10.4.6.2 Decoding Map

Map properties are extracted using `getXXX()` methods. In the example below, the methods `getCount()` and `getKeyDataType()` are used to retrieve the count and key data type.

The presence of total count hint is verified using the `OMMMap.HAS_TOTAL_COUNT_HINT` flag. If available, the number is extracted using `getTotalCountHint()`.

The presence of the key field ID is verified using the `OMMMap.HAS_KEY_FIELD_ID` flag. If available, the key field is extracted using `getKeyFieldId()`.

A Map may contain data definitions. The presence of data definitions is identified by the `OMMMap.HAS_DATA_DEFINITIONS` flag. The data definitions are decoded as described in Section 10.8.3.

The presence of summary data is verified using `OMMMap.HAS_SUMMARY_DATA` flag. Summary data decoding is described in Section 10.6.2.

Map entries are retrieved using the map iterator.

#### 10.4.6.2.1 Decoding Map Example

```
// "in" is an OMMMap
int numberEntries = in.getCount(); // number of map entries contained in the "in" map
short keyDataType = in.getKeyDataType();
int totalCountHint;
short keyFieldId;
if (in.has(OMMMap.HAS_TOTAL_COUNT_HINT)) // Map has total count hint
{
    totalCountHint = in.getTotalCountHint();
}
if (in.has(OMMMap.HAS_KEY_FIELD_ID)) // Map has key field
{
    keyFieldId = in.getKeyFieldId();
}
if (in.has(OMMMap.HAS_DATA_DEFINITIONS)) // Map has data definitions
{
    // decode data definitions
}
if (in.has(OMMMap.HAS_SUMMARY_DATA)) // Map has summary data
{
    // decode summary data
}

// decode entries
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMMapEntry ommEntry = (OMMMapEntry) iter.next();
    decodeMapEntry(ommEntry); // application method
}
```

#### Example 68: Decoding Map

#### 10.4.6.2.2 Decoding Map Entry

The entries retrieved using a Map iterator are casted to an `OMMMapEntry` object.

A map entry may contain an action. If the `getAction()` method returns an action defined by `OMMMapEntry`, the decoding application should apply the action to the map entry. Section 10.7 summarizes dataformats actions.

If any map entry contains permission data, the `OMMMap.HAS_PERMISSION_DATA_PER_ENTRY` flag is set in the Map. Decoding of permission data is described in Section 10.5.2.

The key entry is retrieved by the `getKey()` method. This method returns an `OMMData` object. This object can be cast to the type identified as a Key Data Type, which is a Map property. The type can be also retrieved by calling `getType()` method on `OMMData`.

The data object is retrieved from the map entry using `getData()` after ensuring that the action is not a `DELETE` action.

The code example below shows decoding of Map entries.

```
// "in" is OMMMap
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMMapEntry ommEntry = (OMMMapEntry) iter.next();

    byte action;
    if (ommEntry.getAction() != 0)    // Map entry has action
        action = ommEntry.getAction();
    if (in.has(OMMMap.HASPERMISSION_DATA_PER_ENTRY)    // Map entry may have permission data
        if (ommEntry.has(OMMMapEntry.HAS_PERMISSION_DATA))    // Map entry has permission data
            decodePermissionData(ommEntry.getPermissionData());    // application implemented
    OMMData key = ommEntry.getKey();
    decodeKey(key);    // application method
    if (ommEntry.getAction() != OMMMapEntry.Action.DELETE){
    OMMData data = ommEntry.getData();
        decodeData(data);    // application method
    }
}
```

### Example 69: Decoding Map Entry

## 10.4.7 Series

A **Series** is a container of entries occurring in a sequence. The entry is known as a **series entry**. The entry has no identifier but rather is dependent on the implicit order within the container. Operations on entries are not supported since there is no way of entry identification.

The dataformat for each series entry is identified once in the series header (i.e., all series entries must be the same dataformat). Series optionally contain summary data that applies to the entire structure. Data definitions can also optionally be defined when series entries contain repetitive characteristics.

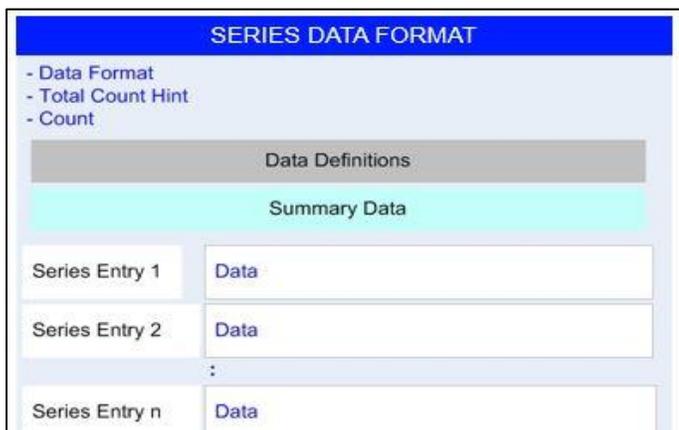


Figure 42: Series Dataformat

### 10.4.7.1 Encoding Series

The Series contains a header and series entries. The entries are of the same type, and can be primitive data or dataformats. The header may contain summary data, data definitions, and a number of series entries.

#### 10.4.7.1.1 Generic Steps

The general process for encoding a Series is shown below:

- Initialization of Series with flags, data type, and total count of entries. The data type applies to entries and summary data.
- Optionally, data definitions encoding. The data definitions should be encoded before summary data.
- Optionally, summary data encoding; this is required only if summary data is available. The summary data may use data definitions encoded in the previous step. Summary data must be of the same type as entries.
- Encoding series entries – the following is done for each entry.
- Encoding the series entry header.
- Encoding the data contained inside the entry.
- Completing series encoding.

#### 10.4.7.1.2 Initializing Series

The Series is initialized using the `encodeSeriesInit()` interface and expects the flags, series entries data type and entries number, as input arguments.

The Series supports the following flags. Multiple flags can be set by combining them with the Java `OR` operator.

- `HAS_DATA_DEFINITIONS`: indicates the presence of data definitions. If this is set, the data definitions should be encoded and the series entries should be either `OMMTypes.ELEMENT_LIST` or `OMMTypes.FIELD_LIST`.
- `HAS_SUMMARY_DATA`: indicates the presence of summary data.
- `HAS_TOTAL_COUNT_HINT`: indicates the presence of number of entries data.

The Series encoding initialization is shown in the example below. In this example, the series has data definitions, summary data, and has 3 elements. The elements are of `OMMTypes.FIELD_LIST` type.

```
// Series encoding initialization.
int flags = OMMSeries.HAS_DATA_DEFINITIONS | OMMSeries.HAS_TOTAL_COUNT_HINT | OMMSeries.HAS_SUMMARY_DATA;
short dataType = OMMTypes.FIELD_LIST;
short count = 3;
ommEncoder.encodeSeriesInit(flags, dataType, count);
```

#### Example 70: Series Initialization

#### 10.4.7.1.3 Encoding Data Definitions

Data definitions encoding is described in Section 10.8.2. The availability of data definitions must be indicated by setting the flag to `HAS_DATA_DEFINITIONS` when the Series is initialized.

#### 10.4.7.1.4 Encoding Summary Data

Summary data encoding is described in Section 10.6.1. The presence of summary data must be indicated by setting the hint flag to `HAS_SUMMARY_DATA` when the Series is initialized.

#### 10.4.7.1.5 Encoding Series Entry

The `encodeSeriesEntryInit()` method is used to initialize a vector entry. This method does not take any parameters, but marks the beginning of a series entry in encoder.

The data encoding depends on the data type.

### 10.4.7.1.6 Completing Series Encoding

The `encodeAggregateComplete()` method is used to indicate completion of the series encoding.

### 10.4.7.1.7 Series Encoding Example

In the following example the series has summary data, defined data, and has one element list entry. The element list uses data definition from the series.

```
// Initialize series that has defined data, summary data, the data type is element list and contains one
// entry
int flags = OMMSeries.HAS_DATA_DEFINITIONS | OMMSeries.HAS_TOTAL_COUNT_HINT | OMMSeries.HAS_SUMMARY_DATA;
short dataType = OMMTypes.ELEMENT_LIST;
short count = 1;
ommEncoder.encodeSeriesInit(flags, dataType, count);

// encode data definition from dataDefDictionary, data definition ID of 0
ommEncoder.encodeDataDefsInit();
DataDefDictionary.encodeDataDef(dictionary, ommEncoder, (short)0);
ommEncoder.encodeDataDefsComplete();

// encode summary data
ommEncoder.encodeSummaryDataInit();
ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
ommEncoder.encodeElementEntryInit("Version", OMMTypes.BUFFER);
bytes [] versionBytes = {01, 01};
ommEncoder.encodeBytes(versionBytes);
ommEncoder.encodeElementEntryInit(RDMDictionary.Summary.Type, OMMTypes.UINT);
ommEncoder.encodeUInt((long)RDMDictionary.Type.FIELD_DEFINITIONS);
ommEncoder.encodeAggregateComplete() // complete element list

// encode entry
FidDef def = dictionary.getFidDef(0); // the parameter is the data definition ID
ommEncoder.encodeSeriesEntryInit();
ommEncoder.encodeElementListInit(OMMElementList.HAS_DEFINED_DATA | OMMElementList.HAS _DATA_DEF_ID, (short)0,
(short)0);
ommEncoder.encodeString(def.getName(), OMMTypes.ASCII_STRING);
ommEncoder.encodeInt((long)def.getFieldId());
ommEncoder.encodeInt((long)def.getRippleFieldId());
ommEncoder.encodeAggregateComplete(); // complete element list

// Complete series encoding
ommEncoder.encodeAggregateComplete();
```

#### Example 71: Encoding Series

### 10.4.7.2 Decoding Series

Series properties are extracted using `getXXX()` methods. In the example below, the method `getCount()` retrieves the number of entries. The presence of total count hint is verified using the `OMMSeries.HAS_TOTAL_COUNT_HINT` flag. If available, the number is extracted using `getTotalCountHint()`.

A Series may contain data definitions. The presence of data definitions is identified by the `OMMSeries.HAS_DATA_DEFINITIONS` flag. The data definitions decoding is described in Section 10.8.3.

The presence of summary data is verified using `OMMSeries.HAS_SUMMARY_DATA` flag. Summary data decoding is described in Section 10.6.2.

The data type of the summary data and entries can be found using the `getDataType()` method.

The series entries are retrieved using the series iterator.

```
// "in" is an OMMSeries
int numberEntries = in.getCount(); // number of entries contained in the "in" series
```

```

int totalCountHint;
if (in.has(OMMSeries.HAS_TOTAL_COUNT_HINT)) // Series has total count hint
{
    totalCountHint = in.getTotalCountHint();
}
if (in.has(OMMSeries.HAS_DATA_DEFINITIONS)) // Series has data definitions
{
    // decode data definitions
}
if (in.has(OMMSeries.HAS_SUMMARY_DATA)) // Series has summary data
{
    // decode summary data
}

// decode entries
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMSeriesEntry ommEntry = (OMMSeriesEntry) iter.next();
    decodeSeriesEntry(ommEntry); // application method
}

```

### Example 72: Decoding Series

#### 10.4.7.3 Decoding Entry

There is no object representing a series entry. Instead, the `OMMEntry` object is used.

The data object is retrieved from the entry using the `getData()` method.

The code example below shows decoding of entries from the series.

```

// "in" is OMMSeries
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMEntry ommEntry = iter.next();
    OMMData data = ommEntry.getData();
    decodeData(data); // application method
}

```

### Example 73: Decoding Series Entry

## 10.4.8 FilterList

A *FilterList* is a container of loosely coupled entries each known as a *filter entry*.

The entry identifier is a numeric known as a *filter ID*.

Entries in the FilterList can be set, updated or cleared, and can optionally each have a separate permissions expression. Even though the dataformat for each filter entry data is identified in the FilterList header, filter entries can optionally define different dataformats per entry.

FILTER LIST DATA FORMAT		
- Data Format		
- Total Count Hint		
- Count		
Filter Entry 1	Id 1	Data
Filter Entry 2	Id 2	Data
:		
Filter Entry n	Id 31	Data

**Figure 43: FilterList Dataformat**

### 10.4.8.1 Encoding FilterList

The Filter List contains header and filter list entries. The entries can be of various types. The header may contain the number of entries.

#### 10.4.8.1.1 Generic Steps

The general process for encoding a filter list is shown below:

- Initialization of filter list with flags, data type, and total count of entries.
- Encoding filter list entries – the following is done for each entry
- Encoding the filter list entry header.
- Encoding the data contained inside the entry.
- Completing filter list encoding.

#### 10.4.8.1.2 Initializing Filter List

The filter list is initialized using the `encodeFilterListInit()` method, and expects the flags, filter list entries data type, and number of entries as input arguments. Multiple flags can be set by combining them with the Java **OR** operator.

The filter list supports the following flags:

- `HAS_PERMISSION_DATA_PER_ENTRY`: indicates that some or all entries have permission data.
- `HAS_TOTAL_COUNT_HINT`: indicates presence of number of entries data.

The filter list encoding initialization is shown in the example below. In this example the vector has permission data and has 4 elements. The elements are of `OMMTypes.ELEMENT_LIST` type.

```
// Filter List encoding initialization.
int flags = OMMFilterList.HAS_PERMISSION_DATA | OMMFilterList.HAS_TOTAL_COUNT_HINT;
short dataType = OMMTypes.ELEMENT_LIST;
short count = 4;
ommEncoder.encodeFilterListInit(flags, dataType, count);
```

#### Example 74: Filter List Initialization

#### 10.4.8.1.3 Encoding Filter List Entry

The `encodeFilterEntryInit()` method is used to initialize a filter list entry. The entry contains a header and data. The header contains a filter ID and may contain action and permission data. The supported actions are CLEAR, SET, UPDATE. If the entry contains permission data, the `OMMFilterEntry.HAS_PERMISSION_DATA` flag is set. If the type of the entry is different then the one specified in filter list, the entry has a type and an `OMMFilterEntry.HAS_DATA_FORMAT` flag set.

The code example below demonstrates initializing encoding of a filter list entry. In this example the entry has permission data, has a different type then specified in the filter list and has an action UPDATE.

```
int flags = OMMFilterEntry.HAS_PERMISSION_DATA | OMMFilterEntry.HAS_DATA_FORMAT;
short type = OMMTypes.FIELD_LIST;
int filterId = 2;
byte action = OMMFilterEntry.Action.UPDATE;
byte [] permissionExpression = { 0x2A, 0x67, 0xBC, 0x3B };
ommEncoder.encodeFilterEntryInit(flags, action, filterId, type, permissionExpression); // initialize field
entry
```

#### Example 75: Encoding Filter Entry Header

The data encoding depends on the data type.

#### 10.4.8.1.4 Completing Filter List Encoding

The `encodeAggregateComplete()` is used to indicate completion of the filter list encoding.

#### 10.4.8.2 Filter List Encoding Example

In the following example, the filter list contains two element list entries. One of the entries has permission data.

```
// Filter List encoding initialization.
int flags = OMMFilterList.HAS_TOTAL_COUNT_HINT | OMMFilterList.HAS_PERMISSION_DATA;
short dataType = OMMTypes.ELEMENT_LIST;
short count = 2;
ommEncoder.encodeFilterListInit(flags, dataType, count);

// The first entry is ElementList with permission data.
int flags1 = OMMFilterEntry.HAS_PERMISSION_DATA;
byte [] permData = {12, 34, 56 };
ommEncoder.encodeFilterEntryInit(flags1, OMMFilterEntry.Action.SET, 1, OMMTypes.ELEMENT_LIST, permData);
encodeElementList();          // application implemented method

// The second entry is a FieldList without permission data. The data type is different than the one specified in
// FilterList initialization, so the HAS_DATA_FORMAT flag is set.
int flags2 = OMMFilterEntry.HAS_DATA_FORMAT;
ommEncoder.encodeFilterEntryInit(flags2, OMMFilterEntry.Action.SET, 1, OMMTypes.FIELD_LIST, null);
ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
encodeFieldList();           // application implemented method

// Complete Filter List encoding
ommEncoder.encodeAggregateComplete();
```

#### Example 76: Encoding Filter List

### 10.4.8.3 Decoding Filter List

Filter list properties are accessed using `getXXX()` methods. In the example below, the methods `getCount()` and `getKeyDataType()` are used to retrieve the count and key data type.

The presence of total count hint is verified using `OMMFilterList.HAS_TOTAL_COUNT_HINT` flag. If available, the number is extracted using `getTotalCountHint()`.

If any filter entry contains permission data, the `OMMFilterList.HAS_PERMISSION_DATA_PER_ENTRY` flag is set in the filter list. For details on decoding permission data, refer to Section 10.5.2.

The filter entries are retrieved using the filter list Iterator.

```
// "in" is an OMMFilterList
int numberEntries = in.getCount();    // number of entries contained in the "in" filter list
int totalCountHint;
if (in.has(OMMFilterList.HAS_TOTAL_COUNT_HINT))    // Filter List has total count hint
{
    totalCountHint = in.getTotalCountHint();
}
// decode entries
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMFilterEntry ommEntry = (OMMFilterEntry) iter.next();
    decodeFilterEntry(ommEntry);    // application method
}
```

#### Example 77: Decoding Filter List

### 10.4.8.4 Decoding Filter Entry

The entries retrieved using a Filter List Iterator are cast to an `OMMFilterEntry` object.

If any filter entry contains permission data, the `OMMFilterList.HAS_PERMISSION_DATA_PER_ENTRY` flag is set in the Filter List. Decoding of permission data is described in Section 10.5.2.

A filter entry may contain an action. If the `getAction()` method returns an action defined by `OMMFilterEntry`, the decoding application should apply the action to the filter entry. Section 10.7 summarizes dataformats actions.

The data object is retrieved from the filter entry using `getData()` method. The entry may have data type different then the type defined in the filter list. If the type is different, the `OMMFilterEntry.HAS_DATA_FORMAT` flag is set. The flag can not be set if the action is `CLEAR`.

The code example below shows decoding of Filter entries:

```
// "in" is OMMFilterList
for (Iterator iter = in.iterator(); iter.hasNext(); )
{
    OMMFilterEntry ommEntry = (OMMFilterEntry) iter.next();

    byte action;
    if (ommEntry.getAction() != 0)    // Filter entry has action
        action = ommEntry.getAction();
    if (in.has(OMMFilterList.HASPERMISSION_DATA_PER_ENTRY)    // Filter entry may have permission data
        if (ommEntry.has(OMMFilterEntry.HAS_PERMISSION_DATA))    // Filter entry has permission data
        OMMData data = ommEntry.getData();
        decodeData(data);    // application method
    }
}
```

### Example 77: Decoding Filter Entry

## 10.5 Permission Data

**Permission Data** is optional authorization information. The Refinitiv Data Access Control System Lock API provides functionality for creation and manipulation of permissioning information. See 6 *Refinitiv Data Access Control System LOCK API Reference Manual* for more information about Refinitiv Data Access Control System usage and permission data creation.

Some OMM Messages allow for permission data to be specified. When permission data is included in a refresh response message or an status response message, this generally defines the authorization information associated with all content on the stream. Permission data can be changed on an existing stream by sending a subsequent refresh or status messages containing the new permission data. When permission data is included in an update response message, this generally defines authorization information that only applies to that specific update.

Some container entries allow the specification of permission data. When a container entry includes permission data, it generally defines authorization information that only applies to that specific container entry. Specific usage and inclusion of permissioning information can be further defined within a domain model specification.

Permission data is typically included to ensure that only the entitled parties are allowed to consume and view the content.

When content flows through Refinitiv Real-Time Distribution System, any content a user is not permissioned to access will be filtered or restricted.

### 10.5.1 Encoding Permission Data

Some entries, such as filter entry, map entry, or vector entry may contain permission data. The permission data is encoded during an entry initialization. The OMMEncoder methods `encodeFilterEntryInit()`, `encodeMapEntryInit()`, and `encodeVectorEntryInit()` take a byte array parameter, which is used to pass the permission data. In addition to the permission data, the flags parameter should contain the `HAS_PERMISSION_DATA` flag and the dataformat should be initialized with the `HAS_PERMISSION_DATA_PER_ENTRY` flag. A code example in Section 10.4.5.1.5 shows encoding of the permission data for a vector entry.

### 10.5.2 Decoding Permission Data

The presence of permission data is identified by two flags, the `HAS_PERMISSION_DATA_PER_ENTRY` residing in the dataformat, and the `HAS_PERMISSION_DATA` flag residing in the entry. An entry contains permission data if both of the flags are set. The permission data, if available, is retrieved using `getPermissionData()` from an entry cast to `OMMFilterEntry`, `OMMMapEntry`, or `OMMVectorEntry`, as applicable. This method returns permission data as a byte array.

## 10.6 Summary Data

Summary data provides some of the dataformats (i.e., map, vector and series) a place to store a common information for all entries within the dataformat. This is typically a metadata that describes the entries in a dataformat. For example, summary data could specify the currency of each entry's price, or rules regarding the sorting of entries. Summary data is represented by an `OMMData` dataformat or primitive data.

### 10.6.1 Encoding Summary Data

The data format that contains summary data should have the `HAS_SUMMARY_DATA` flag set. Summary data is encoded as described below:

- Initialize summary data encoding using `encodeSummaryDataInit()`.
- Encode the data format contained by the summary data. The encoding process is the same as any data format encoding.
- Complete the encoding the dataformat with `encodeAggregateComplete()`.
- The example below shows encoding of summary data. In this example, the summary data is an `OMMElementList` container:

```
ommEncoder.encodeSummaryDataInit();
ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short) 0);
// encode the OMMElementList
...
ommEncoder.encodeAggregateComplete(); // ElementList
```

#### Example 78: Encoding Summary Data

### 10.6.2 Decoding Summary Data

The presence of summary data is indicated by the `HAS_SUMMARY_DATA` flag. If available, the summary data is obtained using `getSummaryData()`. This method returns `OMMData`. The code example below shows retrieving of summary data from an `OMMMap`.

```
if (aMap.has(OMMMap.HAS_SUMMARY_DATA))
{
    OMMData data = aMap.getSummaryData();
    decodeData(data); // application method
}
```

#### Example 79: Decoding Summary Data

## 10.7 Actions

Actions are elements of some OMM entries, such as `OMMFilterEntry`, `OMMMapEntry`, `OMMVectorEntry`. Actions are operations to manage change-processing rules on entries within a data format. A Provider may specify actions. A Consumer needs to comply with the actions.

The table below summarizes the available actions within OMM dataforms:

DATAFORM	ACTION
OMMFilterEntry	<code>OMMFilterEntry.Action.CLEAR</code> Action used to remove an entry.
	<code>OMMFilterEntry.Action.SET</code> Action used to add or replace an entry.
	<code>OMMFilterEntry.Action.UPDATE</code> Action used to update an entry's content.
OMMMapEntry	<code>OMMMapEntry.Action.ADD</code> Action used to add or replace an entry.
	<code>OMMMapEntry.Action.DELETE</code> Action used to remove an entry.
	<code>OMMMapEntry.Action.UPDATE</code> Action used to update an entry's content.
OMMVectorEntry	<code>OMMVectorEntry.Action.CLEAR</code> Action used to remove an entry.
	<code>OMMVectorEntry.Action.DELETE</code> Action used to remove an entry from a sorted vector.
	<code>OMMVectorEntry.Action.INSERT</code> Action used to add an entry to a sorted vector.
	<code>OMMVectorEntry.Action.SET</code> Action used to add or replace an entry.
	<code>OMMVectorEntry.Action.UPDATE</code> Action used to update an entry's content.

Table 69: Actions

### 10.7.1 Encoding Actions

An action is encoded during an entry initialization. The `encodeFilterEntryInit()`, `encodeMapEntryInit()`, and `encodeVectorEntryInit()` methods take a byte parameter that is used to pass the action. If this parameter is `0`, there is no action associated with this entry.

### 10.7.2 Decoding Actions

The `OMMFilterEntry`, `OMMMapEntry` and `OMMVectorEntry` implement the `getAction()` method. This method returns a byte, which is interpreted as an action related to the entry.

## 10.8 Data Definition

**Standard data** is data that unites the data content and its type in an entry. It is represented as one entity. Standard data is self-describing data and does not require data definitions.

**Defined data** is the data content without its type and requires data definitions. The defined primitive data types are similar to the non-defined primitive types, with some differences. While the non-defined primitive types data can be encoded as a variable number of bytes, most of the defined primitive types leverage a fixed length encoding. This fixed length encoding can help to reduce the total number of bytes required to contain the encoded primitive type. `OMMDataTypes` interface defines values between `64` and `127` as defined primitive types. These define fixed length encodings for many of the primitive types (e.g. `INT_1` is a one byte fixed length encoding of `INT`). The defined primitive data is used in data definitions. Both of the primitive data types: defined, and non-defined, are listed in Section 10.3.

A **Data Definition** can be used to define the contents of an `OMMFieldList` or an `OMMElementList`, allowing for additional optimizations to be performed. Use of a data definition can result in smaller encoded content by optimizing the use of repetitive type and length information. A data definition describing an `OMMFieldList` contains `fieldId` and type information specified in the order the contents should be arranged in the encoded field list. A data definition describing an `OMMElementList` contains element `name` and type information specified in the order the contents should be arranged in the encoded element list.

Use of a data definition is especially advantageous when sending repetitive content, such as an `OMMMap` containing `OMMFieldList` content in each entry. In this situation, a data definition can be provided as part of the `OMMMap`. The data definition is used to define the layout of the repetitive `OMMMapEntry`-contained field list information, such as `fieldId`. The entries that are defined by the data definition do not have header, since the type is defined by the data definition.

A data definition can contain defined primitive type data (i.e., `INT_1`), non-defined primitive type data (i.e., `INT`), and dataformats. However, the defined primitive data types such as `INT_1`, `TIME_3` which are length specified are used only, while encoding data definitions for defined data. The defined primitive data types are preferred in the data definitions over the non-defined type, for the reason that the defined primitive type can specify the optimal encoded length.

The dataformats `Map`, `Vector`, and `Series` optionally contain the data definition segment, while the dataformats `FieldList` and `ElementList` use data definitions to contain defined data. The concept of defined data is applicable only to the `FieldList` and `ElementList` dataformats. For the rest of the dataformats such as `Map`, `Vector` and `Series`, the entries always contains both the data type and the data content and can't use defined data.

If the dataformats `FieldList` and `ElementList` need to contain defined data, it must be ensured that some parent dataformat (such as `Map`, `Vector` or `Series`) in the hierarchy contains the data definitions corresponding to the defined data.

The dictionary package of RFA provides the `DataDef` class to support data definitions. The following table defines members of the `DataDef` object.

MEMBERS	ACCESSOR	DESCRIPTION
<code>dataDefId</code>	<code>getDataDefId()</code>	The identifier value associated with this data definition. Any list content that leverages this definition should have the ID matching this identifier. Only values 0 - 15 are valid for local set definition content. The default value is 0.
<code>type</code>	<code>getType()</code>	The type is either <code>OMMTypes.FIELD_LIST_DEF</code> , if this is defined data for the field list, or <code>OMMTypes.ELEMENT_LIST_DEF</code> , if this is defined data for the element list.
<code>entryDef</code>	<code>iterator()</code> , <code>getCount()</code>	An <code>ArrayList</code> of entry definitions. The subsequent elements of this <code>ArrayList</code> define how the subsequent entries from the list ( <code>FieldList</code> or <code>ElementList</code> ) should be encoded.

**Table 70: DataDef Object Members**

The entry definition depends whether the definition is for the `FieldEntry` or `ElementEntry` and can be `FieldEntryDef` or `ElementEntryDef` respectively. The dictionary package of RFA provides `ElementEntryDef` and `FieldEntryDef` class definitions.

The `FieldEntryDef` contains the following members:

- `fielded`: `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 values are defined by `Refinitiv`.
- `dataType`: `dataType` defines a type that this entry should be encoded or decoded as when using this `FieldEntryDef`.

The `FieldEntryDef` class provides `getFieldId()` and `getDataType()` accessor methods.

The `ElementEntryDef` contains the following members:

- `name`
- The `String` that identifies the element entry, known as `name`. This value is domain and usage specific.
- The `dataType` which defines a type that this entry should be encoded or decoded when using this `ElementEntryDef`.

The `ElementEntryDef` class provides `getName()` and `getDataType()` accessor methods.

The diagram below shows an example of using DataDef:



**Figure 44: Defined Data Example**

In this example, the Vector contains a data definition. The DataDef object has ID of **0** and the type `OMMTypes.ELEMENT_LIST_DEF`. It defines two element list entries. The first entry's name is "cba" and the type of data is `OMMTypes.INT_1`. The second entry's name is "abc" and the data type is `OMMTypes.ASCII_STRING`.

The Vector entries are element lists. The second vector entry uses defined data. It has the flag **HAS\_DEFINED\_DATA** set, and also has the defined data ID set to **0**. The DataDef with ID of **0** is the one described above. Thus the two first elements entries contained in this element list are respectively:

- Entry named "cba" with the content of one byte integer, value of 25
- Entry named "abc" with the content of ASCII string, value of "A6"

## 10.8.1 Data Definition Dictionary

Multiple definitions can be grouped together within a *Dictionary*. A dictionary can be helpful when the content leverages multiple data definitions - the dictionary provides an easy way to pass around all data definitions necessary for encoding or decoding information. For instance, an *OMMVector* may contain multiple data definitions via a dictionary. The content of each *OMMVectorEntry* might require a different definition from within the dictionary.

The dictionary package of RFA provides the *DataDefDictionary* class to support a dictionary with data definitions. The following table defines members of the *DataDefDictionary* object:

MEMBERS	ACCESSOR	DESCRIPTION
count	getCount()	The number of data definitions stored in this dictionary
type	getType()	The type is either <i>OMMTypes.FIELD_LIST_DEF_DB</i> , if this is defined data for the field list, or <i>OMMTypes.ELEMENT_LIST_DEF_DB</i> , if this is defined data for the element list.
defs	getDataDef(), putDataDef(), removeDataDef(),	An array of <i>DataDef</i> objects. The array may contain up to 265 definitions. Each definition is put into the array indexed by its ID.
iterator	iterator()	Iterator of all <i>DataDef</i> objects stored in this dictionary.

**Table 71: DataDefDictionary Object Members**

## 10.8.2 Encoding Data Definitions

Data definitions exist only for *FieldList* and *ElementList* dataformats, and respectively they are *FieldEntryDef* or *ElementEntryDef*. The dataformats *Map*, *Vector*, *Series* may contain the data definitions.

If the data definition needs to be used one time only, the application does not need to store the definition in a dictionary. Otherwise, the application may use a *DataDefDictionary* to store the definitions.

The steps of encoding a list of data definitions are shown below:

- Initialize data definition encoding using `encodeDataDefsInit()`.
- Initialize the list definition encoding for the particular dataformat (*FieldList* or *ElementList*) and pass the definition ID to the encode method. Example: for encoding *ElementList* definitions, use the `encodeElementListDefInit()`.
- Encode the entry definitions. In our example, the `encodeElementEntryDef()` is used to encode element entry definitions.
- Complete the definition encoding using `encodeListDefComplete()`.
- Repeat the previous steps for additional definitions.
- Complete the data definition encoding using `encodeDataDefsComplete()`.

The following example illustrates the data definitions and the encoding methods:

```
// start of data definitions encoding
ommEncoder.encodeDataDefsInit();

// encode definition1
ommEncoder.encodeElementListDefInit((short)0); // the ID of this DataDef is 0
ommEncoder.encodeElementEntryDef("NAME", OMMTypes.ASCII_STRING);
ommEncoder.    encodeElementEntryDef("FID", OMMTypes.INT_2);
:
ommEncoder.encodeListDefComplete();

// encode definition2
ommEncoder.encodeElementListDefInit((short)1); // the ID of this DataDef is 1
ommEncoder.encodeElementEntryDef("s2", OMMTypes.REAL);
:
ommEncoder.encodeListDefComplete();

ommEncoder.encodeDataDefsComplete();
```

### Example 80: Encoding Data Definitions

In the above example the data definitions list contains two definitions. The definitions have IDs 0 and 1.

If the list contains only one definition, it is convenient to use ID of 0. The encoding and decoding of the entries is simplified in this case.

If the data definitions are used repeatedly, it is recommended to store the definitions in a dictionary. The encoding of data definitions utilizing [DataDefDictionary](#) is described by the steps below. A [DataDefDictionary](#) stores either field list data definitions or element list data definitions, but not both.

The data definition, is created and stored in the [DataDefDictionary](#) as follows:

- Create the [DataDefDictionary](#) and pass in the type of the definition, i.e., [OMMTypes.FIELD\\_LIST\\_DEF\\_DB](#) or [OMMTypes.ELEMENT\\_LIST\\_DEF\\_DB](#).
- Create the data definition and add it to the dictionary using [DataDefDictionary.putDataDef\(\)](#). The data definition type must match the type of the dictionary.

The data definition created above is encoded as follows:

- Initialize the data definition encoding on the encoder object using [OMMEncoder.encodeDataDefsInit\(\)](#).
- Use a [DataDefDictionary.encodeDataDef\(\)](#) method passing in the data definition ID to be encoded.
- Repeat the previous step for each data definition that needs to be encoded.
- Complete the data definition encoding using [encodeDataDefsComplete\(\)](#).

Instead of encoding individual data definitions, all the data definitions contained in the dictionary can be encoded using [DataDefDictionary.encodeAllDataDefs\(\)](#).

The example below demonstrates utilizing DataDefDictionary for data definition:

```
// create data definition
dataDefDictionary = DataDefDictionary.create(OMMTypes.FIELD_LIST_DEF_DB);

DataDef dataDef = DataDef.create((short)1, OMMTypes.FIELD_LIST_DEF); // create field list data
                                                                    // definition with Id=1
dataDef.addDef((short)3427, OMMTypes.REAL); // ORDER_PRC "ORDER PRICE"
dataDef.addDef((short)3428, OMMTypes.ENUM); // ORDER_SIDE "ORDER SIDE"
dataDef.addDef((short)3429, OMMTypes.REAL); // ORDER_SIZE "ORDER SIZE"
dataDef.addDef((short)3855, OMMTypes.UINT); // QUOTIM_MS
_dataDefDictionary.putDataDef(dataDef); // add data definition to
                                                                    // dictionary

ommEncoder.encodeDataDefsInit();
dataDefDictionary.encodeDataDef(_dataDefDictionary, ommEncoder, (short)1); // encode the above
                                                                    // definition with Id=1
```

### Example 81: Encoding Data Definitions using DataDefDictionary

## 10.8.3 Decoding Data Definitions

The OMMMap, OMMVector and OMMSeries implement the `getDataDefs()` method that returns an OMMDataDefs object. An application decoding OMMMap or OMMVector or OMMSeries should check whether the Data Definitions are contained in the dataformat by checking the `HAS_DATA_DEFINITIONS` flag.

To retrieve the DataDef object with a specified ID from OMMDataDefs the application should follow the steps below:

- Create the DataDefDictionary, using `create()` method.
- Decode the data definitions into the dictionary, using the static `decodeOMMDataDefs()` method.
- Retrieving the Data definition with a specified ID from the dictionary, using `getDataDef()` method.

The code example below shows how to retrieve a DataDef object that has been used for encoding an OMMElementList. In this example, it is assumed that the parent dataformat is an OMMSeries:

```
OMMDataDefs datadefs = series.getDataDefs(); // series is parent dataformat of OMMSeries type
short dbtype = series.getDataType(); // the type is either OMMTypes.FIELD_LIST_DEF_DB
                                                                    // or OMMTypes.ELEMENT_LIST_DEF_DB

DataDefDictionary dictionary = DataDefDictionary.create(dbtype);
DataDefDictionary.decodeOMMDataDefs(dictionary, datadefs);
DataDef datadef = dictionary.getDataDef(ID); // the DataDef with the specified ID
```

### Example 82: Decoding Data Definitions

## 10.9 Complex Data

Several OMM data types, such as OMMItemGroup, OMMPriority and OMMQosReq, do not implement the OMMData interface. These objects represent message header elements. They are set on an OMMMsg object via set methods. Correspondingly, the elements are retrieved by OMMMsg get methods.

### 10.9.1 OMMItemGroup

The OMMItemGroup object represents a group of items within a service that may change state together. Sometimes a service is created by merging data from multiple providers. Item groups are used to efficiently update the state of many item streams that originate from the same provider. Using a single report that represents an entire group of items changed state is more efficient than using multiple status messages for each of the items.

Each open item stream belongs to an item group. The item group affiliation is set by the provider in the initial refresh response message. The group can be modified by the provider via a status response message, another refresh response message, or a source directory update response message.

The item group is represented by a byte array. Typically the provider assigns a series of two byte integer in network byte order. This is not enforced, so the application should handle generally byte arrays.

The table below summarizes methods supported by an `OMMItemGroup` object:

METHOD	DESCRIPTION
<code>create()</code>	Creates an instance of <code>OMMItemGroup</code> . The method takes the byte array that contains the initial bytes. An overloaded method takes an integer and converts it to the bytes. The second overloaded method takes an instance of <code>OMMItemGroup</code> and an integer. The method creates a new instance with the bytes from the first parameter and bytes obtained from the integer parameter.
<code>equals()</code>	This method returns true if the bytes in the two instances are the same.
<code>getBytes()</code>	Returns a byte array that contains bytes.
<code>hashCode()</code>	Returns a hash code of the byte array.
<code>toString()</code>	String representation of the bytes as hex characters.

**Table 72: OMMItemGroup Methods**

## 10.9.2 OMMPriority

A consumer uses `OMMPriority` to notify provider of the preferred execution of the preemption algorithm. If more items are requested than the provider allows, either the new stream is closed or an existing stream is preempted. The provider should make the decision based on the priority of the streams. The stream with the lowest priority should be preempted. The `OMMPriority` object contains the following elements:

- `priorityClass`: This element defines importance of the stream. The highest priority indicates higher importance. The values can be from 0 to 10. Value of 0 indicates that the stream is open on consumer application, but is not watched (i.e., can be paused).
- `Count`: This element identifies how many users are using this stream. A greater count means greater priority, however this parameter compares within the same `priorityClass`. The `PriorityClass` takes precedence over the count.

The `OMMPriority` implements the `Comparable` interface, thus it has implemented the `compareTo()` and `equals()` methods. The class offers the accessor methods `getCount()` and `getPriorityClass()`, and overrides the methods `hashCode()` and `toString()`.

## 10.9.3 OMMQosReq

The `OMMQosReq` interface is an OMM adapter for the `QualityOfServiceRequest` interface. For a description of `QualityOfServiceRequest`, refer to Section 7.8.3.

# Chapter 11 Logging

## 11.1 Logging Overview

RFA Java Edition uses the Java Logging APIs to manage log events. Logging can be turned on and off at run time for each package. The Java Logging APIs can distribute the log events through handlers to files, the console, streams or network sockets. Loggers can be named using a hierarchical dot-separated namespace to allow partitioning of log messages. Logging levels are provided to filter messages selectively. For details of these features, refer to reference 4.

For the Logging Package overview, refer to Section 5.10.5.

## 11.2 Logger Concepts

RFA logging uses logging levels and namespaces to filter and partition the log messages.

CONCEPT	DESCRIPTION
Log levels	A measure of severity for a message, it provides control over which messages will be logged (written to a file, the console, etc.)
Namespace	A hierarchical, dot-separated naming convention aligned with the Java packaging namespace used to group and partition messages by their source

**Table 73: Logger Concepts**

## 11.3 Logging Levels

Logging levels are a measure of severity for messages. The user has the option to only display messages of a certain severity or higher. RFA Java uses four logging levels: **FINE**, **INFO**, **WARNING**, and **SEVERE**.

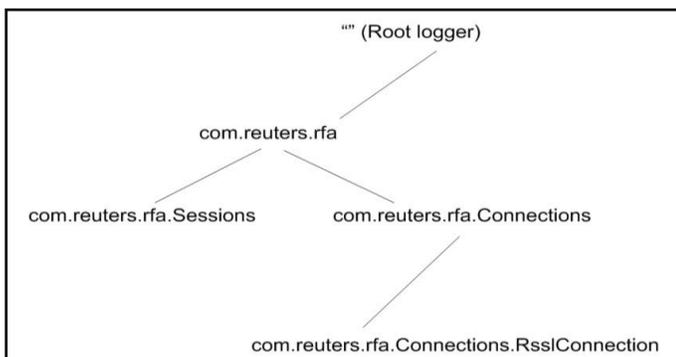
LOGGER LEVEL	DESCRIPTION
SEVERE	Used for catastrophic errors that require immediate attention
WARNING	Used for serious problems that must be noted
INFO	Used for normal informational messages; it is the default logging level
FINE	Used for detailed logging for debug purposes (i.e., it provides tracing information)

**Table 74: Logger Levels**

## 11.4 Logger Names and Hierarchy

The Java Logging API allows loggers to be named using a hierarchical dot-separated namespace. Each of the RFA Java packages and components has its own namespace, enabling end users to target which log messages they are interested in.

The following is an example of the logger hierarchy used by RFA:



**Figure 45: Logger Hierarchy**

A logger inherits the level and handler from its parents. If a logger has “null” or no explicit logging level, it uses the loglevel of its parents. If a logger publishes a log message to its own handlers, it will also publish to its parent’s handlers. (This action can be disabled using the logger’s `setUseParentHandlers()` method. See reference 5 for more details.)

All RFA Java loggers are under the `com.reuters.rfa` namespace hierarchy. If user registers a handler for `com.reuters.rfa`, they will receive RFA log messages from every package or component under RFA.

## 11.5 Logger Monitor

Users can monitor log messages from RFA components via loggers used by RFA. The Java Logging API `getLogger()` method provides users with a means to obtain a specific logger (see example below). The argument to `getLogger()` is the namespace of individual loggers used by RFA components.

A list of the complete namespace for RFA loggers is documented in an individual package’s Config and Logging guide. See reference 3 for all the namespace hierarchies used in the Session Layer package.

```
FileHandler fh = new FileHandler("./session.log"); //create a file Handler
fh.setLevel( Level.FINE ); //set the logging level to FINE (i.e. get tracing information)
Logger sessLogger = Logger.getLogger("com.reuters.rfa.session"); //obtain logger that receives messages
//pertaining to RFA Session Layer
sessLogger.addHandler( fh ); //add the file log Handler to receive messages
```

### Example 83: Create Logger and Specify Name

## 11.6 Resource Files

RFA Java uses the Java ResourceBundle facility to manage localization of log messages. Different versions of the properties file can be created to support different locales. A complete list of resource files is documented in the Config and Logging Guide (e.g. reference 3).

### 11.6.1 Resource File Naming Convention

Resource files are broken down by package or component. The naming convention is:

```
{Package/Component}Messages.properties
```

For example: `SessionMessages.properties`

### 11.6.2 Resource File Location

Resource files reside in the package `com.reuters.rfa.res`. Resource files are co-located with the RFA Java class files, so they are packaged into the corresponding JAR file under the path `/com/reuters/rfa/res`.

The Session Layer package utilizes the Java Logging API to manage log events. Users can monitor log messages from RFA components via loggers used by RFA. The Java Logging API `getLogger()` method provides users with a means to obtain a specific `Logger` used by the Session Layer. Subclasses of `java.util.logging.Handler` can be used to redirect log messages for each logger. These handlers can be activated and configured using the Java Logging API or using the JRE’s `logging.properties` file.

## 11.7 Namespaces of Loggers Used by the RFA Session Layer

The Session Layer components use several component Loggers. These namespaces can be used in the JRE’s `logging.properties` file or the Java Logging API to configure the logging of each component individually.

NAMESPACE	DESCRIPTION
<code>com.reuters.rfa.session</code>	The logger receives messages pertaining to Session Layer.
<code>com.reuters.rfa.connection</code>	The logger receives messages pertaining to all connections.
<code>com.reuters.rfa.config</code>	The logger receives messages pertaining to configuration.

**Table 75: Namespace of Session Layer Component Loggers**

Each connection and session is additionally scoped by the namespace and instance. So for example, the logger of `myNamespace::myConnection` can be configured in the `logging.properties` file with the following:

```
com.reuters.rfa.connection.rssl.myNamespace.myConnection.level= FINE
com.reuters.rfa.connection.rssl.myNamespace.myConnection.handlers= java.util.logging.ConsoleHandler
```

**NOTE:** '.' is used instead of '::' to separate `myNamespace` and `myConnection` in the logger name.

## 11.8 Resource Files (For Localization)

All resource files reside in the package `com.reuters.rfa.res` in `rfa.jar`. The following table lists the resource bundle files used by the Session Layer package.

FILE NAME	DESCRIPTION
SessionMessages.properties	Messages pertaining to sessions
ConnectionMessages.properties	Messages pertaining to connections
ConfigMessages.properties	Messages pertaining to the configuration

**Table 76: Names of Session Layer Resource Files**

By putting a different properties file in the `CLASSPATH`, the text for log events and some status events can be localized. Every resource key from the old properties file must be in the new properties file. The resource messages use the syntax specified by `java.text.MessageFormat`. The new messages must use the same number of arguments.

RFA tries to preserve text from the network whenever possible, so only text that is generated by RFA can be localized.

# Chapter 12 Basic Applications

## 12.1 Application Design

Applications that utilize the RFA API implement the following types of applications: OMM Consumer (abbreviated as “consumer”), OMM Interactive Provider (provider), OMM Non-Interactive Provider (non-interactive provider) and Hybrid. These application types have different purposes: consumer applications are written to (generally) obtain information, provider applications are written to (generally) publish information, and hybrid applications may assume both of these functions.

All the application types described above have a similar lifecycle:

- Initialization
- Functionality
- Cleanup

### 12.1.1 Initialization

All types of applications have to setup resources by initializing context, session, event sources, interest specifications, and event queues. After successfully completing common (to all application types) initialization tasks, applications continue the initialization process by executing application-type specific tasks. For example, a consumer application has to log into a remote provider, a provider application has to open a listening port, etc.

The initialization steps shown below apply to all application types. The type-specific steps are described in the sections that follow.

- Initialize Context

Initialize the RFA library by calling `Context.initialize()`. This initializes and manages the lifecycle of internal RFA threads and loads RFA packages into a single application.

- Initialize Session(s)

To initialize a Session, the application calls the static `Session.acquire()` method (which does not establish a connection).

- For a consumer application, the connection to a back-end system is established after sending a login request (refer to Section 12.2.3.1).
- For a non-interactive application, the connection with an ADH is established after sending a login request (refer to Section 0).
- In interactive provider applications, the port for incoming connections is opened when the application creates and registers an `OMMListenerIntSpec` interest specification (refer to Section 0).

A Session object is created with `Session.acquire()`, and should be destroyed with `Session.release()`.

- Set up Event Source(s)

Initialize Event Source(s) and register for events. The capabilities of the Event Source vary and are based on the type of applications (consumer or provider).

Event Sources, e.g. `OMMProvider`, are created with Session factory methods, e.g. `Session.createEventSource()`. They should be destroyed with the `destroy()` method on the EventSource, e.g. `OMMProvider.destroy()`.

- Create Interest Specifications

Interest Specifications are owned by the application. Interest Specification objects may be modified by the application and then reused on subsequent calls to the RFA API. No special cleanup is required.

- Create an Event Queue to send and receive messages to and from the API.

Event Queues are created with `EventQueue.create()`, and should be destroyed with `EventQueue.destroy()`. Event Queue Groups follow the same pattern.

### 12.1.2 An Overview of Application Type-Specific Functionality

The general functions of an application consist of requesting, sending, and receiving data, though the details will vary by application type:

- Consumer applications generally request data and process responses. The initial request messages sent by a consumer are classified as **administrative-type** requests, i.e., Login requests, Directory requests, and Dictionary requests. The Login

request is mandatory, and should be the first request message sent by the consumer application. Consumer applications may also send or receive generic messages, and may send post messages.

- Provider applications send responses on receiving requests from consumer applications. Provider applications may send or receive generic messages.
- Non-interactive providers send responses without receiving any request. Provider applications may send or receive generic messages.
- Hybrid applications implement the roles of both Consumer and Provider applications.

The messages that flow between applications are associated with event streams. A request message can establish an event stream, modify the stream, or close a stream. Response messages serve the stream (providing content), change the state of the stream, or close the stream, depending on the type of the message and data.

### 12.1.3 Cleanup

The cleanup process consists of cleaning up or deleting objects that were allocated during processing and initialization, and shutting down the application. An application should clean up as follows:

- Unregister clients for Item streams that are active.
- Destroy objects that have been created.
- For every call made to `Context::Uninitialize()`, make a corresponding call to the `Context::Uninitialize()` static method. This cleans up any resources used by RFA internally.

### 12.1.4 Design Considerations

#### 12.1.4.1 Input Parameters Scope

Instances of classes that are used as input parameters to the RFA (i.e., Interest Specifications and Commands) can be reused after the RFA operation has completed (e.g. after a call to `registerClient()` call returns). This is due to the fact that the RFA either makes a copy of the input class, or performs a network write within the RFA function call (e.g. `registerClient()`, `submit()`) before returning the control back to the application.

#### 12.1.4.2 Single Destroy and Release

Although the RFA packages serve different purposes, they share a common approach to defining and structuring their interfaces. The RFA interfaces can be split into two broad groups: **controller** interfaces and **entity** interfaces. The controller interfaces represent the system behavior and functionality, while the entity interfaces are mostly containers for other objects and primitive data types. In many respects, the controller type interfaces and the entity type interface are structured in a similar fashion, but there are some differences, described below:

- Controller interfaces

RFA controller interfaces are represented as abstract classes. As abstract classes, they cannot be instantiated, and they do not have public constructors, copy constructors, or assignment operators. Instead, these classes support instantiation of objects via static factory methods, like `create()` or `acquire()`.

The `create()` and `acquire()` methods supported by controller interfaces accept a String parameter. This parameter gives a name to the instantiated interface instance. If an application does not specify a name when instantiating an interface, the package implementation generates a unique name. All generated names are in the form of `<InterfaceName>.<SequenceNumber>` (e.g., `"Consumer.42"`).

The difference between the `create()` and `acquire()` methods is as follows:

- The interfaces that do not support instance sharing implement the `create()` method. In this case, instantiating an interface via the `create()` method with the same name will get two independent instances of the same interface. All interface classes that implement the `create()` method support the `destroy()` method. The purpose of the `destroy()` method is to tell RFA that the application does not need the corresponding object any more. To avoid resource leaks, an application must call `destroy()` for each `create()` call.
- The interfaces that support instance sharing implement the `acquire()` method. Using the same name in multiple `acquire()` calls will return multiple references to the same object. All interface classes that implement the `acquire()` method support the `release()` method. The release method tells the RFA that the application does not need the corresponding (shared) reference to the object any more. To avoid resource leaks, an application must call `release()` for each `acquire()` call. If an application has acquired the same instance of a shared interface multiple times (by calling the `acquire()` method with the same name more than once), then the shared instance will remain in use until the number of `release()` calls balances out the number of the `acquire()` calls.

- Entity Interfaces

Unlike the controller type interfaces, entity type interfaces do not have the `create()` and `acquire()` methods. Instead, an entity type object can be instantiated directly using the `new()` operator. For entity objects, whoever creates an entity objects owns it and is responsible for freeing up the resources when the object is no longer needed.

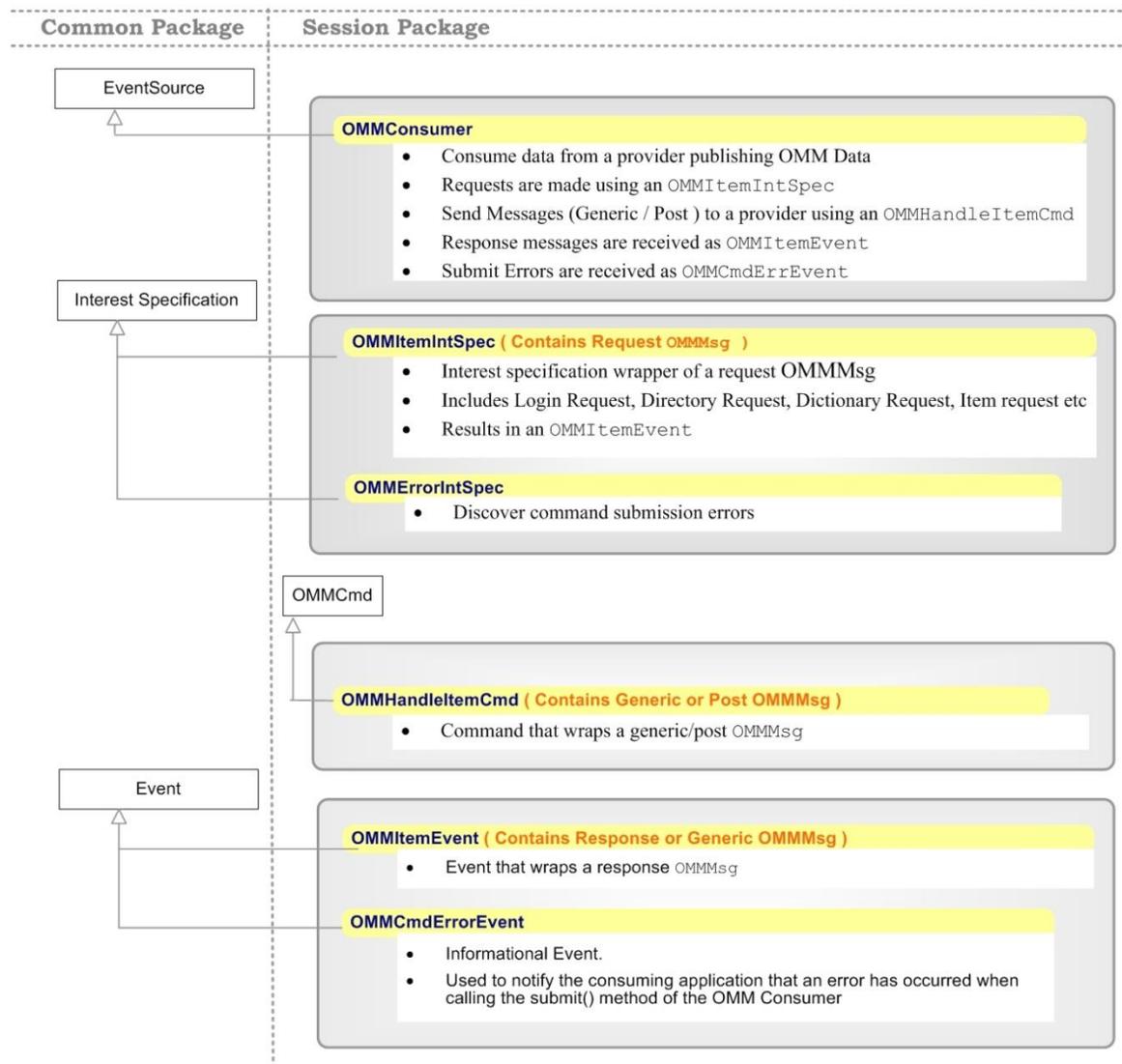
### 12.1.4.3 Event Distribution Using Multiple Threads

RFA fully supports event distribution using multiple thread contexts. However, RFA does not guarantee the order of the operations once multiple threads are used. For example; if two threads change the same event stream at the same time, the RFA does not guarantee the temporal ordering of these changes.

Relinquishing interest in a client from a thread other than the one that calls `dispatch`, or that initiated the operation, is also supported. If the application unregisters a client from a thread other than the dispatching thread, it needs to 'understand' that the dispatching thread may be using an application-defined Event Processing Client with the same interest that is being cancelled. The application should receive an Event indicating that the stream is closed (i.e., the `isEventStreamClosed()` method returns true) as a confirmation that the interest has been cancelled within the RFA.

## 12.2 OMM Consumer Applications

The following diagram shows the interfaces and classes used by an OMM Consumer application :



**Figure 46: Interfaces for Consuming Data**

OMM Consumer is an Event Source that proffers applications the ability to consume services that deliver market information and other information. An instance of OMMConsumer can be created using the `createEventSource()` method of the `session` interface. The resulting OMM Consumer is bound to the set of connections for which the particular Session is configured.

The `OMMConsumer` supports use of the `OMMItemIntSpec` Interest Specification, the `OMMItemEvent` Event, and the `OMMHandleItemCmd` Command. `OMMItemIntSpec` is a wrapper for request messages. An Interest Specification can be registered using the `registerClient()` or `reissueClient()` interfaces. The Event `OMMItemEvent` is a wrapper for response and generic messages and represents inbound responses delivered to the application.

`OMMHandleItemCmd` is a wrapper for outbound messages from the OMM Consumer. They are submitted (via `submit()`) on the `OMMConsumer` interface. After the OMM Consumer Event Source is set up, the consumer application must login by sending a Login Request using the `registerClient()` method.

`OMMConsumer` also supports:

- `OMMErrorIntSpec` Interest Specification and the `OMMCmdErrorEvent` Event.
- `OMMConnectionIntSpec` Interest Specification and the `OMMConnectionEvent` Event. For details, refer to Session Package Sections 8.2, 8.3.2, 8.4.3, and 8.6.

## 12.2.1 Consumer Application Tasks

The table below lists the various tasks a consumer application may execute. (Note: an application will not necessarily execute *all* the tasks shown.) The steps include establishing event streams, modifying existing event streams, sending post messages, and sending and receiving generic messages.

In this table, the interactions between a consumer application and RFA are shown. The  $\rightarrow$  arrow represents requests submitted by the consumer application to RFA, and the  $\leftarrow$  arrow represents the events delivered to the consumer application from RFA. The gray sections denote *what* will be accomplished, and the white sections denote *how* to accomplish them:

CONSUMER APPLICATION	
Startup (Refer to 12.2.2 for details.)	
<ul style="list-style-type: none"> <li>• Initialize context via <code>Context.initialize()</code></li> <li>• Create a session using the <code>Session</code> interface</li> <li>• Create an <code>OMMConsumer</code> event source</li> </ul>	→
Request OMM Data (LOGIN, DIRECTORY, MARKET_PRICE etc.) (Refer to 12.2.3 for details.)	→
<ul style="list-style-type: none"> <li>• Create an <code>OMMMsg</code> per the <i>RDM Usage Guide</i></li> <li>• Create an <code>OMMItemIntSpec</code> interest specification</li> <li>• Encapsulate an <code>OMMItemIntSpec</code> with an <code>OMMMsg</code></li> <li>• Invoke <code>registerClient()</code> on an <code>OMMConsumer</code> event source</li> </ul>	
Modify Existing Event Stream (Refer to 12.2.4, 12.2.5 for details.)	→
<ul style="list-style-type: none"> <li>• Create an <code>OMMMsg</code> as per the <i>RDM Usage Guide</i></li> <li>• Create an <code>OMMItemIntSpec</code> interest specification</li> <li>• Encapsulate an <code>OMMItemIntSpec</code> with <code>OMMMsg</code></li> <li>• Invoke <code>reissueClient()</code> with an existing handle on an <code>OMMConsumer</code> event source</li> </ul>	
Sending a generic / post <code>OMMMsg</code> (Refer to 12.2.5 for details.)	→
<ul style="list-style-type: none"> <li>• Create an <code>OMMMsg</code> as per the <i>RDM Usage Guide</i></li> <li>• Create an <code>OMMHandleItemCmd</code> command</li> <li>• Associate a handle of an open stream with the <code>OMMHandleItemCmd</code></li> <li>• Encapsulate an <code>OMMHandleItemCmd</code> with <code>OMMMsg</code></li> <li>• Invoke <code>submit()</code> on <code>OMMConsumer</code> event source</li> </ul>	
Process an event containing response or generic message (Refer to 12.2.6 for details.)	
<ul style="list-style-type: none"> <li>• Identify the <code>OMMItemEvent</code></li> <li>• Process the response or generic <code>OMMMsg</code> encapsulated in the event</li> </ul>	←
Register Interest in Submission errors and in OMM Connection events. (Refer to 12.2.7 and 12.2.9 for details.)	→
<ul style="list-style-type: none"> <li>• Create <code>OMMErrorIntSpec</code> interest specification</li> <li>• Invoke <code>registerClient()</code> on an <code>OMMConsumer</code> event source</li> <li>• Create <code>OMMConnectionIntSpec</code> interest specification</li> <li>• Invoke <code>registerClient()</code> on an <code>OMMConsumer</code> event source</li> </ul>	
Process an event containing a submission error event or an OMM Connection event (Refer to 12.2.8 and 12.2.10 for details.)	
<ul style="list-style-type: none"> <li>• Identify the <code>OMMCmdErrorEvent</code></li> <li>• Process the error using information from the <code>OMMCmdErrorEvent</code></li> <li>• Identify the <code>OMMConnectionEvent</code></li> <li>• Process the OMM Connection Event using information from the <code>OMMConnectionEvent</code></li> </ul>	←
Shutdown the application (Refer to 12.2.9 for details.)	→
<ul style="list-style-type: none"> <li>• Invoke <code>unregisterClient()</code> with an existing handle on an <code>OMMConsumer</code> event source</li> </ul>	

RFA

Table 77: Consumer Application Steps

## 12.2.2 Startup

A consumer application creates a session and then creates the OMM Consumer Event Source. To create an OMM Consumer, the application uses the Session and invokes the `createEventSource()` method, passing in the event source type `EventSource.OMM_CONSUMER` and a name for the consumer as shown in the following example:

```
// create the consumer session
Session _session = Session.acquire("consumerSession");

// Create an OMMConsumer event source
_ommConsumer = (OMMConsumer) _session.createEventSource(EventSource.OMM_CONSUMER, "myOMMConsumer", false);

// create the event queue to accept requests
EventQueue _eventQueue = EventQueue.create("Consumer Application EventQueue");
```

### Example 84: Setting Up a Consumer Application

The second parameter to the `createEventSource()` method is the name of the Event Source, which is used for identification purposes such as logging. The optional third parameter to `createEventSource()` specifies whether or not Completion Events should be sent. By default, Completion Events are not generated.

Finally, the application creates an Event Queue, from which the incoming events will be dispatched. The Event Queue has an assigned name for identification.

## 12.2.3 Sending Requests

In general, requests made by an OMM Consumer are issued through a single Interest Specification type, and responses are received in a single Event type. The application must initially login prior to requesting of information. The application can login by sending an OMM Data request and setting the message model to `RDMMsgTypes.LOGIN`. After successfully logging in the application may send other requests.

In general, to make a request, the OMM Consumer application needs to do the following:

- Create the message `OMMMsg` with the request information as described in the *RFA Java RDM Usage Guide*, for how to use the message models.
- Create an `OMMItemIntSpec` Interest Specification and populate it with the message using the `setMsg()` method. The Interest Specification essentially acts as a wrapper around the message.
- Invoke the `registerClient()` method on the OMM Consumer Event Source to send a request. The method takes the following arguments: the Interest Specification, a client whose callback will receive the responses, and the Closure.

The Interest Specification, message, and attribute objects may be reused for making subsequent requests. The application may reuse or change the objects at any time after calling `registerClient()`.

### 12.2.3.1 Login

The message model of the Login message is `RDMMsgTypes.LOGIN`.

The Attribute Information element must be set in the Login request message. The request attribute information ApplicationID and Position are encoded as an ElementList. The code example below illustrates encoding and sending a Login request message. Refer to the *RFA Java RDM Usage Guide* for a detailed description of the Login domain.

```
public void sendLoginRequest()
{
    OMMMsg ommmsg = encodeLoginReqMsg();
    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
    ommItemIntSpec.setMsg(ommmsg);
    Handle loginHandle = _ommConsumer.registerClient(_eventQueue, ommItemIntSpec, this, null);
}

private OMMMsg encodeLoginReqMsg()
{
    _ommEncoder.initialize(OMMTypes.MSG, 500);

    OMMMsg msg = _pool.acquireMsg();
```

```

msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setMsgModelType(RDMMsgTypes.LOGIN);
msg.setIndicationFlags(OMMMsg.Indication.REFRESH);
msg.setAttribInfo(null, _user, RDMUser.NameType.USER_NAME);

_ommEncoder.encodeMsgInit(msg, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA);
_ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short) 0);
_ommEncoder.encodeElementEntryInit("ApplicationId", OMMTypes.ASCII_STRING);
_ommEncoder.encodeString(_application, OMMTypes.ASCII_STRING);
_ommEncoder.encodeElementEntryInit("Position", OMMTypes.ASCII_STRING);
_ommEncoder.encodeString(_position, OMMTypes.ASCII_STRING);
_ommEncoder.encodeAggregateComplete();

//Get the encoded message from the _ommEncoder
OMMMsg encMsg = (OMMMsg)_ommEncoder.getEncodedObject();

//Release the message that is owned by the application
_pool.releaseMsg(msg);

return encMsg; //return the encoded message
}

```

### Example 85: Consumer Application requesting Login

#### 12.2.3.2 Requesting Service Directory

A Service Directory is internally (and automatically) requested by RFA for RFA consumer applications after a successful login. Until an application expresses interest in the details of a service directory, the services enumerated in it are known (internally) to RFA, but not to the application. An application may wish to discover what services are available from the infrastructure and receive subsequent updates on the status of said services. This is accomplished by specifying interest in Service Directory Events using the message model `RDMMsgTypes.DIRECTORY`.

By default, information about all available services is returned in an event. If an application wishes to make a request for information only pertaining to a specific service, it can invoke the `setServiceName()` method of the request attribute object `OMMAttribInfo`, or specify the service name in the `setAttribInfo()` method.

The `setFilter()` method of an `OMMAttribInfo` object can be used to specify the information that is required for each service. The bit mask values for the Service Filter are defined in the class `rfa.rdm.RDMService.Filter`. The data associated with each specified bit mask value is returned in a separate filter entry. The ServiceInfo filter entry contains an ElementList that contains the name and capabilities of the source. The ServiceState ElementList contains information related to the availability of the service.

Creating a Service Directory request does not require an encoder. The code fragment below shows an example of requesting a Service Directory.

The code fragment below shows an example of requesting a Service Directory. For a detailed description of the Directory domain, refer to the *RFA Java RDM Usage Guide*.

```

public void sendDirectoryRequest()
{
    OMMMsg ommmsg = encodeSrcDirReqMsg();
    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
    ommItemIntSpec.setMsg(ommmsg);
    Handle dirHandle = _ommConsumer.registerClient(_evenetQueue, ommItemIntSpec, this, null);
}

private OMMMsg encodeSrcDirReqMsg()
{
    OMMMsg msg = _pool.acquireMsg();
    //This application only needs a snapshot of the directory.
    msg.setMsgType(OMMMsg.MsgType.REQUEST);
    msg.setMsgModelType(RDMMsgTypes.DIRECTORY);
    msg.setIndicationFlags(OMMMsg.Indication.NONSTREAMING | OMMMsg.Indication.REFRESH);
}

```

```

OMMAttribInfo attribInfo = _pool.acquireAttribInfo();
//Specifies the filter information needed. See RDMUsageGuide.
attribInfo.setFilter( RDMSERVICE.Filter.INFO | RDMSERVICE.Filter.STATE);
msg.setAttribInfo(attribInfo);
return msg;

```

### Example 86: Consumer Application requesting Service Directory

#### 12.2.3.3 Requesting Dictionary

An application can request a dictionary using the message model `RdmMsgTypes.DICTIONARY`.

The name of the dictionary to be retrieved can be specified using the `setName()` method of the request attribute object `OMMAttribInfo`.

The `setFilter()` method of an `OMMAttribInfo` object can be used to specify the type of information required that is required for each service. The bit mask values for the Service Filter are defined in the Javadoc documentation for `RDMDictionary.Filter`.

The code below shows the process of creating a request a summary of the dictionary information (DictionaryType and version), by setting the filter to `RDMDictionary.Filter.INFO`. Note, this does not include the minimal data required to cache or convert data. For details on the Dictionary domain, refer to the *RFA Java RDM Usage Guide*.

```

public void openDictionaryInfo(String dictionaryName)
{
    OMMMsg msg = _pool.acquireMsg();
    msg.setMsgType(OMMMsg.MsgType.REQUEST);
    msg.setModelType(RDMMsgTypes.DICTIONARY);
    msg.setIndicationFlags(OMMMsg.Indication.NONSTREAMING | OMMMsg.Indication.REFRESH);

    OMMAttribInfo attribInfo = _pool.acquireAttribInfo();
    attribInfo.setName(dictionaryName);
    attribInfo.setServiceName(_serviceName);
    attribInfo.setFilter(RDMDictionary.Filter.INFO);
    msg.setAttribInfo(attribInfo);

    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
    ommItemIntSpec.setMsg(msg);
    Handle dictInfoHandle = _ommConsumer.registerClient(_eventQueue, ommItemIntSpec, this, null );

    _pool.releaseMsg(msg);
}

```

### Example 87: A Consumer Application Requesting Dictionary Info

The entire dictionary can be requested by setting the filter to `RDMDictionary.Filter.NORMAL`. This filter will return summary information, such as `DictionaryType` and version, plus all the dictionary data, except descriptions and comments:

```
public void openFullDictionary(String dictionaryName)
{
    OMMMsg msg = _pool.acquireMsg();
    msg.setMsgType(OMMMsg.MsgType.REQUEST);
    msg.setMsgModelType(RDMMsgTypes.DICTIONARY);
    msg.setIndicationFlags(OMMMsg.Indication.NONSTREAMING | OMMMsg.Indication.REFRESH);

    OMMAttribInfo attribInfo = _pool.acquireAttribInfo();
    attribInfo.setName(dictionaryName);
    attribInfo.setServiceName(_serviceName);
    attribInfo.setFilter(RDMDictionary.Filter.NORMAL);
    msg.setAttribInfo(attribInfo);

    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
    ommItemIntSpec.setMsg(msg);
    Handle dictFullHandle = _mainApp.getOMMConsumer().registerClient(_eventQueue, ommItemIntSpec, this,
                                                                    null );

    _pool.releaseMsg(msg);
}
```

### Example 88: A Consumer Application Requesting Full Dictionary

#### 12.2.3.4 Requesting Level 1 Market Information

After an application has initialized the Event Source and logged in, it typically requests market information. The code fragment below shows how to make a request for Level 1 market information using the message model `RDMMsgTypes.MARKET_PRICE`. The code sets the service name to `DIRECT_FEED`, the item name (identifier) to `TRI.N`, and the item type to `RDMInstrument.NameType.RIC` on the request attributes. The application then populates the Interest Specification, and calls `registerClient()`. Refer to *RFA Java RDM Usage Guide* for details on the Market Price domain.

```
public void sendRequest()
{
    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();

    OMMMsg ommmsg = _pool.acquireMsg();
    ommmsg.setMsgType(OMMMsg.MsgType.REQUEST);
    ommmsg.setMsgModelType(RDMMsgTypes.MARKET_PRICE);
    ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);
    ommmsg.setAttribInfo("DIRECT_FEED", "TRI.N", RDMInstrument.NameType.RIC);

    //Set the message in the interest spec
    ommItemIntSpec.setMsg(ommmsg);
    Handle itemHandle = _ommConsumer.registerClient(_eventQueue, ommItemIntSpec, this, null);

    _pool.releaseMsg(ommmsg);
}
```

### Example 89: A Consumer Application Requesting Level 1 Data

### 12.2.3.5 Requesting Level 2 Market Data

Requests for Level 2 market information are similar to the Level 1 market information requests shown above, except that the following message models are used. For details on the message models, refer to the *RFA Java RDM Usage Guide*.

MESSAGE MODEL TYPE	PURPOSE
MARKET_BY_ORDER	Provides access to a detailed order book for an instrument, containing individual instead of aggregated orders.
MARKET_BY_PRICE	Provides access to a summary order book for an instrument, showing aggregated volume and number of individual orders for different price points on the bid and ask side This message model is also known as Market Depth.
MARKET_MAKER	Provides access to each individual market maker's best bid/ask quotations for an instrument

**Table 78: Level 2 Requests**

## 12.2.4 Modifying Event Stream

Change requests can be issued only for existing open event streams using the corresponding handles held by an application. An application holds Handles associated with open event streams. The item stream may be reissued for the following reasons:

- Changing an item's priority
- Pausing/resuming the stream
- Changing a view
- Requesting an item refresh

In general, to modify an event stream, an application needs to do the following:

- Create a message `OMMMsg` with the request information as described in Section 6.15.
- For details on using message models, refer to the *RFA Java RDM Usage Guide*.
- Create or reuse an `OMMItemIntSpec` Interest Specification and populate it with the message using the `setMsg()` method. The Interest Specification essentially acts as a wrapper around the message.
- Invoke the `reissueClient()` method on the OMM Consumer Event Source to send a request. The handle of the open event stream and the request message are passed as input arguments.

The modification of event stream is demonstrated in the example below, where the stream priority is modified.

```
public void pauseAll()
{
    OMMMsg reissueMsg = getChangePriorityMsg();
    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
    ommItemIntSpec.setMsg(reissueMsg);
    _ommConsumer.reissueClient(_itemHandle, ommItemIntSpec);
}
private OMMMsg getChangePriorityMsg()
{
    OMMMsg msg = _pool.acquireMsg();
    msg.setMsgType(OMMMsg.MsgType.REQUEST);
    msg.setModelType(RDMMsgTypes.MARKET_PRICE);
    msg.setPriority((byte) 3, 1); // set new priority
    // to request a refresh set REFRESH indication flag on msg.
    ...
    return msg;
}
```

### Example 90: A Consumer Application Modifying a Stream

## 12.2.5 Sending Generic and Post Messages

The generic and post type messages are sent on open event streams. A stream is said to be open after a request message sent via `registerClient()` is answered with a refresh response message that has a stream state of 'open'. A consumer application receives a Handle for an open event stream. The application uses the handle to send the message. The steps application follows are shown below. Refer to Sections 13.5 and 13.6 for the description of Generic Messages and Posting features respectively.

- Create an `OMMMsg` message as described in Sections 13.5 and 13.6.
- Create a Command object: `OMMHandleItemCmd`.
- Encapsulate the Command object around the message using the `setMsg()` method. The `OMMHandleItemCmd` essentially acts as a wrapper around the message.
- Associate the handle of the stream with the `OMMHandleItemCmd` via the `setHandle()` method.
- Call the `submit()` method of the OMM Consumer Event Source, passing in the populated `OMMHandleItemCmd` and optionally passing in the closure object containing application specified data.

The code example below shows sending generic or post messages.

```
Handle _handle;
OMMMsg msg = createMsgToSend(...) // application method
OMMHandleItemCmd cmd = new OMMHandleItemCmd(); // Create a new OMMHandleItemCmd.

cmd.setMsg(msg) ;
cmd.setHandle(_handle); // active handle return by registerClient()
_ommConsumer.submit(cmd, null);
```

### Example 91: OMM Consumer Application Sending Generic / Post Messages

## 12.2.6 Handling Inbound Events

### 12.2.6.1 Receiving Inbound Events

Events are delivered to the application via an Event Queue. The Event Queue must be initialized during the start up of the application. Refer to Section 12.2.2 for more details.

Applications use the `dispatch()` method on Event Queue to retrieve Events. The dispatch method is called by an application, typically within a loop. An application alternatively may use a dispatch timer, or implement other methods to facilitate the dispatching of events from an Event Queue. A simple example of an application dispatching events is presented below:

```
Public void run()
{
    while(true)
    try
    {
        _eventQueue.dispatch( 1000 );
    }
    ...
}
```

### Example 92: Dispatching Events

In the example above, the application uses pooling to find whether there is an Event in the Event Queue. The number passed into the `dispatch()` method specifies how many milliseconds the EventQueue waits if there are no events.

In the call to `dispatch()`, if RFA has an event for an application to process, it will invoke the application's client `processEvent()` method. This method takes an Event as an argument.

An application written for a low-latency may not use an Event Queue, but use a call-back model instead. In this case the `processEvent()` method is called by the RFA (in it's own thread context) when the event is available to the application. For details on configuring low-latency application, refer to Section 7.2.

In response to interest in an OMM item, the application receives an `OMMItemEvent` in the client call-out. These events encapsulate the response message. Consumer applications need to retrieve response messages from the events and process them based on the message model type and message type. Generic messages are also encapsulated in the `OMMItemEvent`.

The consumer application needs to execute the following steps to process a response or generic message:

- Determine the concrete event type using the event's `getType()` method. The event type can be either a `COMPLETION_EVENT`, `OMM_ITEM_EVENT` or `OMM_CONNECTION_EVENT`.
  - The `COMPLETION_EVENT` notifies the application that the event stream is closed.
  - The `OMM_ITEM_EVENT` notifies the application of incoming responses or generic messages.
  - The `OMM_CONNECTION_EVENT` contains connection information.
- Cast the event of type `OMM_ITEM_EVENT` to an `OMMItemEvent`, a wrapper around the message.
- Obtain the response message by invoking the `getMsg()` method on the `OMMItemEvent`.
- Get the message model type of the response (e.g. `RDMMsgTypes.LOGIN`, `RDMMsgTypes.MARKET_PRICE`) using `getMsgModelType()`.
- Get the message type using the `getMsgType()` method.
- Parse the response or generic message data based on the message model. For details on message models, refer to 2 the *Java Reference Manual*.

The following example uses the application-implemented `processOMMItemEvent` method to process the response or generic message:

```
public void processEvent(com.reuters.rfa.common.Event event)
{
    switch (event.getType())
    {
        case Event.OMM_ITEM_EVENT :
            procesOMMItemEvent((OMMItemEvent) event);
            break;
        case Event.COMPLETION_EVENT:
            processCompletionEvent((event.getHandle()));
            break;
        case Event.CONNECTION_EVENT:
            processOMMConnectionEvent((OMMConnectionEvent) event);
            break;
        :
    }
}

protected void procesOMMItemEvent(OMMItemEvent e)
{
    OMMMsg respMsg = e.getMsg();

    byte msgType = respMsg.getMsgType();
    short msgModelType = respMsg.getMsgModelType();
    ...
}
```

### Example 93: A Consumer Application Processing Events

The OMM Item Event (and the encapsulated message) is only valid during the Client callback. If the application wishes to retain the message or data beyond the scope of the callback, it must make a copy of the message or data. Application can use the RFA-managed memory to save the message or data using the `OMMPool` or `acquireCopy()` method.

#### 12.2.6.2 Parsing OMM Message

The OMM messages contain a header and payload. Refer to Section 6.2.1 for message elements and accessor methods supported by the OMM interface. Decoding an OMM Message is described in Section 6.15.1.

#### 12.2.6.3 Processing Messages: Actions

Actions are elements of some OMM entries (refer to Table 69 for details). Actions are operations to manage change-processing rules on entries within a data format. A provider may specify actions. A consumer must to comply with the actions.

A consumer application should process the data it receives according to the actions when they are discovered during decoding. The actions are typically associated with the data formats that contain multiple entries, and are managed by an application. This type of data formats is usually used in responses to a streaming request. The actions reveal the change of state on Provider.

Actions may occur in a refresh message or update message. Consumer applications need to apply an action in or after an initial refresh message. This implies an update containing actions may occur prior to the final part of a multi-part refresh message. For example, when an 'add' action is received, consumer applications need to add an entry to their cache and apply any subsequent change such as an update, add or delete action. It is possible that multiple add actions may occur for the same item. In this case, consumer applications should ignore older updates.

#### 12.2.6.4 Processing Multi-part Messages

Refresh responses and generic messages may be single-part or multi-part.

The final refresh is indicated by the presence of an `OMMMsg.Indication.REFRESH_COMPLETE` flag, which is contained as part of the indication mask in the response message.

Refresh messages are guaranteed to be atomic for single part messages. Multi-part refresh messages may be interleaved with update messages and status messages before the final refresh is received.

In the following example, the application checks for the presence of the `OMMMsg.Indication.REFRESH_COMPLETE` flag using the `isSet()` method:

```
if (respMessage.isSet(OMMMsg.Indication.REFRESH_COMPLETE))
    boolean finalRefresh = true;
```

#### Example 94: Identifying the Final Refresh Message

The final generic message is indicated by the presence of the `OMMMsg.Indication.GENERIC_COMPLETE` flag contained as part of the indication mask in the generic message.

Generic messages are guaranteed to be atomic for single part messages. Multi-part generic messages may be interleaved with update messages and status messages, received on the same event stream before receiving the final generic message.

In the following example the application checks for the presence of the `OMMMsg.Indication.GENERIC_COMPLETE` flag using the `isSet()` method.

```
if (genericMessage.isSet(OMMMsg.Indication.GENERIC_COMPLETE))
    boolean finalGeneric = true;
```

#### Example 95: Identifying the Final Generic Message

#### 12.2.6.5 Processing Login Refresh Responses

A consumer application typically starts communicating with a server by requesting a Login. The consumer can use a Login request to request behaviors and determine capabilities when connecting to a system. RFA helps to manage differences between systems and will provide some behaviors that might be missing.

In response, the consumer application receives either a refresh or status message. The application should determine the Login result and if successful, proceed, or otherwise, take appropriate action (i.e., if the stream is **CLOSED**, application should clean up and exit).

The response to the Login request contains `OMMState` information (which conveys the result of the login attempt) and `AttribInfo` information (which conveys the behaviors and capabilities of RFA), as follows:

INFORMATION TYPE	DESCRIPTION	
OMMState	<ul style="list-style-type: none"> <li>Stream.OPEN</li> <li>Data.OK</li> <li>Code.NONE</li> </ul>	The Login has been accepted by the provider. The consumer application established the Login event stream.
	<ul style="list-style-type: none"> <li>Stream.OPEN</li> <li>Data.SUSPECT</li> </ul>	The connection is down or login is pending on all connections.
	<ul style="list-style-type: none"> <li>Stream.CLOSED</li> <li>Code. NOT_ENTITLED</li> </ul>	The Login failed. If the provider did not accept Login credentials, it will set the Code to <b>NOT_ENTITLED</b> .
	<ul style="list-style-type: none"> <li>Stream.CLOSED</li> <li>Code.?</li> </ul>	The Login can fail for other reasons: for example, a timeout occurred on the network while processing the request. In this case, the Code will be set accordingly. An application can determine how to proceed based on the Code. Refer to Table 65 for the Code definitions.

INFORMATION TYPE		DESCRIPTION
AttribInfo	SingleOpen	The login response always matches the client request, because RFA can provide the requested behavior whether or not the server supports it. If the client does not request these attributes, RFA indicates support for these behaviors. RFA can ensure proper behavior even if capabilities differ among providers or if a server does not support it.
	AllowSuspectData	
	SupportBatchRequests	RFA's login response always indicates support for Batch Request, Batch Reissue, and Batch Close. RFA can ensure proper behavior even if capabilities differ among providers or if a server does not support it.
	SupportOMMPost	RFA's login response will always indicate support for posting. If the server does not support posting, a warning message will be logged. If RFA receives a post, RFA sends a NACK message to the client application.
	SupportOptimizedPauseResume	RFA always indicates support for Optimized Pause and Resume. If the server does not support Optimized Pause and Resume, RFA logs a warning message and converts Optimized Pause and Resume requests to individual item Pause and Resume requests.
	SupportPauseResume	RFA always indicates support for Pause and Resume. If the server does not support Pause and Resume, RFA logs a warning message and ignores Pause/Resume requests.
	SupportViewRequests	RFA always indicates support for View Requests. If the server does not support View Requests, a warning message is logged, and RFA will convert View requests to full item requests.
AttribInfo (cont...)	SupportEnhancedSymbolList	RFA always indicates support for advanced symbol lists. If the server does not support advanced symbol lists, a warning message is logged. RFA will open data stream on behalf of the client
	ProvidePermissionExpressions	These attributes are set on the login response if they are available on the server response.
	ProvidePermissionProfile	

Table 79: Login Response Contents

### 12.2.6.6 Processing Item Refresh Responses

The initial refresh response contains an OMMState element. Subsequent refresh messages may or may not contain OMMState. The table below shows the possible values of the OMMState that an item stream can have:

OMMSTATE	DESCRIPTION
<ul style="list-style-type: none"> <li>Stream.OPEN</li> <li>Data.OK</li> <li>Code.NONE</li> </ul>	The item is served by the provider. The consumer application established the item event stream.
<ul style="list-style-type: none"> <li>If the SingleOpen attribute is set on this stream:</li> <li>Data.SUSPECT</li> <li>Stream.OPEN</li> <li>Code. NO_RESOURCES</li> </ul>	The provider does not offer data for the requested item at this time. However, the system will try to recover this item when available.
<ul style="list-style-type: none"> <li>If the SingleOpen attribute is not set on this stream:</li> <li>Data.SUSPECT</li> <li>Stream.CLOSED_RECOVER</li> <li>Code. NO_RESOURCES</li> </ul>	The provider does not offer data for the requested item at this time. The application can try to re-request the item later.
<ul style="list-style-type: none"> <li>Data.SUSPECT</li> <li>Stream.CLOSED</li> <li>code as applies</li> </ul>	The item is not open on the provider, and the application should close this stream.

**Table 80: OMMState in an Item Response**

An application typically manages the items that have an open stream and cleans up the resources, when the stream is closed. If the data is OK, the application retrieves the data obtained in this message and consumes it. If the data is SUSPECT, the application either waits for the status message indicating data OK, or closes the item and cleans up resources.

### 12.2.6.7 Processing Acknowledge Messages

The acknowledge message is optionally used as a response to post message. Refer to Section 13.6 for details.

### 12.2.6.8 Processing Update Response Messages

The update response message brings updates to the data contained in the refresh response or update response that was delivered previously. An application typically consumes this data.

### 12.2.6.9 Processing Status Response Messages

The status response message obtained on an item stream can originate from the provider or, or the message may be generated by RFA itself. For example, after submitting a login request, the very first login response an application receives will be generated by RFA, and the state will be "OPEN, SUSPECT, NONE". Assuming the provider is reachable on the network, and the login attempt is successful, the next login response message received by the application will originate from the provider, and the state may be "OPEN, OK, NONE".

If the response to the Login request is successful, RFA internally generates and sends the provider a Directory request message. This message returns information about services offered by the provider. For each service, the response includes the ServiceInfo, the service state, a range of supported Quality of Service, ect. For details on Directory domain messages, refer to the *RFA Java RDM Usage Guide*.

The received data is cached by the RFA. The provider may send updates if the services change. RFA aggregates the data and keeps the cache updated. If there is a change in the state of item stream, the RFA sends event messages for each item that is affected.

The table below lists the possible Source Directory responses, and the events the application receives from RFA as a result.

SOURCE DIRECTORY	EVENT
A new service is added, state is OK.	None
A service is deleted	Each stream using this service receives status response messages with the OMMState set as follows: <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.CLOSED</li> <li>• Code. NO_RESOURCES</li> </ul>
A service state changes from OK to DOWN	Each stream using this service receives a status response message with the OMMState set as follows: <p>If the SingleOpen attribute is set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.OPEN</li> <li>• Code. NO_RESOURCES</li> </ul> <p>If the SingleOpen attribute is not set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.CLOSED_RECOVER</li> <li>• Code. NO_RESOURCES</li> </ul>
A service state changes from DOWN to OK	RFA will attempt to recover each item stream that uses this service. If successful, the application will receive a refresh response message with the OMMState set as follows: <ul style="list-style-type: none"> <li>• Data.OK</li> <li>• Stream.OPEN</li> <li>• Code. NONE</li> </ul>
An item group is merged to another group and the service state did not change	Each stream that is merged to the new OMMGroup receives a status response message with the OMMGroup set to the new value.
An item group is merged to another group and the state of the “merged to” group changed from OK to DOWN	Each stream that is merged to the new OMMGroup receives a status response message with the OMMGroup set to the new value, and OMMState set as follows: <p>If the SingleOpen attribute is set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.OPEN</li> <li>• Code. NO_RESOURCES</li> </ul> <p>If the SingleOpen attribute is not set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.CLOSED_RECOVER</li> <li>• Code. NO_RESOURCES</li> </ul> <p>Also the items that belong to the “merged to” group receive status response message with the OMMState set the way.</p>
An item group is merged to another group and the state of the “merged to” group changed from DOWN to OK	Each stream that is merged to the new OMMGroup receives a status response message with the OMMGroup set to the new value. <p>RFA will attempt to recover each item stream that belongs to the “merged to” group. If successful, the application will receive a refresh response message with the OMMState set as follows:</p> <ul style="list-style-type: none"> <li>• Data.OK</li> <li>• Stream.OPEN</li> <li>• Code. NONE</li> </ul>

SOURCE DIRECTORY	EVENT
A service's QualityOfService attribute has downgraded	<p>Each stream that uses this service and requested a Quality of Service greater than the level currently offered by this service receives a status response message with the OMMState as follows:</p> <p>If the SingleOpen attribute is set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.OPEN</li> <li>• Code. NO_RESOURCES</li> </ul> <p>If the SingleOpen attribute is not set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.CLOSED_RECOVER</li> <li>• Code. NO_RESOURCES</li> </ul>

**Table 81: Events From a Source Directory Response**

The provider may send a status response message for an item. The status message may include an OMMState element. If OMMState is not included, it is assumed the state is good. The table below shows the possible values of the OMMState that an item stream can receive.

OMMSTATE	DESCRIPTION
<ul style="list-style-type: none"> <li>• Stream.OPEN</li> <li>• Data.OK</li> <li>• Code.NONE</li> </ul>	The item is served by provider. The consumer application established the item event stream.
<p>If the SingleOpen attribute is set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.OPEN</li> <li>• Code. NO_RESOURCES</li> </ul>	<p>The provider does not offer data for this item at this time. However the system will try to recover this item when available.</p> <p>The provider can send a message for an individual item, or can use the Directory domain to notify the application of service status changes (refer to Table 81). In the latter case, RFA issues the status messages for affected item streams.</p>
<p>If the SingleOpen attribute is not set on this stream:</p> <ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.CLOSED_RECOVER</li> <li>• Code. NO_RESOURCES</li> </ul>	<p>The provider does not offer data for this item at this time. The application can try to re-request the item later.</p> <p>The provider can send a message for an individual item, or can use Directory domain to notify of service status changes (refer to Table 81). In the latter case, RFA issues the status messages for affected item streams.</p>
<ul style="list-style-type: none"> <li>• Data.SUSPECT</li> <li>• Stream.CLOSED</li> <li>• code as applies</li> </ul>	<p>The item is not open on the provider and the application should close this stream.</p> <p>The provider can send a message for an individual item, or can use Directory domain to notify of service status (refer to Table 81). In the latter case, RFA issues the status messages for affected item streams.</p>

**Table 82: OMMState In an Item Response**

An application typically manages the items that have open streams and cleans up the resources, when the stream is closed. If data is SUSPECT, an application may either wait for a status message indicating the data is OK, or close the item and clean up resources. If the data is OK, the application retrieves the data obtained in the message and consumes it.

The status message may also bring an update to the Item Group the stream belongs to.

## 12.2.7 Registering Interest In Submission Error Events

The application sets up Error notifications by registering the `OMMErrorIntSpec` Interest Specification. This allows the application to receive Error events related to the `client`.

```
// Subscribe to error Event stream
OMMErrorIntSpec errorSpec = new OMMErrorIntSpec();
Handle errorHandle = _ommConsumer.registerClient(eventQueue, errorSpec, client, null);
```

### Example 96: An OMM Consumer Listening for Submission Errors

## 12.2.8 Handling Submission Error Events

A consumer application receives error notifications related to invocations of command `submit()` call via `OMMCmdErrorEvent` events. This event gives access to the command, command ID, error closure, submit closure, and status for the command that failed. These details can be accessed by invoking the methods `getCmd()`, `getCmdID()`, `getClosure()`, `getSubmitClosure()`, and `getStatus()` respectively..

```
protected void processOMMCmdErrorEvent(OMMCmdErrorEvent errorEvent)
{
    System.out.println ("Received OMMCmd ERROR EVENT for id: "
        + errorEvent.getCmdID() + " "
        + errorEvent.getStatus().getStatusText());
}
```

### Example 97: An OMM Consumer Handling Error Notifications

## 12.2.9 Registering Interest In OMM Connection Events

The application sets up OMM Connection notifications by registering the `OMMConnectionIntSpec` Interest Specification. This allows the application to receive OMM Connection events related to the `client`.

```
// Subscribe to OMM Connection events
OMMConnectionIntSpec connectionIntSpec = new OMMConnectionIntSpec();
Handle connectionInterestHandle = _ommConsumer.registerClient(eventQueue, connectionIntSpec, client, null);
```

### Example 99: An OMM Consumer Listening for OMM Connection Events

## 12.2.10 Handling OMM Connection Events

A consumer application receives OMM Connection notifications via `OMMConnectionEvent` events. This event gives access to the host name and port for the active connection and the client's component version for all the connected connections. These details can be accessed by invoking `getConnectedHostName()`, `getConnectedPort()`, and `getConnectedComponentVersion()`, respectively. Details for multicast connections can also be accessed by invoking `getConnectedRecvAddress()`, `getConnectedRecvPort()`, `getConnectedSendAddress()`, `getConnectedSendPort()`, and `getConnectedUnicastPort()`.

The application can determine the connected component version in use at either end of the connection by calling the `getConnectedComponentVersion()` method of the `OMMConnectionEvent`.

For warm standby failover, RFA notifies the application when the connection switches to a new active server through an OMM Connection, from which the application can determine the new active host name and port. The connection status will still have State UP, but the Status Code will be `SERVER_SWITCHED`. See also Section [Error! Reference source not found.](#)

```
protected void processOMMConnectionEvent(OMMConnectionEvent connectionEvent)
{
    System.out.println ("Received OMM_CONNECTION EVENT: "
        + "Name: " + connectionEvent.getConnectionName() + " "
        + "Status: " + connectionEvent.getConnectionStatus().toString() + " "
        + "Host: " + connectionEvent.getConnectedHostName() + " "
        + "Port: " + connectionEvent.getConnectedHostPort() + " "
        + "ComponentVersion: " + connectionEvent.getConnectedComponentVersion());
}
```

### Example 99: An OMM Consumer Handling OMM Connection Events

## 12.2.11 Registering Interest in OMM Connection Stats Events

The application sets up OMM connection statistic notifications by registering the `OMMConnectionStatsIntSpec` interest specification. The interface receives connection statistics events for a connection or a list of connections. Connection statistics are comprised of the bytes read and written on the wire provided by RFA, periodically over duration specified by the application when registering for this event.

The application can specify a connection name or a list of comma-separated connection names for which statistics are desired, by using the `OMMConnectionStatsIntSpec::setConnectionName()` method. The application can also specify the duration over which statistics are desired by using the `OMMConnectionStatsIntSpec::setStatsInterval()` method. If the application does not specify the duration on the interest specification, then RFA generates these events every second. If the application does not specify a connection name on its interest specification, then RFA generates these events for all the connections listed in the `Session\connectionList` parameter.

```
// Subscribe to OMM Connection statistic events
OMMConnectionStatsIntSpec connectionStatsIntSpec = new OMMConnectionStatsIntSpec();
Handle connectionStatsInterestHandle = _ommConsumer.registerClient(_eventQueue, connectionStatsIntSpec,
    _loginClient, null);
```

### Example 99: An OMM Consumer Registering Interest in OMM Connection Stats Events

## 12.2.12 Handling OMM Connection Stats Events

An application receives OMM Connection statistic notifications via `OMMConnectionStatsEvent` events. This event gives access to bytes read and written on the wire for the active connection.

```
protected void processOMMConnectionStatsEvent( OMMConnectionStatsEvent connectionStatsEvent)
{
    System.out.println ("Received Connection statistics Event: " + connectionStatsEvent.getConnectionName()
        + " on handle " + connectionStatsEvent.getHandle()
        + " with bytes Read = " + connectionStatsEvent .getBytesRead()
        + " and bytes written = " + connectionStatsEvent .getBytesWritten());
}
```

### Example 99: An OMM Consumer Handling OMM Connection Events

## 12.2.13 Shutting Down an Application

An application may close all streams and then destroy Event Source. Alternatively an application may just destroy Event Source, in which case the closing of the streams is handled by the RFA.

An open stream is closed by using `unregisterClient()` and passing in the handle of an open stream or `null`. The handle would have been previously obtained from a call to `registerClient()`.

### 12.2.13.1 Unregistering Interest In OMM Market Information

An application can close a single OMM Data event stream by calling the `unregisterClient()` method on the Event Source. This method accepts one parameter: the Handle returned from the `registerClient()` call.

It is important to note that if the event Stream had already been closed by RFA (which can be determined by invoking the `isEventStreamClosed()` method on an Event during Event processing), the application does not need to not call `unregisterClient()`. The event stream is already closed, and the handle is no longer valid. This rule applies to event streams from all message models.

```
// Unsubscribe for OMM Data Item Events
_ommConsumer.unregisterClient(itemHandle);
```

#### Example 98: A Consumer Closing an Item

### 12.2.13.2 Unregister Interest in OMMErrorIntSpec

The following code closes an event stream opened for error Events resulting from `submit()` failures.

```
// Unsubscribe for OMMErrorIntSpec
_ommConsumer.unregisterClient(errorHandle);
```

#### Example 99: A Consumer Closing an OMMErrorIntSpec

### 12.2.13.3 Unregister Interest in OMMConnectionIntSpec

The following code closes an event stream opened for OMM Connection Events.

```
// Unsubscribe for OMMConnectionIntSpec
_ommConsumer.unregisterClient(connectionInterestHandle);
```

#### Example 100: A Consumer Closing an OMMConnectionErrorIntSpec

### 12.2.13.4 Unregister Interest in OMMConnectionStatsIntSpec

The following code closes an event stream opened for OMM Connection Stats Events.

```
// Unsubscribe for OMMConnectionStatsIntSpec
_ommConsumer.unregisterClient(connectionStatsInterestHandle);
```

#### Example 101: A Consumer Closing an OMMConnectionStatsIntSpec

### 12.2.13.5 Unregistering Interest In Service Directory

The following code closes an event stream opened for Service Directory messages.

```
// Unregister Service Directory
_ommConsumer.unregisterClient(directoryHandle);
```

#### Example 102: A Consumer Closing a Service Directory

### 12.2.13.6 Closing the Login Stream

The application can log out by passing the login stream handle (i.e., the handle obtained from making a login request) to the `OMMConsumer.unregisterClient()` method. This closes the login stream, and all item event streams associated with the login stream. Since the RFA API handles closing all related items with the single logout request, this is the most efficient way of closing all event streams.

```
// Logout
_ommConsumer.unregisterClient(loginHandle);
```

#### Example 103: Client Session Logout

### 12.2.13.7 Unregistering All OMM Data Event Streams Except for the Login Stream

Instead of unregistering on a per-handle basis, an application can invoke the `unregisterClient()` method on an `OMMConsumer` to close all event streams opened for an OMM Consumer by passing in `null` as a parameter. The login stream for the Event Source will remain open and not be closed.

### 12.2.13.8 Application Cleanup

The remaining application cleanup tasks are shown in the example cleanup. Cleanup of objects is generally done in the opposite order they were created/initialized. The OMM Consumer's Event Source must be cleaned up by invoking the `destroy()` method.

An application may invoke the `destroy()` method on an Event Source without the need to explicitly unregister all its event streams first. In this case, RFA internally unregisters all open event streams.

```
// application specific cleanup
:
if (_ommConsumer != null)
    _ommConsumer.destroy();

_eventQueue.destroy ();
_session.release();
Context.uninitialize();
```

#### Example 104: Consumer Application Cleanup

## 12.3 OMM Interactive Provider Application

### 12.3.1 Interactive Provider Architecture

The following illustration lists the interfaces and classes used by an OMM interactive provider application:

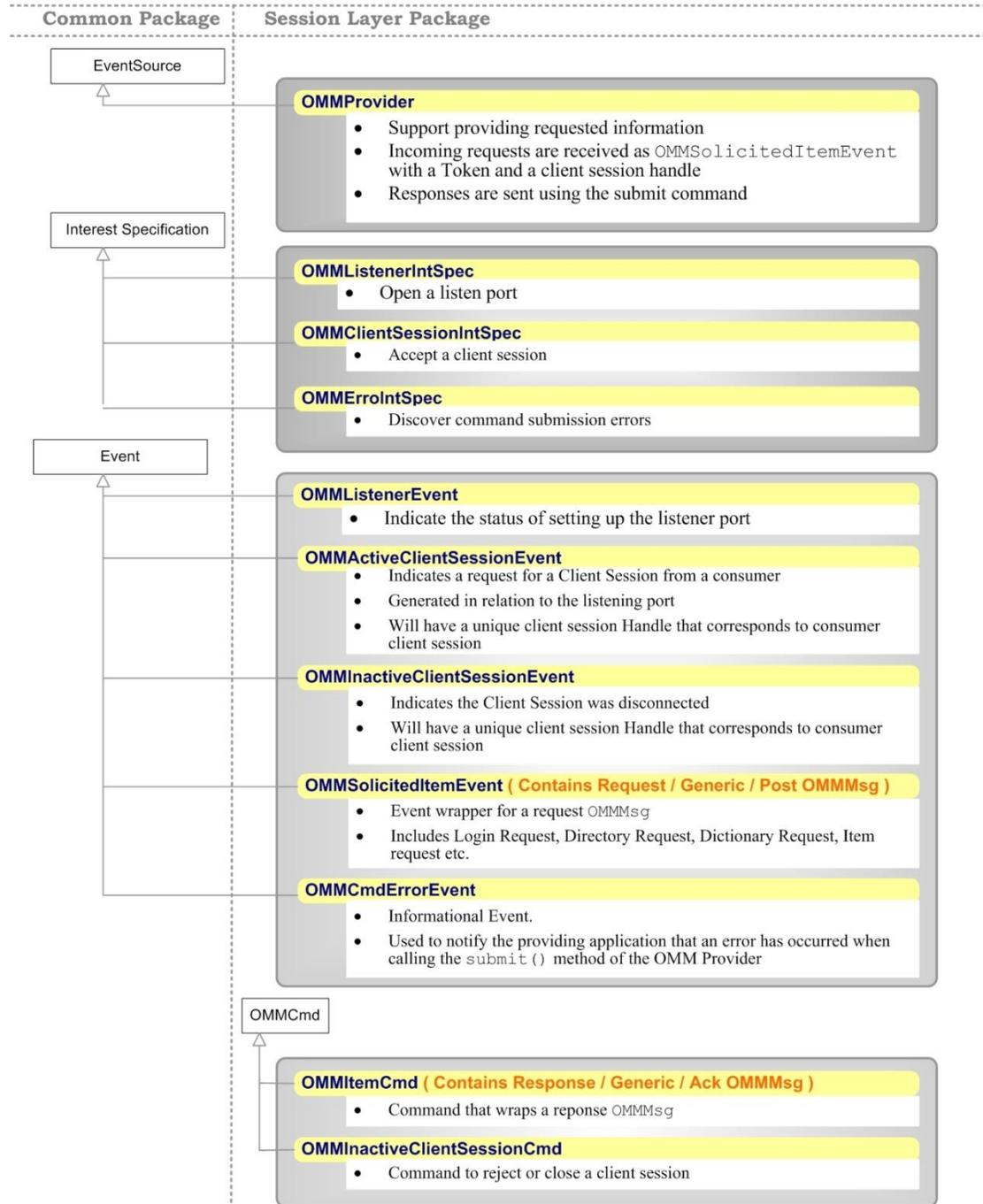


Figure 47: OMM Interactive Provider Interfaces

The OMM Provider is an Event Source that offers the ability to provide services. The services supply market information and other information. An instance of an OMMPProvider can be created using the `createEventSource()` method of the `Session` interface.

The OMM Provider supports several concrete Interest Specification, Event and Command interfaces. Events are the wrapper objects that are used for requests to the OMM Provider. Events represent inbound requests and generic messages from the consumer client session. Commands are the wrapper objects that are used for responses and generic messages from the OMM Provider. Commands are submitted via `submit()` method.

Refer to Session Package Sections 8.5, 8.6, and 8.4 for details.

## 12.3.2 Interactive Provider Task Overview

The table below lists the various tasks a provider application may execute. (Note: an application will not necessarily execute *all* the tasks shown.) In the diagram, the RFA is to the left of the provider application. The  $\rightarrow$  arrow represents the events delivered to the provider application and the  $\leftarrow$  arrow represents the Interest Specifications/Commands used by the provider application.

The gray sections denote *what* will be accomplished, and the white sections denote *how* to accomplish them:

INTERACTIVE PROVIDER APPLICATION		
	←	Startup (Refer to 0 for details.)
		<ul style="list-style-type: none"> <li>• Create a session using the <a href="#">Session</a> interface</li> <li>• Create an <a href="#">OMMProvider</a> event source</li> <li>• Create an <a href="#">OMMListenerIntSpec(connectionName)</a></li> <li>• Invoke <a href="#">registerClient()</a> on <a href="#">OMMProvider</a> (an event source); this returns the listener handle (for details on handles and tokens, refer to the topic</li> <li>• <b>Client</b> Session Handles and Request Tokens.</li> <li>• Create an <a href="#">OMMErrorIntSpec()</a></li> <li>• Call <a href="#">registerClient()</a> on <a href="#">OMMProvider</a>; this returns an Error handle</li> </ul>
	→	Listening port setup (Refer to 0 for details.)
		<ul style="list-style-type: none"> <li>• Receive an <a href="#">OMMListenerEvent</a> (with a successful state)</li> </ul>
	→	Accepting/Rejecting Client Session (Refer to 12.3.6 for details.)
		<ul style="list-style-type: none"> <li>• Receive an <a href="#">OMMActiveClientSessionEvent</a> with the ClientSessionHandle</li> <li>• Create an <a href="#">OMMClientSessionIntSpec</a> using the Client Session Handle</li> <li>• Invoke <a href="#">registerClient()</a> on the <a href="#">OMMProvider</a> event source; this returns the client session handle</li> <li>OR</li> <li>• Invoke <a href="#">submit()</a> i.e., <a href="#">submit(OMMInactiveClientSessionCmd)</a></li> </ul>
	→	Processing messages from Consumer Application (Open Request / Change Request / Close Request / Generic Message / Post Message) (Refer to 12.3.7 for details.)
		<ul style="list-style-type: none"> <li>• Receive an <a href="#">OMMSolicitedItemEvent</a> with an OMMMsg and Request Token</li> <li>• Process Requests / Generic messages / Post Messages if applicable to message model type</li> </ul>
	←	Sending messages (Response / Generic / Ack Message to Consumer Application) (Refer to 12.3.8 for details.)
		<ul style="list-style-type: none"> <li>• Create an <a href="#">OMMMsg</a> as per the <i>RFA Java RDM Usage Guide</i></li> <li>• Create an <a href="#">OMMItemCmd</a> with an OMMMsg and Request Token</li> <li>• Call <a href="#">submit()</a> i.e., <a href="#">submit(OMMItemCmd)</a></li> </ul>
	→	Error Notifications (Refer to 12.3.9 for details.)
		<ul style="list-style-type: none"> <li>• Receive <a href="#">OMMCmdErrorEvent</a></li> </ul> Handle Errors
	→	Client Session Disconnect (Refer to 12.3.10 for details.)
		<ul style="list-style-type: none"> <li>• Receive <a href="#">OMMInactiveClientSessionEvents</a> with the ClientSessionHandle</li> <li>• Clean up client resources</li> </ul>
	←	Shut down the application (Refer to 12.3.11 for details.)
		<ul style="list-style-type: none"> <li>• Clean up the application</li> </ul>

Table 83 OMM Interactive Provider Application Steps

### 12.3.3 Client Session Handles and Request Tokens

A client session handle represents a particular client session. An example of a client session would be an OMM consuming application. There will be one client session handle for each client session and each client session handle will typically have multiple request tokens associated with it. Client session handles are unique and will never be duplicated.

A request token object represents a unique consumer client session item request. The item requests can be for any message model type. When a consumer application requests a unique item, the provider application receives a unique request token. The provider application may receive this token again when the same consumer invokes `reissueClient()`, or `unregisterClient()` for the same item, or when a generic message is received. The request tokens are associated with an `OMMSolicitedItemEvent` event and are represented by the `Token` interface.

The provider application must maintain the request tokens of streaming requests as they represent open items. The request tokens must be associated with the client session handle obtained from the event. The lifespan of these request tokens lasts until either the provider application closes the stream, or the consumer client session issues a close for the request token. The request tokens will be used when sending responses or generic messages on an open stream.

Not all request tokens need to be maintained by the provider application. Request tokens of non-streaming requests can be discarded after a message relating “refresh complete” has been submitted to the OMM Provider. Note that depending on the size of the refresh, the application may choose to send the refresh in multiple `submit()` calls.

If two consumers request the same item, the provider will receive two unique tokens, and is required to fan-out responses to each token.

The client session handle can be obtained from `OMMInactiveClientEvent` and `OMMSolicitedItemEvent` using the `getHandle()` method.

### 12.3.4 Startup

First an interactive provider application creates a session. After creating the session, the application creates an Event Queue to accept events encapsulating the requests made by consumer. Next, the OMM Provider Event Source is created by invoking the `createEventSource()` method on the session. The Event Source type `EventSource.OMM_PROVIDER` and a name for the provider are passed to `createEventSource()` as input parameters:

```
// create the interactive server provider session
Session providerSession = Session.acquire("providerSession");

// create the event queue to accept requests
EventQueue _eventQueue = EventQueue.create("Provider Application EventQueue");

// create OMMProvider event source
OMMProvider _ommProvider = (OMMProvider) providerSession.createEventSource(EventSource.OMM_PROVIDER,
    "myOMMProvider");

// OMMErrorIntSpec is used to register interest for any Error events during the publishing cycle.
OMMErrorIntSpec errIntSpec = new OMMErrorIntSpec();
_errIntSpecHandle = _ommProvider.registerClient(_eventQueue, errIntSpec, myClient, null);
```

#### Example 105: OMM Interactive Provider Application Startup

The `createEventSource()` method also accepts an optional parameter which specifies whether or not Completion Events should be sent. By default, Completion Events are not generated.

The application must create a client call-out that implements the `Client` interface and its `processEvent()` method (to process inbound events). Some events encapsulate an `OMMMsg` containing requests originating from a consumer.

## 12.3.5 Listening Port Setup

The application creates an Interest Specification `OMMListenerIntSpec` Interest Specification to listen (on a port) for incoming client sessions from consumer applications. The listener name is set on the Interest Specification using `setListenerName()`. The Interest Specification is registered using `registerClient()`, as shown in the following example:

```
// OMMListenerIntSpec is used to register interest for any incoming sessions on the server port.
OMMListenerIntSpec listenerIntSpec = new OMMListenerIntSpec();

String connection = CommandLine.variable("listenerName"); // name of the prov connection
listenerIntSpec.setListenerName(connection);
_csListenerIntSpecHandle = _ommProvider.registerClient(_eventQueue, listenerIntSpec, myClient, null);
```

### Example 106: Opening a Listening Port

If the port is established successfully, the application receives a single `OMMListenerEvent` event with a state indicating success. The application is now ready to receive `OMMActiveClientSessionEvent` events, which represent client session connect requests. The code below shows the use of `getStatus()` method to obtain the status of the listener port setup:

```
public void processEvent(Event event)    {
    switch (event.getType())    {
        case Event.OMM_LISTENER_EVENT:
            OMMListenerEvent listenerEvent = (OMMListenerEvent)event;
            Status status = listenerEvent.getStatus();
            ...
            break;
    }
}
```

### Example 107: Handling OMMListenerEvents

If the port is in use by another application on the same machine, it is indicated by a failed `OMMListenerEvent` and a Status Code of `OMMListenerEvent.WILL_RETRY`. The application will receive multiple instances of these events until either the port is successfully set up, or the application is shut down. If the listening port has already been established by the application, then the application receives a single failed `OMMListenerEvent` and a Status Code of `NONE`.

If the application receives a successful event, it will start receiving client session connect requests. No response message is required from the provider application for this Event.

## 12.3.6 Accepting/Rejecting Client Sessions

A client session request is obtained as an `OMMActiveClientSessionEvent` event. The provider application should handle this event in the `processEvent()` method:

```
public void processEvent(Event event)    {
    switch (event.getType())    {
        case Event. OMM_ACTIVE_CLIENT_SESSION_PUB_EVENT:
            processActiveClientSessionEvent((OMMActiveClientSessionEvent)event);
            break;
    }
}
```

The provider application may accept or reject the request. If the provider does not respond within a specific timeout period, the client session channel will be dropped by the consuming side since it has not received a response within the timeout.

### 12.3.6.1 Accepting Client Session Requests

A provider application accepts a client session request by creating an `OMMClientSessionIntSpec`, populating it with the client session handle, and calling `registerClient()`. The client session handle is obtained from the Event using the `getClientSessionHandle()` method, and set on the Interest Specification using the `setClientSessionHandle()` method.

The `registerClient()` method will return a client session handle. The client session handle will need to be maintained by the provider application, since this will be the unique identifier representing that consumer client session.

This completes accepting a client session request. The application can now receive `OMMSolicitedItemEvents` from the consuming client session. The process of accepting a client session is illustrated by the following example:

```
protected void processActiveClientSessionEvent(OMMActiveClientSessionEvent event)
{
    // Accepting a session through the registerClient interface:
    // This will return a client session handle
    Handle handle = event.getClientSessionHandle();
    OMMClientSessionIntSpec intSpec = new OMMClientSessionIntSpec();
    intSpec.setClientSessionHandle(handle);
    _clientSessionHandle = _ommProvider.registerClient(_eventQueue, intSpec, this, null);
}
```

#### Example 108: Accepting a Consumer Client Session

### 12.3.6.2 Rejecting Client Session Requests

To reject a client session, the provider application needs to create an `OMMInactiveClientSessionCmd`, specify the client session handle, set the appropriate status and call `submit()`.

Once a client session has been rejected, the provider application needs to remove all references to that client session handle and all of its associated request tokens. The client session can be rejected at the time of request, or any time after it is registered with the provider.

```
private void rejectClientSessionEvent(OMMActiveClientSessionEvent event)
{
    System.out.println("Pub session denied.");
    OMMInactiveClientSessionCmd cmd = new OMMInactiveClientSessionCmd();
    cmd.setClientSessionHandle(event.getClientSessionHandle());
    _ommProvider.submit(cmd, null);
}
```

#### Example 109: Rejecting a Consumer Client Session

## 12.3.7 Processing Messages from Clients

Once the provider application accepts a client session, the application handles messages from the consumer, including OMM Data requests, change requests, and close requests.

The provider application receives messages as `OMMMsg` objects encapsulated in an `OMMSolicitedItemEvent` event. The message could be of any message model type. The client session handle for the request is available via the `getHandle()` method on the Event.

- The provider application retrieves the request message from the Event and processes it based on its type.
- The procedure to process request messages is as follows:
  - Determine the concrete event type using the event's `getType()` method. The event type for request messages is `OMM_SOLICITED_ITEM_EVENT`.
  - Cast the Event to the appropriate event, i.e., `OMMSolicitedItemEvent`. This Event is a wrapper around the message with request details.
  - Obtain the request message using the `getMsg()` method of `OMMSolicitedItemEvent`.
  - Obtain the message model type of the request using `getMsgModelType()`.
  - Obtain the message type of the request using `getMsgType()`.

- Process the request message based on the message model and message type. Refer to Section 6.15 for details regarding decoding an OMM Message. The processing depends on the message that was received.

The code example below demonstrates a provider application receiving a message by a provider application:

```
public void processEvent(Event event)    {
    switch (event.getType())    {
        case Event.OMM_SOLICITED_ITEM_EVENT:
            OMMSolicitedItemEvent itemEvent = (OMMSolicitedItemEvent)event;
            processItemEvent(itemEvent);    // application implemented
            break;
        ...
    }
}
private void processItemEvent(OMMSolicitedItemEvent event)
{
    OMMMsg msg = event.getMsg();
    switch (msg.getMsgModelType())    {
        case RDMMsgTypes.LOGIN:
            processLoginRequest(event);    // application implemented
            break;
        case RDMMsgTypes.DIRECTORY:
            processDirectoryRequest(event);    // application implemented
            break;
        case RDMMsgTypes.MARKET_PRICE:
            processMarketDataReq(event);    // application implemented
            break;
        ...
    }
}
```

### Example 110: Processing a Request from a Consumer

As with other Events, the OMM request Event (and its contained request message if applicable) is only valid during the client callback. If the application wishes to retain the data beyond the scope of the callback, it must make a copy of the contained message. Other event types that an OMM Provider application may receive are [Complevent](#) and [LoggerNotifyEvent](#).

A provider application may receive the following requests:

- Request for data

A request for a data can be streaming or nonstreaming. When the application receives a nonstreaming request, it sends a response with the requested data and closes the stream. When the request is streaming, the provider application allocates the stream for the request and manages the state of the stream until the stream is closed, or the client session is closed.

- Change request

A consumer application may send subsequent requests on an open stream and request changes in the attributes. Provider applications should change handling of the request accordingly.

- Close Request

A consumer application may close an open stream by sending a close request message.

- Generic message

A provider application may receive generic-type message from a consumer on an open stream.

- Posting

A provider application may receive post-type messages from consumer on an open stream.

A provider application should maintain the state and characteristics of each open stream. The state and characteristics are domain dependent. For example, a stream opened for Directory Request can't be paused, whereas an item stream can.

A stream has an associated priority, which may be paused or active. If the stream is active, the application should provide update response messages. The update messages are sent either periodically or if the provided data changes.

A provider application processes a request based on the message model type. A provider application responds to a request by creating and sending a refresh response message or status message.

Retrieving and processing of attribute information and payload information is described in Section 6.15.3.

### 12.3.7.1 Handling Login Requests

The first request received by a provider after accepting a client session is the client's request for login, followed by a request for a Source Directory. Subsequent requests may be of any type.

The provider application may grant or deny the login. In either case, a response is sent to the consumer application.

If the provider application denies a login, it must clean up all request tokens for that particular client session. In addition, it must ignore further incoming requests for the same client session after the login denial message has been submitted. Note that a provider application can reject a login at any time after it has accepted a particular login.

When a provider application accepts the consumer login, it responds with a Login response. The response should include flags notifying the recipient about capabilities of the provider. The response message should include Attribute Information element with the following Attributes elements set accordingly:

- `RDMUser.Attrib.SupportOptimizedPauseResume`
- `RDMUser.Attrib.SupportOMMPost`
- `RDMUser.Attrib.SupportStandby`
- `RDMUser.Attrib.SupportViewRequests`
- `RDMUser.Attrib.SupportBatchRequests`

The following code illustrates a provider application composing a Login response containing Attribute element of Attribute Information set to advertise that the provider supports Post, Optimized Pause and Resume, and Warm Standby, but not batch or views (declared by omission). For details on the Login domain, refer to the *RFA Java RDM Usage Guide*.

```
OMMMsg outmsg = _pool.acquireMsg();
outmsg.setMsgType(OMMMsg.MsgType.REFRESH_RESP);
outmsg.setMsgModelType(RDMMsgTypes.LOGIN);
outmsg.setIndicationFlags(OMMMsg.Indication.REFRESH_COMPLETE);
outmsg.setState(OMMState.Stream.OPEN, OMMState.Data.OK, OMMState.Code.NONE, "login accepted");
outmsg.setRespTypeEnum(OMMMsg.RespType.SOLICITED);
_encoder.initialize(OMMTypes.MSG, 1000);
_encoder.encodeMsgInit(outmsg, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA);
// add the "SupportOMMPost" element
_encoder.encodeElementEntryInit(RDMUser.Attrib.SupportOMMPost, OMMTypes.UINT);
_encoder.encodeUInt(1);
// add the "SupportOptimizedPauseResume" element
_encoder.encodeElementEntryInit(RDMUser.Attrib.SupportOptimizedPauseResume, OMMTypes.UINT);
_encoder.encodeUInt(1);
// add the "SupportStandby" element
_encoder.encodeElementEntryInit(RDMUser.Attrib.SupportStandby, OMMTypes.UINT);
_encoder.encodeUInt(1);

_encoder.encodeAggregateComplete();
respMsg = (OMMMsg) _encoder.getEncodedObject();
```

#### Example 111: Notifying the Consumer Application of the Features Supported By a Provider

### 12.3.7.2 Handling Item Open Requests and Modification Requests

When it receives an open item request, the provider application needs to establish an event stream for the requested item. Event streams are maintained by the application in an application-implemented data store.

When it receives a reissue request, the provider application needs to find the associated event stream in the data store and change the attributes.

The example below shows the provider handling the open item requests and item reissues. The application-implemented `itemReqTable()` method manages open item event streams:

```

OMMMsg requestMsg = event.getMsg();
Token requestToken = event.getRequestToken(); // A token is associated with each unique request.
Item item = itemReqTable(requestToken); // this application-implemented method returns null if the
// item has not been requested yet

int messageType = requestMsg.getMsgType();
if (messageType == OMMMsg.MsgType.REQUEST)
{
    if (item == null) // the item has not been requested yet - open request
    {
        add(requestToken); // remember that we processed this item request
        processItemReq(); // application implemented
    }
    else // the item is in the data base
    {
        processItemModification(item); // application implemented
    }
}

```

### Example 112: Processing Item Open Request and Item Modification Request

#### 12.3.7.3 Handling an Item Close Request

A provider application shall close an item previously opened when the application receives a close request. The request token for the item opened previously is retrieved from the Event using [getRequestToken\(\)](#).

```

OMMMsg requestMsg = event.getMsg();
Token requestToken = event.getRequestToken(); // A token is associated with each unique request.

int messageType = requestMsg.getMsgType();
if (messageType == OMMMsg.MsgType.CLOSE_REQ)
{
    remove(requestToken); // removes the reference to the Token
    ...
}

```

### Example 113: Processing an Item Close Request

## 12.3.8 Sending Messages

The procedure to create and send messages in a provider application is described below:

- Create the response [OMMMsg](#)
- Create the [OMMItemCmd](#) command object
- Encapsulate the command object with the response message using the [setMsg\(\)](#) method. The [OMMItemCmd](#) essentially acts as a wrapper around the response message.
- The token is set using the [setToken\(\)](#) method.
- Invoke the [submit\(\)](#) method on the OMM Provider Event Source, passing in the closure object containing the application specified data.

---

**NOTE:** The Command, response message, and attribute objects may be reused for making subsequent responses.

---

The process of sending message is the same for each type of message: refresh response, update response, status, generic, and acknowledge. For each of these cases, the provider application must retrieve an event associated with the event stream and retrieve a request token from this event. There might be specific constraints depending on the type of the message and the message model.

- For a description of OMM message types, refer to Chapter 1.
- For information on message domains refer to the *Java Reference Manual*.

```

...
OMMMsg msg = encodeOutMsg(...);           // application method
OMMItemCmd cmd = new OMMItemCmd();        // Create a new OMMItemCmd
cmd.setMsg(msg);
Token rq = event.getRequestToken();      // Token is associated with request (i.e., event stream)
cmd.setToken(rq);
_ommProvider.submit(cmd, null);
...

```

## Example 114: Creating and Sending Responses

### 12.3.8.1 Refresh Responses

The refresh response message is sent by a provider application in the following cases:

- The application received a request to open a new event stream from a consumer, or a snapshot request
- The application received a request for a refresh on an open stream
- The application received a request to modify an open stream with an option to send refresh

The Refinitiv Source Sink Library protocol that the RFA uses send messages has a limitation on the maximum size of a messages.. For some domains, the refresh message may exceed the limit. In this case, the refresh response message may be sent to the user in several parts. The last part should have `OMMMsg.Indication.REFRESH_COMPLETE` flag set.

### 12.3.8.2 Update Responses

Update responses are sent by a provider application on open event streams. Depending on the stream parameters (i.e., Quality of Service and/or attributes information received on request messages) the provider should implement an appropriate timing mechanism to provide the updates.

In order to send a message on an open stream, the provider needs a token, associated with the open stream. The provider application typically stores the open streams tokens that use the same timing mechanism in some a data structure. When there is a time to send the updates, (for example a periodic timer expires), the application iterates through the collection of tokens and generates an update for each token.

### 12.3.8.3 Status Responses

A status response message is sent by a provider application in the following cases:

- As a response to the Login Request
- This response is defined by Refinitiv Source Sink Library protocol. If the login is granted to the consumer, the application sends a status response with the state set to relate an "OK" status.
- The application received a request, but the provider can't provide the requested service
- If the provider can't fulfill a request, the status response sent by the provider will have a state that indicates why the request could not be satisfied. If the state is final, i.e., the item can't be open, the provider application releases all resources associated with this request.
- The state of service changed, and this change affects open event streams
- If the provider's state changed and it can't provide the service, the status response sent by the provider will have a state indicating the problem. If the state is final, i.e., the stream can't be open, the provider application releases all resources associated with this event stream.

### 12.3.8.4 Generic Messages

A generic message is sent by a provider application on an open event stream. A provider may generate a custom data in the message payload. If the data size exceeds a limit that the Refinitiv Source Sink Library transport can handle, the message can be partitioned into several generic messages. The OMM package defines an `OMMMsg.Indication.GENERIC_COMPLETE` indication flag and a secondary sequence number, as an element of `OMMMsg`, to support generic message partitioning.

### 12.3.8.5 Acknowledge Messages

For details on Acknowledge messages, refer to Section 13.6.

## 12.3.9 Error Notifications

A provider application receives error notifications related to invocations of `submit()` via `OMMCmdErrorEvent` events. This event gives access to the command, command ID, error closure, submit closure, and status for the command that failed. These can be accessed via the `getCmd()`, `getCmdID()`, `getClosure()`, `getSubmitClosure()`, and `getStatus()` methods respectively.

```
public void processEvent(Event event)    {
    switch (event.getType())    {
        case Event.OMM_CMD_ERROR_EVENT:
            OMMCmdErrorEvent errorEvent = (OMMCmdErrorEvent)event;
            processErrorEvent(errorEvent);    // application implemented
            break;
        ...
    }
}
protected void processErrorEvent(OMMCmdErrorEvent event)
{
    OMMCmd cmd = event.getCmd();
    int cmdId = event.getCmdId();
    OMMErrorStatus status = event.getStatus();
}
```

### Example 115: Handling Error Notifications

## 12.3.10 Client Session Disconnections

When a client session gets disconnected, the provider application receives the `OMMInactiveClientSessionEvent`. The client session handle is available in the Event.

On receiving this Event, the provider application must do the following:

- Invoke `unregisterClient()` with the corresponding client session handle. This should be called for proper cleanup with the OMM Provider.
- Stop sending data to the client session.
- Remove the client session handle and its associated request tokens.

## 12.3.11 Interactive Provider Application Shut Down

The graceful shut down of application should include the clean up steps described below. The application should notify the clients of closing items first, and also notify them the session is closed.

### 12.3.11.1 Closing Items

In the case where the providing application needs to close an item, the application needs to send an `OMMItemCmd` with a closed status for a specific request token. The provider application then needs to make sure that no further data is being sent using the request token, and any reference to it should be removed.

### 12.3.11.2 Handling a Login Close

In the case where the connected consumer issues a login close, the providing application will receive an `OMMSolicitedItemEvent` for login with an `OMMMsg` and a request token. The provider application will then receive an `OMMInactiveClientSessionEvent` to signal that the consumer has disconnected.

Once the `OMMSolicitedItemEvent` for the login has been received, the provider application then needs to make sure that no further data is sent specifying any of the request tokens associated with that login. All the associated request tokens are required to be discarded. The provider application would need to discard the associated request tokens on the first event, either login close event or inactive client session event.

For details on handling `OMMInactiveClientSessionEvent`, refer to the topic **OMM Inactive Client Session Event**.

### 12.3.11.3 Handling Item Close

In cases where the connected consumer closes a particular item, the providing application will then receive an `OMMSolicitedItemEvent` for an item with an `OMMMsg` and a request token. The provider application then needs to make sure that no further data is sent specifying that request token. That particular request token is required to be discarded.

### 12.3.11.4 Closing the Client Session

At any time after the client session has been accepted, the provider application has the ability to close the session. The provider application must clean up all the token references for the associated client session handle. A client session can be closed in the following ways:

- Using the Login domain's [OMMMessage](#)

This involves creating a login message, setting the message type to STATUS and state to CLOSED. The reason for the logout can be specified in the status text. This message is sent to the consumer client session:

```
OMMMsg responseMessage = pool.acquireMsg();
responseMessage.setMsgModelType(RDMMsgTypes.LOGIN);
responseMessage.setMsgType(OMMMsg.MsgType.STATUS_RESP);
responseMessage.setState(OMMState.Stream.CLOSED, OMMState.Data.SUSPECT, OMMState.Code.NOT_FOUND,
    "Logging out Client Session");

OMMItemCmd cmd = new OMMItemCmd();
cmd.setMsg(responseMessage);
cmd.setToken(token);

_ommProvider.submit(cmd, null);
```

#### Example 116: Closing Down Client Session Using a (Login) OMMMsg

- Using [OMMInactiveClientSessionCmd](#)

The provider application can choose to disconnect a client session by submitting an [OMMInactiveClientSessionCmd](#) and specifying the client session handle to be disconnected. It is not possible to specify the reason for the logout using this method:

```
// Disconnect the client session
OMMInactiveClientSessionCmd cmd = new OMMInactiveClientSessionCmd();
cmd.setClientSessionHandle(_clientSessionHandle);

_ommProvider.submit(cmd, null);
System.out.println("Client Session " + _clientSessionHandle + " has been disconnected");
```

#### Example 117: Closing Down a Client Session Using [OMMInactiveClientSessionCmd](#)

### 12.3.11.5 Unregistering Interest in Client Session Connection Request

When a provider application no longer has interest in receiving any more consumer client session requests, it may relinquish interest via the [unregisterClient\(\)](#) method, specifying the handle returned from the [registerClient\(\)](#) with the [OMMListenerIntSpec](#):

```
// Unregister inbound Consumer Client Session Requests)
_ommProvider.unregisterClient( _listenerHandle );
```

#### Example 118: Unregistering Interest in Client Session Connection Requests

### 12.3.11.6 Unregistering Interest in Command Error Events

A provider may unregister interest in Command Error events by passing an error handle to `unregisterClient()`:

```
// Unregister Error notifications
_ommProvider.unregisterClient( _ErrorHandle );
```

#### Example 119: Unregistering Interest in Command Error Notifications

### 12.3.11.7 Other Cleanup Tasks

The example below shows how the application should finish shutting down after the above tasks are completed. Section 12.3.11.1 discusses closing open item streams in detail. Applications may choose not to implement this step (closing open item streams) during shutdown. In this case, RFA will internally close the open event stream when the `destroy()` method on an event source is invoked by the application.

```
// Destroy the EventQueue
_eventQueue.deactivate();
_eventQueue.destroy();

// Destroy Event Source
_ommProvider.unregisterClient(_listenerIntSpecHandle);
_ommProvider.unregisterClient(_errIntSpecHandle);
_ommProvider.destroy();

// Release Session
session.release();

//Uninitialize Context
Context.uninitialize();
```

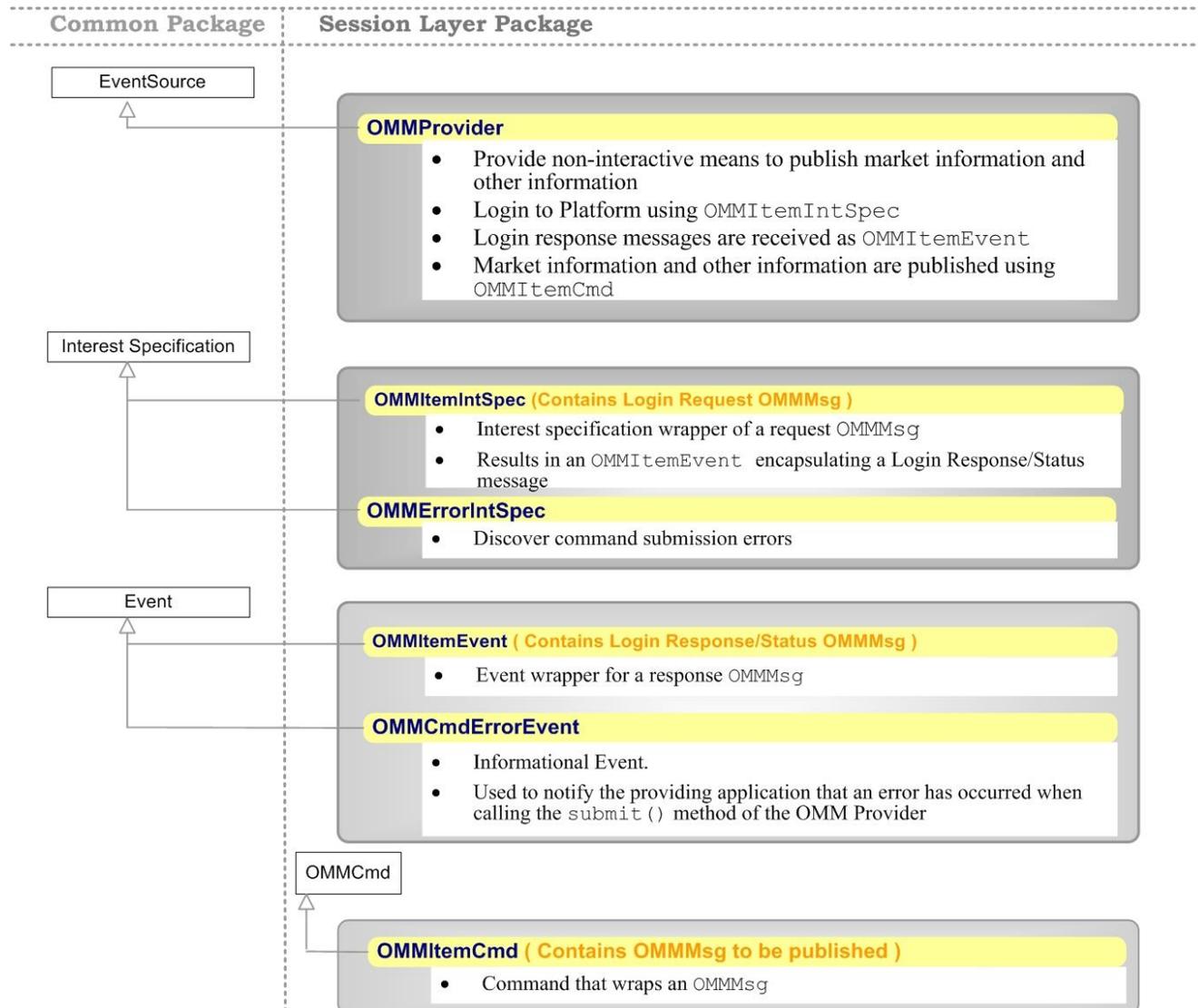
#### Example 120: OMM Interactive Server Provider Application Cleanup

## 12.4 OMM Non-Interactive Provider

### 12.4.1 Non-interactive Provider Architecture

The purpose of the OMM Non-interactive Client Provider is to afford applications the capability of publishing OMM data without the need for a consumer to specifically request every item being published. The OMM Non-interactive Provider is useful for a provider application that does not have a cache, and is publishing OMM Data to a non-interactive data source (e.g. an ADH).

The following illustration lists the interfaces and classes used by an OMM Data non-interactive provider application:



**Figure 48: OMM Non-Interactive Provider Interfaces**

The non-interactive provider uses the OMM Provider event source. The OMM Provider is an Event Source that offers the ability to provide services. The services supply market information and other information. An instance of an OMMProvider can be created using the `createEventSource()` method from the `Session` interface.

## 12.4.2 Non-interactive Provider Task Overview

The following table lists the tasks applying to a non-interactive application.

- The RFA Provider is assumed to be to the left of the provider application.
- An arrow (e.g., →) represents an event and points in the direction of the event.

NON-INTERACTIVE PROVIDER APPLICATION	
ROBUST FOUNDATION API	← Startup (Refer to 0 for details.)
	<ul style="list-style-type: none"> <li>• Create a session using the <a href="#">Session</a> interface</li> <li>• Create an <a href="#">OMMProvider</a> event source</li> <li>• Create an <a href="#">EventQueue</a> to receive the events</li> </ul>
	← Login to Platform (Refer to 12.4.4 for details.)
	<ul style="list-style-type: none"> <li>• Create an <a href="#">OMMMsg</a> (Login) per the <i>RDM Usage Guide</i></li> <li>• Create an <a href="#">OMMItemIntSpec</a> interest specification</li> <li>• Encapsulate an <a href="#">OMMItemIntSpec</a> with an <a href="#">OMMMsg</a></li> <li>• Invoke <a href="#">registerClient()</a> on <a href="#">OMMProvider</a> event source; this returns the login handle</li> <li>• Create an <a href="#">OMMErrorIntSpec()</a></li> <li>• Invoke <a href="#">registerClient()</a> on <a href="#">OMMProvider</a> event source; this returns the error handle</li> <li>• Create <a href="#">OMMConnectionIntSpec()</a></li> <li>• Invoke <a href="#">registerClient()</a> on <a href="#">OMMProvider</a> event source; this returns the connection interest handle</li> </ul>
	→ Process Event (Login Response or generic message) (Refer to 0 for details.)
	<ul style="list-style-type: none"> <li>• Receive an <a href="#">OMMItemEvent</a></li> <li>• Process the login response or generic message</li> </ul> Or <ul style="list-style-type: none"> <li>• Receive an <a href="#">OMMCmdErrorEvent</a></li> <li>• Handle Errors</li> </ul> Or <ul style="list-style-type: none"> <li>• Receive an <a href="#">OMMConnectionEvent</a></li> <li>• Handle OMM Connection Events</li> </ul>
	← Sending message (Unsolicited Refresh Response / Update Response/Generic) (Refer to 12.4.6 for details.)
	<ul style="list-style-type: none"> <li>• Create an <a href="#">OMMMsg</a> as per the <i>RDM Usage Guide</i></li> <li>• Generate Token</li> <li>• Create an <a href="#">OMMItemCmd</a> with an <a href="#">OMMMsg</a> and Token</li> <li>• Invoke <a href="#">submit()</a> i.e., <a href="#">submit(OMMItemCmd)</a></li> </ul>
← Closing application (Refer to 12.4.7 for details.)	
<ul style="list-style-type: none"> <li>• Invoke <a href="#">unregisterClient()</a> with a <a href="#">ClientSessionHandle</a> on an <a href="#">OMMProvider</a> event source</li> <li>• Clean up client resources</li> <li>• Cleanup the application</li> </ul>	

**Table 84 OMM Non-Interactive Provider Application Steps**

### 12.4.3 Startup

The application creates the session and an event queue to accept events. The OMM Provider Event Source is created next using the session and calling the `createEventSource()` method. The example below shows the initial steps:

```
// create the provider session
Session session = Session.acquire("providerSessionNI");

// create the event queue to accept requests
EventQueue eventQueue = EventQueue.create("Non-interactive Client Provider Application EventQueue");

// create OMMProvider event source.
OMMProvider ommProvider = (OMMProvider) session.createEventSource(EventSource.OMM_PROVIDER,
    "NoninteractiveProvider");
```

#### Example 121: OMM Non-interactive Client Provider Application Setup

### 12.4.4 Login

The application must first login successfully prior to publishing the Service Directory and market data. The Login Request is encapsulated in `OMMItemIntSpec`. For an example of encoding a Login Request used in a consumer application, refer to Section 12.2.3.1. The encoding is the same for the Non-Interactive Provider application, though some parameters may be different. The login parameter role `RDMUser.Attrib.Role` indicates if this is a provider (`RDMUser.Role.PROVIDER`) or a consumer (`RDMUser.Role.CONSUMER`). For a Non-Interactive Provider, this element is set to `RDMUser.Role.PROVIDER`.

The application sets up error notifications by registering the interest specification `OMMErrorIntSpec`. This allows the application to receive error events related to `submit()` calls.

The application sets up OMM Connection Event notifications by registering the interest specification `OMMConnectionIntSpec`.

```
public void sendLoginRequest()
{
    OMMMsg ommmsg = encodeLoginReqMsg(); // application implemented
    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
    ommItemIntSpec.setMsg(ommmsg);
    Handle loginHandle = _ommConsumer.registerClient(eventQueue, ommItemIntSpec, this, null);
    // OMMErrorIntSpec is used to register interest for the error occurring before the message is written to the
    // transport. myClient is an application-implemented client handling errors
    OMMErrorIntSpec errIntSpec = new OMMErrorIntSpec();
    errorHandle = ommProvider.registerClient(eventQueue, errIntSpec, myClient, null);

    // OMMConnectionIntSpec is used to register interest for any OMM Connection events during the publishing
    // cycle.
    // myClient is an application-implemented client handling OMM Connection events
    OMMConnectionIntSpec connectionIntSpec = new OMMConnectionIntSpec();
    connectionInterestHandle = ommProvider.registerClient(eventQueue, connectionIntSpec, myClient, null);
}
```

#### Example 122: OMM Non-interactive Provider Application Login

## 12.4.5 Process Event

The application must create a client call-out that implements the `Client` interface and its `processEvent()` method to process inbound events. Some events (i.e., the login responses) encapsulate an `OMMMsg`.

The concrete event type is determined through the event's `getType()` method. Based on the event type, the provider application must cast the Event to the appropriate event type:

```
public void processEvent(Event event)    {
    switch (event.getType())    {
        case Event.OMM_CMD_ERROR_EVENT :
            processOMMcmdErrorEvent((OMMcmdErrorEvent) event);    // application implemented
            break;
        case Event.OMM_ITEM_EVENT:
            processMessage((OMMItemEvent) event);    // application implemented,
            break;
        case Event.OMM_CONNECTION_EVENT :
            processOMMConnectionEvent((OMMConnectionEvent) event);    // application implemented
            break;
        default :
            // unhandled event type
            break;
    }
}
```

### Example 123: Processing Inbound Events

In this example, the application implements `processEvent()` by switching on the event type, casting the Event to the appropriate event, and calling the corresponding application implemented method, e.g. `processMessage()`. The incoming message can be a login response, or generic-type message.

#### 12.4.5.1 Handling Error Notifications

Non-interactive provider applications handle error notifications using the same pattern implemented by interactive provider applications. For details, refer to Section 12.3.9.

#### 12.4.5.2 Handling OMM Connection Events

Non-interactive provider applications handle OMM Connection Events using the same pattern implemented by consumer applications. For details, refer to section 12.2.10.

## 12.4.6 Sending Messages

A non-interactive provider publishes service directory and item information through response messages. Unlike the interactive provider, which receives requests that establish event streams, the non-interactive provider has to establish an event stream for each item that it publishes. The application sends a refresh response, followed by updates for each item. The sequence of establishing an event stream is shown below:

- Create a refresh response `OMMMsg` message.
- The initial message is an unsolicited refresh message, which opens a stream. This must include a token generated using the `OMMProvider.generateToken()` method.
- Create Command object `OMMItemCmd`.
- Encapsulate a Command object with the response message using the `setMsg()` method. The `OMMItemCmd` acts as a wrapper around the response message. The token can be set using the `setToken()` method.
- Invoke the `submit()` method of the OMM Provider Event Source, passing in the command, and optionally a closure object containing the application specified data.

#### 12.4.6.1 Sending Service Directories

A non-interactive provider application typically publishes a Service Directory to the ADH after a successful login. The Service Directory is based on the Directory model as defined in the *RFA Java Edition RDM Usage Guide*. The Service Directory contains information of multiple services that the provider offers.

### 12.4.6.2 Publishing Item Refresh and Update Messages

Following the Service Directory, the Non-interactive Client Provider can publish item information. Each item publishing starts with establishing an event stream and sending an unsolicited refresh response message. The application follows with sending unsolicited update response messages, if an item's information changed.

**NOTE:** Some components, depending on their specific functionality and configuration, require that non-interactive provider applications publish attributes ([AttribInfo](#)) in their update and status messages. To verify requirements, refer to the component into which you publish.

The example below shows a non-interactive provider application establishing an event stream and sending the initial message.

```
Token token = _ommProvider.generateToken();           // Token is generated.
OMMItemCmd cmd = new OMMItemCmd();                 // Create a new OMMItemCmd.
OMMMsg responseMessage = encodeRefresh("TRI.N");    // The instrument name is "TRI.N", application implemented
cmd.setMsg(responseMessage);                       // set the cmd with response message
cmd.setToken(token);                               // set cmd with token
_ommProvider.submit(cmd, null);                    // submit the cmd

private encodeRefresh(String itemName)
{
    String serviceName = "DIRECT_FEED";           // assume service name
    OMMEncoder ommEncoder = pool.acquireEncoder();
    ommEncoder.initialize(OMMTypes.MSG, 1000);
    responseMessage.setMsgType(OMMMsg.MsgType.REFRESH_RESP);
    responseMessage.setMsgModelType(RDMMsgTypes.MARKET_PRICE);
    responseMessage.setIndicationFlags(OMMMsg.Indication.REFRESH_COMPLETE);
    responseMessage.setRespTypeNum(OMMMsg.RespType.UNSOLICITED);
    responseMessage.setAttribInfo(serviceName, itemName, RDMInstrument.NameType.RIC);
    ommEncoder.encodeMsgInit(responseMessage, OMMTypes.NO_DATA, OMMTypes.FIELD_LIST);
    ommEncoder.encodeFieldListInit(OMMFieldList.HAS_STANDARD_DATA | OMMFieldList.HAS_INFO, (short)0, (short)1,
        (short)0);
    // RDNDISPLAY
    ommEncoder.encodeFieldEntryInit((short)2, OMMTypes.UINT);
    ommEncoder.encodeUInt(100);
    // RDN_EXCHID
    ommEncoder.encodeFieldEntryInit((short)4, OMMTypes.ENUM);
    ommEncoder.encodeEnum(155);
    ommEncoder.encodeAggregateComplete();

    OMMMsg encodedMsg = (OMMMsg)ommEncoder.getEncodedObject();
}
```

#### Example 124: OMM Non-Interactive Provider Application Publishing

## 12.4.7 Closing a Non-Interactive Provider Application

### 12.4.7.1 Unregister the Login Client

The Non-interactive Provider application must logout by invoking [unregisterClient\(\)](#) and passing in the login handle.

### 12.4.7.2 Cleanup Resources

The application should clean up the resources that it allocated for the event streams.

### 12.4.7.3 Application Cleanup

The application should gracefully close all objects that it instantiated during initialization. Cleanup of objects is generally done in the opposite order they were created/initialized. Interest in command error events is unregistered by calling `unregisterClient()` and passing in the error handle. The application may call the `destroy()` method on an Event Source without having closed all event streams. In this case, RFA internally unregisters all open event streams.

The code example below shows closing a non-interactive provider application.

```
// Unregister Error notifications
_ommProvider.unregisterClient(errorHandle);

// Unregister OMM Connection Event notifications
_ommProvider.unregisterClient(connectionInterestHandle);

// Unregister login
_ommProvider.unregisterClient(loginHandle);

// Destroy Event Source
_ommProvider.destroy();

// Destroy the EventQueue
_eventQueue.deactivate();
_eventQueue.destroy();

// Release Session
session.release();

//Uninitialize context
Context.uninitialize();
```

#### Example 125: OMM Non- Interactive Client Provider Application Cleanup

## 12.5 Hybrid

### 12.5.1 Hybrid Architecture

RFA supports hybrid applications that receive request messages from a consumer and forward the same request message to another provider. In turn, a hybrid application also receives response messages from a provider and forwards the same response message to the originating consumer. The hybrid application may edit contents of the messages before forwarding them.

The diagram below shows a hybrid application in a system:

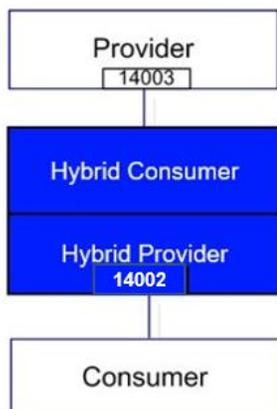


Figure 49: A Hybrid Application

The hybrid application consists of a consumer part and a provider part. In the diagram above, the consumer part acts as a directly connected consumer to an OMM Provider, via port 14003, and the provider part is directly connected to another OMM Consumer, via port 14002. The Hybrid Consumer and Hybrid Provider do not need to send messages to each other, because it is one application.

A Hybrid Provider's communication with a Consumer is implemented using the same pattern as the Interactive Provider (described in Section 1.1). The Hybrid Consumer's communication with Interactive Provider is the same as a Consumer (described in Section 12.2). The "backbone" (i.e., passing message from Hybrid Provider to Hybrid Consumer, and vice versa) portion of the hybrid application is different from previously described applications. This portion of hybrid application is the focus of this section.

Hybrid applications use one common session for both the consumer and provider. Thus, a session configured for the above example contains two connections. The consumer connection is of the RSSL type, and specifies port 14003 (i.e., the port of the serving provider). The provider connection is RSSL\_PROV type, and sets port number to 14002. The ports are specific for this example, and the applications should ports that are available.

## 12.5.2 Hybrid Task Overview

The following table lists the scenarios applicable to a hybrid application. The RFA Provider is assumed to be on the left of the hybrid application, and the RFA Consumer on the right. The → arrow represents an event, and points in the direction of the event.

HYBRID APPLICATION		
←	Hybrid Provider and Hybrid Consumer: Startup	→
	<ul style="list-style-type: none"> <li>Create a session using the <code>Session</code> interface</li> <li>Create <code>OMMProvider</code> and <code>OMMConsumer</code> event sources</li> <li>For each event source (<code>OMMProvider</code> and <code>OMMConsumer</code>) do the following: <ul style="list-style-type: none"> <li>Create <code>EventQueue</code> to receive the events</li> </ul> </li> </ul> <ol style="list-style-type: none"> <li>Create <code>OMMErrorIntSpec()</code></li> <li>Invoke <code>registerClient()</code> on event source; This returns the Error handle</li> </ol>	
	3. Hybrid Provider : Listening port setup	
	<ul style="list-style-type: none"> <li>Receive an <code>OMMListenerEvent</code> with the successful state</li> </ul>	
→	Hybrid Provider : Accepting/Rejecting Client Session	
	<ul style="list-style-type: none"> <li>Receive an <code>OMMActiveClientSessionEvent</code> with the <code>ClientSessionHandle</code></li> <li>Create an <code>OMMClientSessionIntSpec</code> with the Client Session Handle</li> <li>Call <code>registerClient()</code> on an <code>OMMProvider</code> event source; this returns the client session handle</li> </ul> <p>OR</p> <ul style="list-style-type: none"> <li>Invoke <code>submit()</code> (i.e., <code>submit(OMMInactiveClientSessionCmd)</code>)</li> </ul>	
→	Hybrid Provider : Receive messages from Consumers (Open Requests / Change Requests / Close Requests / Generic Messages / Post Messages)	
	<ul style="list-style-type: none"> <li>Receive an <code>OMMSolicitedItemEvent</code> with an <code>OMMMsg</code> and Request Token</li> </ul>	
	Hybrid Provider to Hybrid Consumer : Process Messages from Consumer	
	Get the OMM message, and if needed, decode and reencode it, then pass it to the Hybrid Consumer	
	Hybrid Consumer : Send messages to Provider	→
	<ul style="list-style-type: none"> <li>Create an <code>OMMItemIntSpec</code> interest specification</li> <li>Encapsulate an <code>OMMItemIntSpec</code> with the OMM message from Hybrid Provider</li> <li>Invoke <code>registerClient()</code> on the <code>OMMProvider</code> event source</li> </ul>	
	Hybrid Consumer : Receive Events from Provider	←
	<ul style="list-style-type: none"> <li>Receive <code>OMMItemEvents</code>; Receive <code>OMMCmdErrorEvents</code></li> </ul>	
	Hybrid Consumer to Hybrid Provider : Process Events	
	<ul style="list-style-type: none"> <li>Get the OMM message, and if needed, decode and reencode it, then pass it to the Hybrid Provider</li> </ul>	
←	Hybrid Provider : Send message (Unsolicited Refresh Response / Update Response)	
	<ul style="list-style-type: none"> <li>Generate a Token</li> <li>Create an <code>OMMItemCmd</code> with the Token and an OMM message from Hybrid Consumer</li> <li>Invoke <code>submit()</code> (i.e., <code>submit(OMMItemCmd)</code>)</li> </ul>	
←	Hybrid Provider and Hybrid consumer : Closing application	→
	<ul style="list-style-type: none"> <li>Invoke <code>unregisterClient()</code> on <code>OMMProvider</code> and <code>OMMConsumer</code> event sources</li> <li>Clean up clients resources; clean up the application</li> </ul>	

ROBUST FOUNDATION API

ROBUST FOUNDATION API

Table 85 Hybrid Application Steps

### 12.5.3 Startup

A hybrid application creates the session and then creates the OMMConsumer and OMMProvider Event Sources using the same session. The startup process includes the consumer startup tasks, as described in Section 12.2.2, and the provider startup tasks, described in Section 0. The two event sources have separate event queues.

### 12.5.4 Listening Port Setup

The Hybrid Provider sets up a listening port similar to the an Interactive Provider, as described in Section 0.

### 12.5.5 Accepting/Rejecting Client Sessions

A Login request from a consumer is received by the Hybrid Provider and is processed similar to Interactive Provider receiving login request, as decribed in Section 12.3.6.

When a Hybrid Provider receives a request for a client session, it passes the login request to the Hybrid Consumer part of application. Depending on the hybrid application, the message may be updated by reencoding it with modified data, or it may be used as received. The login request message is then sent by the Hybrid Consumer to the provider.

The following example shows receiving a Login Request from a consumer and passing it to the provider.

```
void processLoginRequest(OMMMsg loginReq, Token token)
{
    OMMMsg reqMsg;
    if (_reencode)
        reqMsg = OMMMsgReencoder.getEncodedMsg(loginReq);
    else
        reqMsg = loginReq;

    _intSpec.setMsg(reqMsg);
    _loginHandle = _consumer.registerClient(_eventQueue, _intSpec, this, token);
}
```

#### Example 126: Processing a Login Request by Hybrid Application

In the above example, the token is used as a closure. The Hybrid Consumer will receive a response to this request from the provider. This response is similarly passed to the Hybrid Provider.

The following example shows receiving a Login response from a provider and passing it to the consumer:

```
void processLoginResp(OMMItemEvent event)
{
    OMMMsg msg = event.getMsg();
    Token token = (Token)event.getClosure();

    _providerServer.submitResp(msg, token, 2000);
    if (msg.isFinal())
        _parent.destroySession(_sessionHandle);
}
```

#### Example 127: Processing a Login Response by Hybrid Application

## 12.5.6 Receiving Messages From Consumers

A Hybrid Provider part of a hybrid application receives messages from a consumer, in a manner similar to an Interactive Provider. Refer to Section 12.3.7 for details. A Hybrid Provider passes the messages to the Hybrid Consumer. The Hybrid Consumer, in turn, sends the messages to the provider, similar to the Consumer application, as described in Sections 12.2.3, 12.2.4, and 12.2.5. The messages can be updated by the Hybrid application (i.e., reencoded with modified data).

The example below shows the hybrid application passing requests from consumer to provider:

```
void processRequest(OMMSolicitedItemEvent event)
{
    OMMMsg msg = event.getMsg();
    int msgType = msg.getMsgType();
    Token token = event.getRequestToken();

    if ((msgType == OMMMsg.MsgType.REQUEST)
    {
        OMMMsg reqMsg;
        if (_reencode)
            reqMsg = OMMMsgReencoder.getEncodeMsgfrom(msg, 1000);
        else
            reqMsg = msg;

        _intSpec.setMsg(reqMsg);

        // First check if this token is in the watchlist, if yes, call reissueClient
        ItemInfo itemInfo = (ItemInfo)_reqMap.get(token);
        if (itemInfo != null)
        {
            _consumer.reissueClient(itemInfo.handle, _intSpec);
        }
        else
        {
            Handle handle = _consumer.registerClient(_eventQueue, _intSpec, this, token);
            itemInfo = new ItemInfo();
            itemInfo.handle = handle;
            _reqMap.put(event.getRequestToken(), itemInfo);
        }
    }
    else if (msgType == OMMMsg.MsgType.CLOSE_REQ)
    {
        ItemInfo itemInfo = (ItemInfo)_reqMap.remove(token);
        if (itemInfo != null) {
            _consumer.unregisterClient(itemInfo.handle);
        }
    }
}
```

**Example 128: Passing a Message from a Consumer to a Provider in a Hybrid Application**

## 12.5.7 Receiving Events From Providers

The Hybrid Consumer part of a hybrid application receives events from providers in a manner similar to a consumer application receiving events. Refer to Section 12.2.6 for details. A Hybrid Consumer passes the messages to a Hybrid Provider. A Hybrid Provider, in turn, sends the messages to a consumer, similar to an Interactive Provider application, as described in Section 12.3.8. The messages can be updated by the Hybrid application (i.e., reencoded with modified data).

The example below shows a hybrid application passing messages from a provider to a consumer.

```
void processResponse(OMMItemEvent event)
{
    OMMMsg msg = event.getMsg();
    Token token = (Token) event.getClosure();

    if (msg.isFinal())
        ItemInfo itemInfo = (ItemInfo)_reqMap.remove(token);

    _providerServer.submitResp(msg, token);
}
```

### Example 129: Passing Messages From a Provider to a Consumer In Hybrid Applications

## 12.5.8 Closing Hybrid Applications

A hybrid application should gracefully close all objects that it instantiated during initialization. Cleanup of objects is generally done in the opposite order they were created/initialized. Interest in command error events is unregistered by invoking `unregisterClient()` and passing in the error handle. The application may invoke the `destroy()` method on an Event Source without having closed all event streams. In this case, RFA internally unregisters all open event streams. Refer to Sections 12.2.9 and 12.3.11 for details on closing Consumer and Interactive Provider applications respectively.

## Chapter 13 RFA Feature Details

### 13.1 Feature Detail Overview

The sections below contain detailed descriptions of features supported by RFA. In addition to code snippets, there are also references to the related example applications, when applicable.

The picture below provides an example of a basic configuration tree that will be referenced when specific features are described.

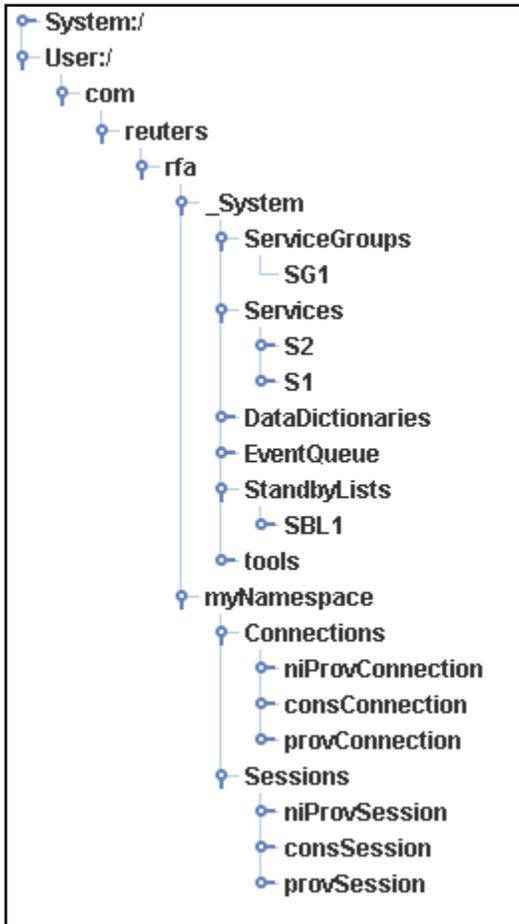


Figure 50: A Basic Configuration Tree

For example, if an RFA Consumer connection requires configuration parameters that support the feature, it will be shown as follows:

#### consConnection

```

|
newParameter1
newParameter2
  
```

## 13.2 Batch

### 13.2.1 Overview

Refer to Section 3.2.3 for introduction of the “Batch” concept.

- **Batch Requests** allow a consumer application to specify interest in multiple items of the same domain (e.g. Market Price) via a single request message. As a result, the application establishes multiple event streams, with each stream related to a requested item. Response messages are received on these event streams.
- **Batch Reissue** allows a consumer application to modify multiple event streams, or request refresh responses through one reissue operation.
- **Batch Close** allows a consumer application to close multiple event streams through a single close request.

The purpose of this feature is to minimize the upstream bandwidth.

### 13.2.2 Configuration

Configuration support for the Batch feature is described below:

#### consConnection

|

#### throttleBatchCount

The Batch feature uses the **throttleBatchCount** configuration parameter, a configuration element defined for each RSSL-type connection. The default for this element is 10. The value of **throttleBatchCount** defines the maximum batch items included into the batch request message that goes on the wire.

When an application requests a batch and the number of items exceeds the **throttleBatchCount**, RFA internally will split the message in several batch request messages, with each request containing no more items than the parameter. Splitting the batch message is called sub-batching.

Sub-batching applies only to request messages that request refresh responses (i.e., an initial open request and a reissue with refresh).

### 13.2.3 OMM Interface Support

The OMMMsg interface defines an indication flag **OMMMsg.Indication.BATCH\_REQUEST**. When the flag is set, the request message contains payload that contains items names for the batch register.

### 13.2.4 Session Package Support

The OMMConsumer interface defines the following methods:

- `registerClient(EventQueue aQueue, InterestSpec anInterest, Client aClient, Object aClosure)` to support Batch.
- `reissueClient(List<Handle> aHandleList, InterestSpec anInterest)` to support Batch Reissue.
- `unregisterClient(List<Handle> aHandleList, InterestSpec anInterest)` to support Batch Close.

### 13.2.5 Details

RFA supports all of these Batch features. If RFA connects to a provider that does not support one or more of these batch features, instead of sending a batch request, RFA will split it into individual requests. The RFA consumer interface uses the **RDMUser.Batch.REQUEST**, **RDMUser.Batch.REISSUE**, and **RDMUser.Batch.CLOSE** mask values from the **SupportBatchRequests** element of the login response to determine whether a provider supports batch requests, batch reissues, and batch closes. For batch features that a provider does not support, RFA will split the batch into individual requests.

For more details on the Login response message, refer to the *RFA Java RDM Usage Guide*.

### 13.2.5.1 Batch Register

A consumer application can specify interest in multiple items of the same domain through a single request message. The batch request is similar to an item request, with a few differences:

- The `OMMMsg.Indication.BATCH_REQUEST` indication flag is set.
- The `itemName` in the `OMMAttribInfo` is set to null.
- The item names are encoded into the payload of the request message.

Once the `OMMItemIntSpec` is built, the consumer application calls `OMMConsumer.registerClient()`. An event stream will be opened for each valid item. Response messages for an item are received on the item's event streams.

The "batch" handle returned from the `OMMConsumer.registerClient()` is for status only and cannot be used to reissue or close.

### 13.2.5.2 Batch Register Examples

The code example below shows how to make a batch request with three items.

```
// assume _mainApp is defined and set elsewhere.
// assume _numItemsRequested is defined elsewhere as ArrayList<Handle>
// Refer to examples.com.reuters.rfa.examples.omm.batchviewcons

public void sendRequest()
{
    String itemNames = "ABC.O, BCA.N, XYZ.X";
    String serviceName = "IDN_RDF";
    String mmt = "MARKET_PRICE";
    short capability = RDMMsgTypes.msgModelType(mmt);
    int indicationFlags = 0;

    //Note: "," is a valid character for RIC name.
    //This application need to be modified if RIC names have ",".
    String[] itemNamesList = itemNames.split(",");

    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();

    //Preparing to send item request message
    OMMPool pool = _mainApp.getPool();
    OMMMsg ommmsg = pool.acquireMsg();

    ommmsg.setMsgType(OMMMsg.MsgType.REQUEST);
    indicationFlags |= OMMMsg.Indication.REFRESH;
    ommmsg.setMsgModelType(capability);
    ommmsg.setPriority((byte) 1, 1);
    // Note: item name not needed for a batch request.
    ommmsg.setAttribInfo(serviceName, null, RDMinstrument.NameType.RIC);

    List<String> itemsList = new ArrayList<String>(itemNamesList.length);
    System.out.println("Application subscribing items for Batch:");
    for(int i=0; i < itemNamesList.length; i++)
    {
        itemsList.add(itemNamesList[i].trim());
        _numItemsRequested++; // keep track of the requests for processEvent()
        System.out.println("        " + itemNamesList[i].trim());
    }

    while (itemsList.size() > 0)
    {
        //Set the message into interest spec
        ommItemIntSpec.setMsg(encodeRequestPayload(ommmsg, itemsList, indicationFlags));
        Handle itemHandle = _mainApp.getOMMConsumer().registerClient(
            _mainApp.getEventQueue(), ommItemIntSpec, this, null);
    }
}
```

```

    pool.releaseMsg(ommmmsg);
}

public OMMMsg encodeRequestPayload(OMMMsg ommmsg, List<String> itemList, int indicationFlags)
{
    int estimatedSize = itemList == null ? 0 : itemList.size()*10;
    if (estimatedSize > 64000)
        estimatedSize = 64000;
    estimatedSize += 200; // add room for non-batch encoded data (ElementList/Entry, Array)

    ommmsg.setIndicationFlags(indicationFlags | OMMMsg.Indication.BATCH_REQ);

    // encode itemList into payload.
    OMMEncoder encoder = _mainApp.getEncoder();
    encoder.initialize(OMMTypes.MSG, estimatedSize);
    encoder.encodeMsgInit(ommmmsg, OMMTypes.NO_DATA, OMMTypes.ELEMENT_LIST);
    encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
    encoder.encodeElementEntryInit(RDMUser.Feature.ItemList, OMMTypes.ARRAY);
    encoder.encodeArrayInit(OMMTypes.ASCII_STRING, 0);
    String itemName;
    while (itemList.size() > 0)
    {
        itemName = itemList.get(0);
        if ( (itemName.length() + encoder.getEncodedSize()) > estimatedSize)
        {
            break; // don't encode any more items in this payload.
        }
        else
        {
            encoder.encodeArrayEntryInit();
            encoder.encodeString(itemList.remove(0), OMMTypes.ASCII_STRING);
        }
    }
    encoder.encodeAggregateComplete(); //completes the array
    encoder.encodeAggregateComplete(); //completes the element list
    return (OMMMsg)encoder.acquireEncodedObject();
}

```

### Example 130: Batch Request

This example shows how to process the response messages and add new handles to the `_itemHandles` list.

```

// assume _mainApp and _numItemsRequested are defined and set elsewhere.
// assume _itemHandles defined elsewhere and set elsewhere as an ArrayList<Handle>.
// Refer to examples.com.reuters.rfa.examples.omm.batchviewcons

public void processEvent(Event event)
{
    if (event.getType() == Event.COMPLETION_EVENT)
    {
        System.out.println("Receive a COMPLETION_EVENT, " + event.getHandle());
        _itemHandles.remove(event.getHandle()); //update the handle list
        return;
    }

    System.out.println("Received Item Event...");
    if (event.getType() != Event.OMM_ITEM_EVENT)
    {
        System.out.println("ERROR: Received an unsupported Event type.");
        _mainApp.cleanup(-1);
    }
}

```

```

        return;
    }
    OMMItemEvent ie = (OMMItemEvent) event;
    OMMMsg respMsg = ie.getMsg();
    GenericOMMParser.parse(respMsg);

    // add the batch request item response to the item handle list for proper closing
    if(_numItemsRequested != 1
        && respMsg.getMsgType() == OMMMsg.MsgType.REFRESH_RESP
        && respMsg.isSet(OMMMsg.Indication.REFRESH_COMPLETE))
    {
        _itemHandles.add(event.getHandle());
    }
}

```

### Example 131: Adding Batch Handles

#### 13.2.5.3 Batch Reissue

A consumer application can modify multiple event streams through one reissue operation. To use batch reissue, the consumer application does the following:

- Builds an OMMItemIntSpec with updated stream information (priority, pause or resume, view, refresh).
- Builds a List of Handles of the event streams to be modified.
- Calls `OMMConsumer.reissueClient()`, passing the OMMItemIntSpec and the List of Handles, as input parameters.
- The indication flag `OMMMsg.Indication.BATCH_REQ` does not need to be set, since the List of Handles signify to `reissueClient()` that this is a batch type operation.

#### 13.2.5.4 Batch Reissue Example

The code example below shows how to make a batch reissue. This example simply changes the priority class and requests a refresh response.

```

// assume _mainApp is defined and set elsewhere.
// assume _itemHandles defined and set elsewhere as an ArrayList<Handle>.
// Refer to examples.com.reuters.rfa.examples.omm.batchviewcons

private void sendReissue()
{
    byte newPriorityClass = 2;
    int newPriorityCount = 1;
    String serviceName = "IDN_RDF";
    String mmt = "MARKET_PRICE";
    short capability = RDMMsgTypes.msgModelType(mmt);

    OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();

    //Preparing to send batch reissue message
    OMMPool pool = _mainApp.getPool();
    OMMMsg ommmsg = pool.acquireMsg();

    ommmsg.setMsgType(OMMMsg.MsgType.REQUEST);
    ommmsg.setMsgModelType(capability);
    ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);
    ommmsg.setPriority(newPriorityClass, newPriorityCount); // new priority class

    // Note: item name not needed for a batch reissue.
    ommmsg.setAttribInfo(serviceName, null, RDMInstrument.NameType.RIC);

    ommItemIntSpec.setMsg(ommmsg);

    _mainApp.getOMMConsumer().reissueClient(_itemHandles, ommItemIntSpec);
}

```

```
pool.releaseMsg(ommmsg);
}
```

### Example 132: Batch Reissue

#### 13.2.5.5 Batch Close

A consumer application can close multiple event streams through one unregister operation. The consumer application builds a List of Handles of the event streams to close and calls `OMMConsumer.unregisterClient()`. The indication flag `OMMMsg.Indication.BATCH_REQ` does not need to be set, since the List of Handles signifies to `unregisterClient()` that this is a batch type operation.

#### 13.2.5.6 Batch Close Example

The code example below shows how to make a batch close.

```
// assume _itemHandles is defined elsewhere as an ArrayList<Handle>.
// Refer to examples.com.reuters.rfa.examples.omm.batchviewcons

public void closeRequest()
{
    if(_itemHandles.size() > 0)
        _mainApp.getOMMConsumer().unregisterClient(_itemHandles, (InterestSpec) null);
}
```

### Example 133: Batch Close

## 13.2.6 Application Examples

Two application examples are provided to illustrate the Batch feature: `omm.batchviewcons` and `omm.batchviewprov`.

## 13.3 Views

Refer to Section 3.2.4 for introduction of the “View” concept.

By default, all fields applicable to an item are returned by a provider in response to a request issued by a consumer. (A “full image” is returned by default.) However, a consumer application may not require all of the fields contained in a response. The View feature was introduced to afford applications the ability to specify a limited set of data when requesting an item. A ‘View’ is a constructed specifying a list of fields or elements that a user wants to receive for a particular item.

The View feature can be used for streaming requests or non-streaming requests, and it may be used for item and batch requests. In a batch request, the specified View is applied to all items contained in the batch.

This feature increases consumer application performance by:

- Reducing bandwidth usage (by reducing size of response message)
- Reducing decoding time by the client application (by reducing the number of entries to decode)

The following application examples illustrate the View feature: `omm.batchviewcons` and `omm.batchviewprov`.

### 13.3.1 OMM and RDM Interface Support

The OMM package provides the indication flag `OMMMsg.Indication.VIEW`, that when set, indicates the request message contains a payload with view information. Refer to Table 14 for the OMM Message Indication Flags summary.

View information contains view type and view data. The RDM package provides a `RDMUser.View` definition class. This class defines the `RDMUser.View.ELEMENT_NAME_LIST` and `RDMUser.View.FIELD_ID_LIST` view types for the view information containing a list of element names or a list of field IDs, respectively. If the view type is `RDMUser.View.ELEMENT_NAME_LIST`, the view data is an array of Strings representing element names. Otherwise, if the type is `RDMUser.View.FIELD_ID_LIST`, the view data is an array of integers representing field IDs. For details on the RDM interface, refer to the *RFA Java RDM Usage Guide*.

The OMM package provides error codes, such as `Code.FULL_VIEW_PROVIDED`, `Code.INVALID_VIEW`, and `Code.UNSUPPORTED_VIEW_TYPE`, to notify the application of an error condition. Refer to Table 65 for the description of the `OMMState.Code` definitions.

## 13.3.2 Details

Each RDM or Domain Message Model should describe how a specified View applies to a specific message model type, if at all. A view can be used for Refinitiv Domain Mode for any user-defined domain message model that has pairs of FieldID/FieldValue or ElementName/ElementValues, if the providing application supports the View feature and knows how to provide the data requested in the View.

RFA will pass a specified View from a consumer application to a provider, however it is up to provider to decide how to handle the View. Ideally, the provider sends responses that contain only the fields or elements specified by the View. However, if the provider cannot supply the exact set of requested fields, the provider can send back responses containing additional fields, or all fields. A request for a View is treated as a suggestion by a provider, but does not guarantee that the View will be fulfilled. As a result, the consumer can receive more fields than the view set for which it requested. It is the consumer client's responsibility to maintain a view definition for use in filtering subsequent response messages.

For performance reasons, a View longer than 99 fields is treated as a request for all fields. Additionally, a provider can set its own View limitations.

A client can dynamically change a View on an open item stream by reissuing a request with a new View. If there is a View flag present on a reissued request message, but no View definition in payload, the reissued request does not change the previously specified View, and responses will continue to have the same View. Although fields or elements in a View can be modified through a reissue, the **ViewType** cannot be changed after the stream is opened. To remove a view, the user can perform a reissue with no **OMMMsg.Indication.VIEW** set. All re-issue requests must contain a view definition, otherwise the request will be interpreted to mean the view should be removed, and all fields should be included in the next response.

After a failover/reconnect occurs, the previously requested View is recovered automatically by RFA or the server through the regular request caching/recovery scheme.

A login refresh response message from a provider includes information whether the provider supports a View. For more detail on the login **RespMsg**, refer to the *RFA Java RDM Usage Guide*. If the provider does not support View specifications, RFA will internally remove the **ViewFlag**, and send the request to the provider asking for a full image (all available fields), instead of sending the request for the specific View.

### 13.3.2.1 Creating Request Message With View

The process of submitting a view request follows the general request-making process as outlined in Section 12.2.3. The consumer request message should include the indication flag **OMMMsg.Indication.VIEW** and a payload encoded in an **ElementList**.

The code snippet below shows how to create a request with View. For the complete example refer to the [omm.batchviewcons](#) and [omm.batchviewprov](#) examples.

```
// set VIEW indication flag
ommmsg.setIndicationFlags(otherIndicationFlags | OMMMsg.Indication.VIEW);

// set ommmsg other data elements

// must use encoder to encode payload
encoder.initialize(OMMTypes.MSG, 100);
encoder.encodeMsgInit(ommmsg, OMMTypes.NO_DATA, OMMTypes.ELEMENT_LIST);
encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);

//Encode View Request
//1 - viewType:- RDMUser.View.FIELD_ID_LIST or RDMUser.View.ELEMENT_NAME_LIST
//2 - viewData:- Array of field ids or element names
// the view type is RDMUser.View.FIELD_ID_LIST
{
    encoder.encodeElementEntryInit(RDMUser.View.ViewType, OMMTypes.UINT);
    encoder.encodeUInt(RDMUser.View.FIELD_ID_LIST);
    encoder.encodeElementEntryInit(RDMUser.View.ViewData, OMMTypes.ARRAY);

    // assume viewData = {6, 22, 32}
    encoder.encodeArrayInit(OMMTypes.INT, 2);    // init array
        encoder.encodeArrayEntryInit();        // first entry
    encoder.encodeInt(6);
        encoder.encodeArrayEntryInit();        // second entry
    encoder.encodeInt(22);
        encoder.encodeArrayEntryInit();        // third entry
}
```

```

encoder.encodeInt(32);
encoder.encodeAggregateComplete();           //completes the array
    encoder.encodeAggregateComplete();       //completes the element list
}

```

### Example 134: Encoding View Request

#### 13.3.2.2 Creating Response Messages

An OMM Provider application supporting the View feature should handle request messages containing a View specification. First, the provider decodes the request message payload to get the View definition, and then the provider determines whether it supports the specified **ViewType**.

If the provider supports the **ViewType**, it sends only those fields or elements specified in the **ViewData** in subsequent response messages. However, if the provider does not support the **ViewType** or cannot filter for some reason, it has the option to send response messages with all fields (i.e., without a filter) or fields in addition to those specified in the **ViewData**.

The code snippet below demonstrates the provider handling a request with a view. For the complete example, refer to the [omm.batchviewcons](#) and [omm.batchviewprov](#) examples.

```

// check the presence of view flag in the received request message
if (! reqMsg.has(OMMMsg.Indication.VIEW)    // flag not set
{
    // requesting the full image
}
else if (reqMsg.getDataType() == OMMTypes.NO_DATA) // flag set, no payload
{
    // requesting the same view in a reissue message
}
else // flag set and payload included
{
    // decode the view to find which elements are requested
    // check if the provider supports the requested view
    // send response with requested data only
}
}

```

### Example 135: Handling a Request Message Containing a View

## 13.4 Pause and Resume

Refer to Section 3.2.7 for introduction of the “Pause/Resume” concept.

A display application may wish to “pause” data subscriptions that are not currently visible (hidden by other windows or tabs, or scrolled out of view). RFA provides the Pause and Resume set of features to pause and resume streams. In this section the Pause and Resume set of features will be referred to as PAR.

### 13.4.1 OMM Interface Support

The OMMMsg interface defines the following indication flag:

- **OMMMsg.Indication.PAUSE\_REQ**: Pause a stream.

### 13.4.2 Details

The provider communicates support for the Pause and Resume features via the LOGIN response’s **AttribInfo.Attrib** elements **SupportPauseResume** and **SupportOptimizedPauseResume** (for details, refer to the *RFA Java RDM Usage Guide*). When a provider receives a Pause on any domain, the data flow, with the exception of any state related messages, should be paused.

The Pause and Resume feature applies to the Market Price RDM only. It supports the following:

- Ability to pause (with refresh or without refresh) individual items on existing streams.
- Ability to resume (with refresh or without refresh) individual items.
- Pause all items associated with a Login Stream.
- Resume (without refresh) all items associated with a Login Stream.

Optimized Pause and Resume extends the existing Pause and Resume domain support to any domain, whether it is a customer defined domain message model or a RDM. In addition it supports the following functionality:

- Adds the ability to open new streams in a paused state.
- Allows for a single “Pause All” or “Resume All” message to be sent to the network, which should minimize the use of bandwidth with this functionality.

If a reissue occurs on a paused item stream and the `OMMMsg.Indication.PAUSE_REQ` is not specified, the stream will be resumed.

When the same item is requested multiple times on the same login stream, RFA will use one item stream to save bandwidth and fan out the responses to the handles of the multiple requests. This is known as multiple request entries. When the consumer application pauses an item, RFA will only pause the item stream if all MREs associated with the item stream are paused.

If an application intends to pause or resume a subset of items associated with a login stream, or a list of items from one or more login streams, it can alternately use the Batch Reissue feature. Refer to Section 13.2.5.3.

## 13.4.3 Application Examples

### 13.4.3.1 Pause an Item (Existing Stream)

```
//OMMConsumer _ommConsumer
//Handle _itemHandle;
//OMMPool _pool;
//String _serviceName
//String _itemName;

OMMMsg ommmsg = _pool.acquireMsg();

ommmsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommmsg.setMsgModelType(RDMMsgTypes.MARKET_PRICE);
ommmsg.setAttribInfo(_serviceName, _itemName, RDMInstrument.NameType.RIC);
ommmsg.setIndicationFlags(OMMMsg.Indication.PAUSE_REQ);

OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ommItemIntSpec.setMsg(ommmsg);
_ommConsumer.reissueClient(_itemHandle, ommItemIntSpec);

_pool.releaseMsg(ommmsg);
```

#### Example 136: Pausing Open Item Stream

### 13.4.3.2 Pause an Item (New Stream)

```
//OMMConsumer _ommConsumer
//OMMPool _pool;
//String _serviceName;
//String _itemName;
//EventQueue _eventQueue;

OMMMsg ommsg = _pool.acquireMsg();

ommsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommsg.setMsgModelType(RDMMMsgTypes.MARKET_PRICE);
ommsg.setAttribInfo(_serviceName, _itemName, RDMInstrument.NameType.RIC);
ommsg.setIndicationFlags(OMMMsg.Indication.REFRESH | OMMMsg.Indication.PAUSE_REQ);

OMMItemIntSpec omItemIntSpec = new OMMItemIntSpec();
omItemIntSpec.setMsg(ommsg);
Handle itemHandle = _ommConsumer.registerClient(_eventQueue, omItemIntSpec, this, null);

_pool.releaseMsg(ommsg);
```

#### Example 137: Pausing New Stream

### 13.4.3.3 Resume an Item

```
//OMMConsumer _ommConsumer
//Handle _itemHandle;
//OMMPool _pool;
//String _serviceName
//String _itemName;

OMMMsg ommsg = _pool.acquireMsg();

ommsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommsg.setMsgModelType(RDMMMsgTypes.MARKET_PRICE);
ommsg.setAttribInfo(_serviceName, _itemName, RDMInstrument.NameType.RIC);

OMMItemIntSpec omItemIntSpec = new OMMItemIntSpec();
omItemIntSpec.setMsg(ommsg);
_ommConsumer.reissueClient(_itemHandle, omItemIntSpec);

_pool.releaseMsg(ommsg);
```

#### Example 138: Resuming Item Stream

### 13.4.3.4 Resume an Item with Refresh

```
//OMMConsumer _ommConsumer
//Handle _itemHandle;
//OMMPool _pool;
//String _serviceName
//String _itemName;

OMMMsg ommsg = _pool.acquireMsg();

ommsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommsg.setMsgModelType(RDMMMsgTypes.MARKET_PRICE);
ommsg.setAttribInfo(_serviceName, _itemName, RDMInstrument.NameType.RIC);
ommsg.setIndicationFlags(OMMMsg.Indication.REFRESH);

OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ommsgItemIntSpec.setMsg(ommsg);
_ommConsumer.reissueClient(_itemHandle, ommItemIntSpec);

_pool.releaseMsg(ommsg);
```

#### Example 139: Resuming Item Stream With Refresh

### 13.4.3.5 Pause All Items Associated with a Login Stream

```
//OMMConsumer _ommConsumer
//OMMEncoder _ommEncoder
//Handle _loginHandle;
//OMMPool _pool;

_ommEncoder.initialize(OMMTypes.MSG, 500);

OMMMsg ommsg = _pool.acquireMsg();

ommsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommsg.setMsgModelType(RDMMMsgTypes.LOGIN);
ommsg.setAttribInfo(null, CommandLine.variable("user"), RDMUser.NameType.USER_NAME);
ommsg.setIndicationFlags(OMMMsg.Indication.PAUSE_REQ); //Pause all

_ommEncoder.encodeMsgInit(ommsg, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA);
_ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short) 0);
_ommEncoder.encodeElementEntryInit("ApplicationId", OMMTypes.ASCII_STRING);
_ommEncoder.encodeString(CommandLine.variable("application"), OMMTypes.ASCII_STRING);
_ommEncoder.encodeElementEntryInit("Position", OMMTypes.ASCII_STRING);
_ommEncoder.encodeString(CommandLine.variable("position"), OMMTypes.ASCII_STRING);
_ommEncoder.encodeAggregateComplete();

//Get the encoded message from the _ommEncoder
OMMMsg encMsg = (OMMMsg)_ommEncoder.getEncodedObject();

OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ommsgItemIntSpec.setMsg(encMsg);
_ommConsumer.reissueClient(_loginHandle, ommItemIntSpec);

_pool.releaseMsg(ommsg);
```

#### Example 140: Pausing All Streams

### 13.4.3.6 Resume All Items Associated with a Login Stream

```

//OMMConsumer _ommConsumer
//OMMEncoder _ommEncoder
//Handle _loginHandle;
//OMMPool _pool;

_ommEncoder.initialize(OMMTypes.MSG, 500);

OMMMsg ommmsg = _pool.acquireMsg();

ommmsg.setMsgType(OMMMsg.MsgType.REQUEST);
ommmsg.setModelType(RDMMsgTypes.LOGIN);
ommmsg.setAttribInfo(null, CommandLine.variable("user"), RDMUser.NameType.USER_NAME);

_ommEncoder.encodeMsgInit(ommmsg, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA);
_ommEncoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short) 0);
_ommEncoder.encodeElementEntryInit("ApplicationId", OMMTypes.ASCII_STRING);
_ommEncoder.encodeString(CommandLine.variable("application"), OMMTypes.ASCII_STRING);
_ommEncoder.encodeElementEntryInit("Position", OMMTypes.ASCII_STRING);
_ommEncoder.encodeString(CommandLine.variable("position"), OMMTypes.ASCII_STRING);
_ommEncoder.encodeAggregateComplete();

//Get the encoded message from the _ommEncoder
OMMMsg encMsg = (OMMMsg)_ommEncoder.getEncodedObject();

OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ommItemIntSpec.setMsg(encMsg);
_ommConsumer.reissueClient(_loginHandle, ommItemIntSpec);

_pool.releaseMsg(ommmsg);

```

#### Example 141: Resuming All Streams

## 13.5 Generic Messages

### 13.5.1 Overview

Refer to Section 5.8.3 for introduction of the “Generic message” concept.

Generic messages can be sent from an interactive provider to a consumer or from a consumer to an interactive provider or from a non-interactive provider to an ADH. The messages can be sent on an open event stream only. The generic message can contain any payload, from a simple byte buffer to a complex, nested hierarchy comprised of OMMData constructs.

The purpose of the Generic Messages feature is to provide a mechanism to pass user-defined information between consuming and providing applications, or RFA.

### 13.5.2 OMM Interfaces Support

The OMMMsg interface defines the `OMMMsg.MsgType.GENERIC` message type to support generic-type messages, and the `OMMMsg.Indication.GENERIC_COMPLETE` indication flag to support generic message fragmentation.

The OMMMsg interface defines a Secondary Sequence Number message element that is used in multi-part Generic messages. If this element is present, the `HAS_SECONDARY_SEQ_NUM` hint flag is set on the message.

For more information on OMM interface support, refer to Section 6.8.

### 13.5.3 Details

The generic message type is a part of the OMM messages suite supported by RFA. As one of the basic messages, the handling of generic messages has been described throughout this document in various sections.

Generic messages may be a single or multi-part. If the message is a single-part only, it contains the `OMMMsg.Indication.GENERIC_COMPLETE` indication flag. If the generic message contains multiple parts, only the last part contains the flag. In addition, the multi-part generic message contains a Secondary Sequence Number, which is used to reconstruct the order of the complete generic message.

#### 13.5.3.1 Sending Generic Messages in Consumer Applications

For details regarding the Session package's support for sending Generic Messages, refer to Section 12.2.5. To create a generic message, the message type must be set to `GENERIC`. The domain type is application-specific. The code example below shows an application creating the last segment of a multi-part generic message. The secondary sequence number is set to 2 (this implies that the message has two parts: 1 and 2). The flag indicating the last segment is set.

```
OMMMsg genericMessage = pool.acquireMsg();
genericMessage.setMsgType(OMMMsg.MsgType.GENERIC);
genericMessage.setMsgModelType(RDMMMsgTypes.DIRECTORY); // any Reuters or a user defined OMM
genericMessage.setSecondarySeqNumber(seqNum);
genericMessage.setIndicationFlags(OMMMsg.Indication.GENERIC_COMPLETE); // if the last part
```

#### Example 142: An OMM Consumer Application Creating a Generic Message

#### 13.5.3.2 Receiving Generic Message in Consumer Applications

Refer to Section 12.2.6.1 for details on Session support for receiving generic messages. A generic message is identified by the type `OMMMsg.MsgType.GENERIC`.

A generic message should be processed based on the message model type.

An application receiving a generic-type message should check if it contains the Secondary Sequence Number by testing the `HAS_SECONDARY_SEQ_NUM` hint flag. If it is present, the payload from the incoming messages should be ordered according to the number. The `OMMMsg.Indication.GENERIC_COMPLETE` indication flag indicates the last segment of a multi-part generic message.

#### 13.5.3.3 Sending Generic Message in Interactive Provider Applications

For details of the Session package's support for sending Generic Messages, refer to Section 12.3.8.

The procedure for creating a generic message in a provider application is the same as the procedure performed by a consumer application, as shown in Section 13.5.3.1.

#### 13.5.3.4 Receiving Generic Message in Interactive Provider Applications

For details of the Session package's support for receiving Generic Messages, refer to Section 12.3.8.

The procedure for creating a generic message by provider application is the same as the procedure performed by a consumer application, as shown in Section 13.5.3.2.

#### 13.5.3.5 Sending a Generic Message in Non-Interactive Provider Applications

The procedure for creating a generic message in a non-provider application is the same as the procedure performed by an interactive provider application, as shown in Section 13.5.3.3.

#### 13.5.3.6 Receiving a Generic Message in Interactive Provider Applications

The procedure for creating a generic message in a non-provider application is the same as the procedure performed by an interactive provider application, as shown in Section 13.5.3.4.

### 13.5.3.7 Using Generic Messages

Generic messages are available for the user-defined purposes. The use of the Generic message is defined on a per-Domain Message Model basis, allowing each model to define the meaning and semantics of the use of Generic messages. For message formats and for more details, refer to *RFA Java RDM Usage Guide*.

RFA utilizes Generic Messages in the following cases:

- Warm Standby generic message

RFA Consumer internally generates this message to notify the provider about its active or warm standby status. This generic message is sent on the Login event stream. The provider application should handle this message. For details, refer to the Warm Standby feature in Section 13.14.

- Source Mirroring generic message.

This message is generated by the Platform and received by a provider application. This message is informational and notifies the provider whether it is any of the following:

- An active provider with no hot standbys
- An active provider with hot standbys
- An inactive provider that is a hot standby

## 13.5.4 Application Examples

Two application examples are provided to illustrate the Generic Message feature: `omm.ommgenericmsgcons` and `omm.ommgenericmsgprov`.

## 13.6 Posting

### 13.6.1 Overview

For an overview of the “Post” concept, refer to Sections 3.2.8 and 3.3.1.2.

The OMM Post feature allows a consumer application to post OMM messages or OMM data to a provider application. The Post messages are sent to the provider on an open login stream or on an open item stream. The post message may request acknowledgement message in response.

The Posting feature significantly adds to RFA functionality by allowing for the contribution of market data by a consumer application.

### 13.6.2 Configuration

Configuration support for the Posting feature is shown below:

consConnection

|

**itemPostTimeoutInterval**

The Posting feature uses `itemPostTimeoutInterval`, an element defined for each Refinitiv Source Sink Library type connection. The default value for this element is 15000. The `itemPostTimeoutInterval` defines the maximum time in milliseconds RFA waits for an acknowledgement message in reply to a post. If the timer expires, RFA sends a status response message with the NACK code set to `OMMState.NackCode.NACK_NO_RESPONSE`. If the timer is set to 0, RFA waits infinitely.

### 13.6.3 OMM Interfaces Support

The OMMMsg interface defines the `OMMMsg.MsgType.POST` message type to support post type messages, and defines the `OMMMsg.MsgType.ACK_RESP` message type to support acknowledgement type messages. Refer to Sections 6.9 and 6.10 for more information on post and acknowledgement messages respectively.

The OMMMsg interface defines the following data and associated hint flags that are used in Post-type messages:

- A Sequence Number message element that is used in multi-part Post messages. If this element is present, the **HAS\_SEQUENCE\_NUMBER** hint flag is set on the message.
- ID element, used in multi-part Post messages and Post messages that requested acknowledgement. If this element is present, the **HAS\_ID** hint flag is set on the message.
- User Rights, and the **HAS\_USER\_RIGHTS** hint flag. Refer to Section 6.2.7 for details.

- Permission Expression, and **HAS\_PERMISSION\_DATA** hint flag. Refer to Section 13.6.3.5 for details.

A Post message may include the following indication flags: **OMMMsg.Indication.NEED\_ACK**, **OMMMsg.Indication.POST\_COMPLETE**, **OMMMsg.Indication.POST\_INIT**. Refer to Table 14 for the details regarding indication flags.

STREAM TYPE	ITEM NAME	SERVICE NAME / SERVICE ID
On stream posting (Item stream)	Optional	Optional
Off stream posting (Login stream)	Required	Required

**Table 86: Item Name/Service Name Requirements for Posting**

SINGLE PART MESSAGE	POST ID	SEQUENCE NUMBER
Ack required	Required	Optional
Ack not required	Optional	Optional

**Table 87: Requirements for Posting Single Part Messages**

MULTI-PART MESSAGE	POST ID	SEQUENCE NUMBER
Ack required	Required	Required
Ack not required	Required	Required

**Table 88: Requirements for Posting Multi-part Messages**

REQUIREMENT	INDICATION FLAGS
Single Part Message	OMMMsg.Indication.POST_INIT   OMMMsg.Indication.POST_COMPLETE
Multi Part Message – 1st part	OMMMsg.Indication.POST_INIT
Multi Part Message – last part	OMMMsg.Indication.POST_COMPLETE
Multi Part Message – middle part	
Ack Required	OMMMsg.Indication.NEED_ACK

**Table 89: Indication Flag Requirements for Posting**

If an error occurs during posting, the consumer application should receive a cmd error event message.

In the provider's response to a login request, the provider application advertises its ability to support the OMM Post feature. If any of its services supports posting, the provider advertises support of posting feature.

Posting is restricted only to services, thus posting to ServiceGroups is not supported. In the case of multiple connections, post messages will only be delivered on connections that support posting.

In general, RFA allows posting either an OMMMsg or OMMData. The format of the post is determined by the providing application. For example, RFA allows setting the attribute information on the post message, but it is the providing application that must accept the post with the attribute information. For On Stream posting, which will be defined in following section, a providing application may require that the post attribute information matches the item request attribute information. In such cases, a post may be rejected if the post contains an item attribute information, but the item request did not.

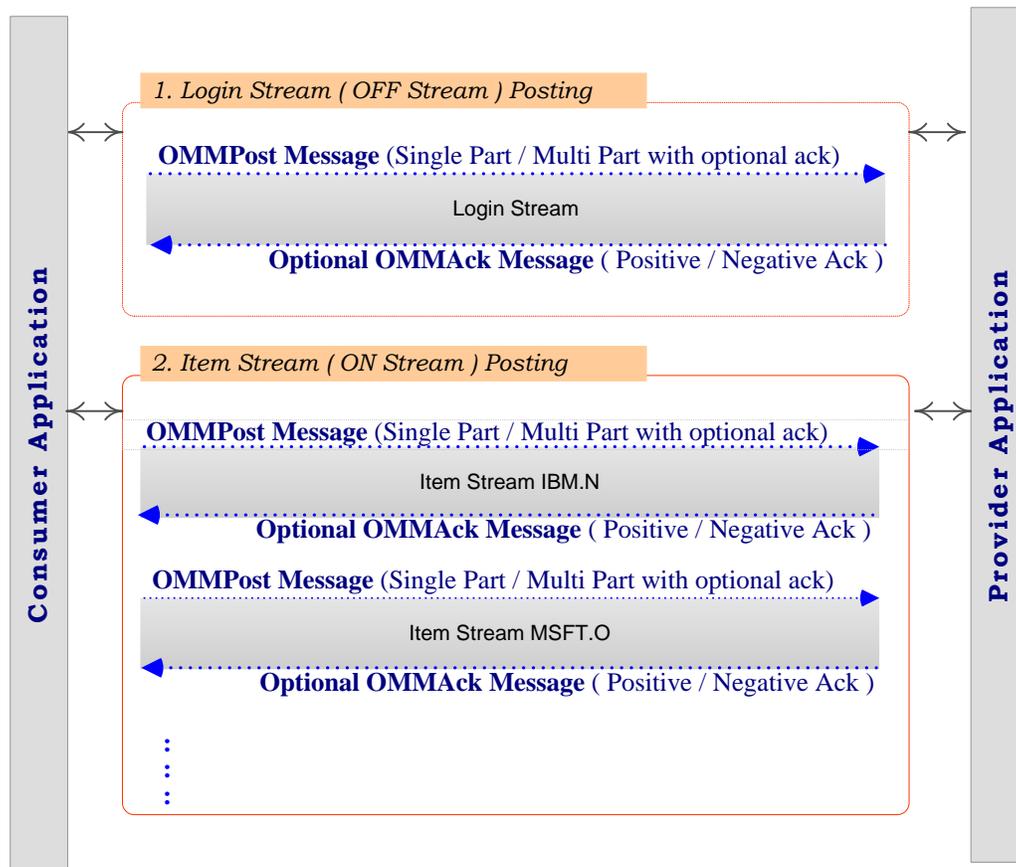
### 13.6.3.1 Consumer Applications Sending Post Messages

The RFA notifies consumer application that the posting is supported, regardless of the response from the provider. The posting feature is supported per service, and not per provider. If the message is sent to the service not supporting posting, the posting application will receive NACK in an acknowledge message in response.

For details of the Session package's support for sending Post Messages, refer to Section 12.2.5. The OMM Post feature allows a consumer application to post OMM Messages or OMM Data to a provider application. The OMMMsg or OMMDData object is contained by the payload of the Post message.

RFA supports posting on open streams of the following types:

- Item Stream, referred to as **On Stream Posting**. **On Stream Posting** requires an item stream to be instantiated for the item being posted prior to posting. The post message does not need to contain the item name and service name.
- Login Stream, referred to as **Off Stream Posting**. **Off Stream Posting** does not require an item stream to be instantiated. The post message must contain the item name and service name.



**Figure 51: Stream Types for Posting**

A Consumer application may request an acknowledge response from the receiving application by including the **OMMMsg.Indication.NEED\_ACK** flag. If this flag is set, the post message must also include a Post ID.

RFA supports single-part and multi-part message posting.

The single-part post message must include the **OMMMsg.Indication.POST\_INIT** and **OMMMsg.Indication.POST\_COMPLETE** flags.

There are three kinds of multi-part messages:

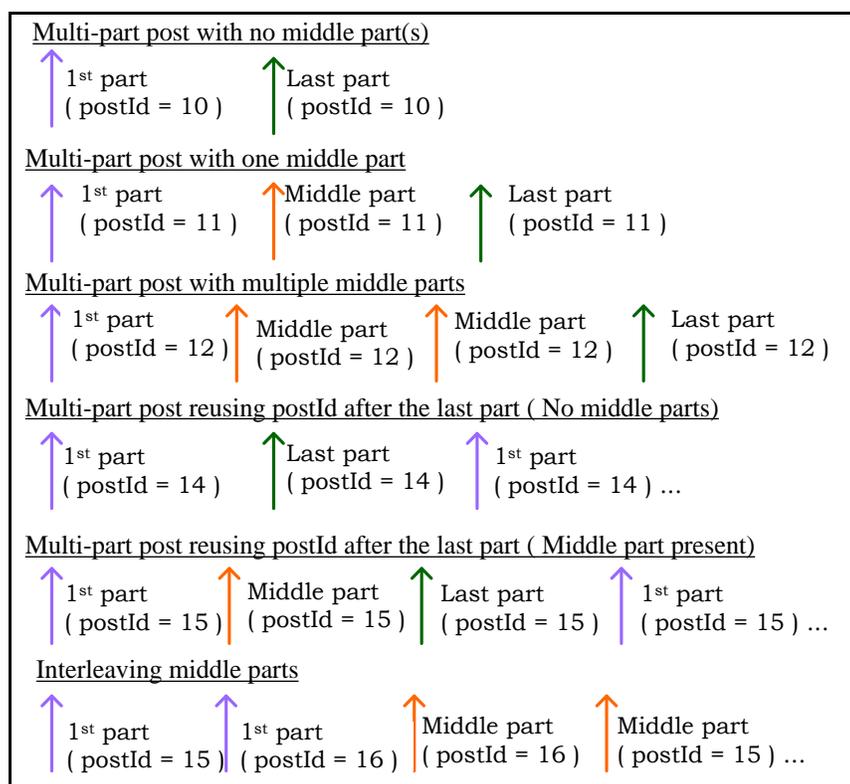
- The First part is the beginning part of a post. It must include the **OMMMsg.Indication.POST\_INIT** flag.
- Middle part: (optional) This indicates the middle parts of a post message.
- The last part or final part of the multi-part message must include the **OMMMsg.Indication.POST\_COMPLETE** flag.

Applications posting multi-part messages must use the following rules:

- All parts of a multi-part message must use the same postId.
- Every multi-part post must have one first part and one last part; middle parts are optional. Thus there can be none / one / many multi-part.

- The middle parts of different post Ids can be interleaved.

The diagrams below show different sequences that are supported by RFA when sending post messages:



**Figure 52: Multi-part Post Examples**

The example below illustrates setting the post related data on a post message. This example shows a single-part message that requires an acknowledge response.

```
_postMsg.setMsgModelType( RDMMsgTypes.MARKET_PRICE );
_postMsg.setMsgType(OMMMsg.MsgType.POST);
_postMsg.setMsgId( 2000 ); // must be set if ack required

// a single-part post message
int indicationFlags = OMMMsg.Indication.POST_COMPLETE | OMMMsg.Indication.POST_INIT;
// needs acknowledge
indicationFlags = indicationFlags | OMMMsg.Indication.NEED_ACK;

_postMsg.setIndicationFlags( indicationFlags );
_postMsg.setAttribInfo("DIRECT_FEED", "IBM.N", RDMInstrument.NameType.RIC);
```

**Example 143: A OMM Consumer Application Setting Parameters on Post Message**

### 13.6.3.2 Handling Post Message In Provider Application

Refer to Section 12.2.6.1 for details on the Session's support to receive post messages. A post message is identified by the type `OMMMsg.MsgType.POST`.

Post Messages can be received on an open item stream (*On Stream Posting*) or on a login stream after a login is granted (*Off Stream Posting*). The stream type is identified via the request token. The application retrieves the request token from the event using `getRequestToken()`.

A provider application has to check whether the sender of the post message requested an ack response. If the indication flag `OMMMsg.Indication.NEED_ACK` is set on the received post message, the provider application must send an acknowledge response message. The post message that requests the ack must include a Post ID. The presence of this element is identified through `OMMMsg.HAS_ID` flag.

The received post message can be an atomic post message, or a part of a multi-part post message. The multi-part post message contains a Sequence Number element, and its presence is identified with the `OMMMsg.HAS_SEQ_NUM` flag. A multi-part post message must also contain a Post ID. The presence of this element is identified through `OMMMsg.HAS_ID` flag. A multi-part post message may be the last part, which is identified by an `OMMMsg.Indication.POST_COMPLETE` indication flag. The application should read the Sequence Number and Post ID, and use the data to compose the complete post message.

`PublisherPrincipalIdentity` contains Publisher Information (i.e., the publisher's IP address and a publisher ID). You can obtain this information from a post message using `getPrincipalIdentity()`.

The post message payload, can contain either `OMMData` or an `OMMMsg`. You can get the payload using the `getPayload()` accessor method.

The code below demonstrates a provider application checking the posting type (on stream/off stream), checking whether the message is multipart, and if requires an ack response (and processing accordingly):

```
// process post message
if( ommMsg.getMessageType() == OMMMsg.MsgType.POST )
{
    // save login request token
    if( ommMsg.getMessageModelType() == RDMMMsgTypes.LOGIN )
    {
        if (ommMsg.getMessageType() == OMMMsg.MsgType.REQUEST)
        {
            _loginRequestToken = requestToken;
        }
    }
    // login stream (off stream posting)
    if( requestToken == _loginRequestToken )
        System.out.println( " Provider app received post message on login stream " );
    else
    // item stream (on stream posting)
        System.out.println( " Provider app received post message on item stream " );

    // ack response required
    if( ommMsg.isSet(OMMMsg.Indication.NEED_ACK) && ( ommMsg.has(OMMMsg.HAS_ID))
    {
        int postId = (int) ommMsg.getId();
        sendAck(postId); // application implemented
    }

    // check sequence
    if( ommMsg.has(OMMMsg.HAS_SEQ_NUM) && ( ommMsg.has(OMMMsg.HAS_ID))
    {
        int seqNumber = (int) ommMsg.getSeqNum();
        int postId = (int) ommMsg.getId();
        completePost(postId, seqNumber, OMMMsg.Indication.POST_COMPLETE); // application implemented
    }

    // get publisher details
    PublisherPrincipalIdentity pi = (PublisherPrincipalIdentity) ommMsg.getPrincipalIdentity();
    verifyPublisher(pi); // application implemented

    // get payload type
    short payloadDataType = ommMsg.getDataType();
    OMMData payload = ommMsg.getPayload();
    processPostPayload(payloadDataType, payload); // application implemented
}
...
```

#### Example 144: A Interactive Provider Application Processing Post Messages

A provider application can receive a post message with the payload as an `OMMMsg`, and the provider application can resend the `OMMMsg` to interested clients. If the payload is an `OMMMsg` of type `Update / Refresh / Status`, the provider application can set the publisher information on the messages before resending them. For refresh messages, the provider application must change the response type from `solicited` to `unsolicited`, if required.

The payload message can contain attribute information, but the service name might be null. In such cases, the provider application must explicitly set the service name. The code shown below shows the process of resending a message received as post message payload:

```
void forwardPostPayloadMessage(Token messageToken, OMMMsg ommPostMsg)
{
    OMMMsg ommPayloadMsg = (OMMMsg) ommPostMsg.getPayload();
    short messageType = ommPayloadMsg.getMsgType();
    PublisherPrincipalIdentity pi = null;
    OMMMsg reEncodedMessage = null;

    pi = (PublisherPrincipalIdentity) ommPostMsg.getPrincipalIdentity();
    OMMMsgReencoder.setEncodePublisherInfo(true, pi);

    String serviceName = ommPayloadMsg.determineServiceName() // application method
    OMMMsgReencoder.setServiceName(true, serviceName);

    if( messageType == OMMMsg.MsgType.REFRESH_RESP)
    {
        reEncodedMessage = OMMMsgReencoder.changeResponseTypeToUnsolicited( ommPayloadMsg, 1000);
    }

    reEncodedMessage = OMMMsgReencoder.getEncodeMsgfrom(ommPayloadMsg, 1000);

    _submitCmd.setMsg(reEncodedMessage);
    _submitCmd.setToken(messageToken);

    _provider.submit(_submitCmd, null);
}
```

### Example 145: Forwarding Post-Payload Messages

#### 13.6.3.3 Sending Acknowledge Messages In Provider Applications

For details of the `Session` package's support for sending Acknowledgement Messages, refer to Section 12.3.8.

A provider application sends an acknowledge message in response to a post message that has the `OMMMsg.Indication.NEED_ACK` indication flag set. An acknowledge response message should include a Post ID set to value of the Post ID from the corresponding Post message. The received post message may include a Sequence Number (i.e., the post is one part of multi-part post). If the Sequence Number is included in post message, it should also be included in the corresponding acknowledge response message.

The ack message indicates either a positive acknowledgement or negative acknowledgement. Absence of state indicates a positive ack. State information must be used to specify the ack type and can be set on the message directly using `setState()` or indirectly using `OMMState` object. Only code and text is applicable for specifying the ack type. Stream state and data state is not applicable.

Positive ack can be specified by one of the following:

- By not setting the state information
- By setting the code to `OMMState.NackCode.NONE`
- By not setting the code

A negative ack can be specified by setting the code to one of the codes from `OMMState.NackCode`, except for `OMMState.NackCode.NONE`.

The following table describes the usage of `OMMState` for setting the ack type.

\*: indicates the absence of information      ✓: indicates the presence of information

DESIRED ACK TYPE	CODE	TEXT	INTERFACE USAGE
Positive Ack	*	*	State Not Set
Positive Ack	✓	*	Setting state using <code>OMMState</code> object: <pre>OMMState state = _pool.acquireState(); state.setCode( OMMState.NackCode.NONE ); _ommAckMsg.setState( state );</pre> Setting state on <code>OMMMsg</code> object: <pre>_ommAckMsg.setState(OMMState.Stream.UNSPECIFIED,                     OMMState.Data.NO_CHANGE,                     OMMState.NackCode.NONE, null );</pre>
Positive Ack	*	✓	Setting state using <code>OMMState</code> object: <pre>OMMState state = _pool.acquireState(); state.setText( "all is well" ); _ommAckMsg.setState( state );</pre> Setting state on <code>OMMMsg</code> object: <pre>_ommAckMsg.setState(OMMState.Stream.UNSPECIFIED,                     OMMState.Data.NO_CHANGE,                     OMMState.NackCode.NONE, "all is well" );</pre>
Positive Ack	✓	✓	Setting state using <code>OMMState</code> object: <pre>OMMState state = _pool.acquireState(); state.setCode( OMMState.NackCode.NONE ); state.setText( "all is well" ); _ommAckMsg.setState( state );</pre> Setting state on <code>OMMMsg</code> object: <pre>_ommAckMsg.setState(OMMState.Stream.UNSPECIFIED,                     OMMState.Data.NO_CHANGE,                     OMMState.NackCode.NONE, "all is well" );</pre>
Negative Ack	✓	*	Setting state using <code>OMMState</code> object: <pre>OMMState state = _pool.acquireState(); state.setCode( OMMState.NackCode.NACK_GATEWAY_DOWN ); _ommAckMsg.setState( state );</pre> Setting state on <code>OMMMsg</code> object: <pre>_ommAckMsg.setState(OMMState.Stream.UNSPECIFIED,                     OMMState.Data.NO_CHANGE,                     OMMState.NackCode.NACK_GATEWAY_DOWN, null );</pre>
Negative Ack	✓	✓	Setting state using <code>OMMState</code> object: <pre>OMMState state = _pool.acquireState(); state.setCode( OMMState.NackCode.NACK_GATEWAY_DOWN ); state.setText( "down" ); _ommAckMsg.setState( state );</pre>

DESIRED ACK TYPE	CODE	TEXT	INTERFACE USAGE
			Setting state on OMMMsg object: <pre>_ommAckMsg.setState(OMMState.Stream.UNSPECIFIED,                     OMMState.Data.NO_CHANGE,                     OMMState.NackCode.NACK_GATEWAY_DOWN, "down" );</pre>

**Table 90: Setting ack Type Using OMMState**

The following example illustrates how to compose a negative acknowledgement response.

```
// set the message type
_ommAckMsg.setMsgType( OMMMsg.MsgType.ACK_RESP );

// set the post id;
_ommAckMsg.setId( ommPostMsg.getId() );

// set the sequence number if available on the post message
if( ommPostMsg.has(OMMMsg.HAS_SEQ_NUM) )
    _ommAckMsg.setSeqNum( ommPostMsg.getSeqNum() );

// set the state with the denied Nack code
_ommAckMsg.setState( OMMState.Stream.UNSPECIFIED,      OMMState.Data.NO_CHANGE,
                    OMMState.NackCode.NACK_DENIED_BY_SRC,      "Ok");
```

### Example 146: An OMM Interactive Provider Application Creating a Negative Acknowledgement to a Post

#### 13.6.3.4 Processing Acknowledge Message in Consumer Application

For details of the Session package's support for receiving Acknowledgement Messages, refer to Section 12.3.8.

An ack message contains the Post ID that came with the post, and also contains a sequence number, if one was included in the initial post message.

An ack message can indicate either a positive acknowledgment or negative acknowledgment. This information is contained in the `OMMState` object obtained using `getState()` on the message. Absence of the state object indicates a positive acknowledgement. The following table contains details on processing ack messages:

ACK TYPE RECEIVED	INFORMATION AVAILABLE	STATE INFORMATION
Postive Ack	State Not Available	
Postive Ack	State Not Available <ul style="list-style-type: none"> <li>• <code>OMMState.isNackCode() = TRUE</code></li> <li>• <code>OMMState.NackCode = NONE</code></li> </ul>	Stream State = <code>OMMState.Stream.UNSPECIFIED</code> Data State = <code>OMMState.Data.NO_CHANGE</code> Code = <code>OMMState.NackCode.NONE</code> Text = Available or empty
Negative Ack	State Not Available <ul style="list-style-type: none"> <li>• <code>OMMState.isNackCode() = TRUE</code></li> <li>• <code>OMMState.NackCode != NONE</code></li> </ul>	Stream State = <code>OMMState.Stream.UNSPECIFIED</code> Data State = <code>OMMState.Data.NO_CHANGE</code> Code != <code>OMMState.NackCode</code> Text = Available or empty

**Table 91: Identifying the ack Type from OMMState Information**

- The state information 'stream state' and 'data state' is not applicable for ack messages, and should be ignored.
- The state code indicates a positive ack or negative ack. A state code of `OMMState.NackCode.NONE` indicates a positive ack, while the rest indicate a negative ack. You can use the `isNackCode()` method on the state object to ensure the code is a Nack Code.
- The text may contain a string or be empty.
- An ack message may contain attribute information and/or payload information which be decoded using the application implemented methods

The following code illustrates how to process an ack message:

```

if (respMsg.getMsgType() == OMMMsg.MsgType.ACK_RESP)
{
    // get post id
    int PostId = respMsg.getId();

    // get sequence number if available
    if (respMsg.has(OMMMsg.HAS_SEQ_NUM) )
    int SeqNo = respMsg.getSeqNum();

    // check for state information availability
    if ( ! respMsg.has(OMMMsg.HAS_STATE))
    {
        // positive ack
    }
    else
    {
        // get state information
        OMMState status = respMsg.getState();
        if ((status.isNackCode() == true) && (status.getCode() == OMMState.NackCode.NONE ))
        {
            // positive ack
        }
        else
        {
            // negative ack
        }
    }
}

```

### Example 147: Processing Ack Messages

#### 13.6.3.5 Post User's Rights

The Post message optionally contains `OMMMsg.UserRights` data. This data contains information whether the user has the ability to create the record, delete it, or modify permission data. Refer to Section 6.2.7 for the **UserRights** flags.

#### 13.6.3.6 Permission Expression

When content is being contributed, the permission data included in the Post message is used to permission the user that is posting the information. If the payload of the Post message is another message type (i.e., refresh message) that contains permission data, the nested message's permission data can be used to change the permission expression associated with the item being posted. If the permission data for the nested message is the same as the permission data on the Post message, no permission data is necessary on the nested message.

The permission data is used in conjunction with the `OMMMsg.UserRights`. (refer to previous Section 13.6.3.5). The **UserRights** indicate whether the posting user is allowed to create or destroy items in the cache of record. This is also used to indicate whether the user has the ability to change the `permData` associated with an item in the cache of record.

### 13.6.4 Application Examples

Two application examples are provided to illustrate Posting feature: `examples.com.reuters.rfa.example.omm.postingConsumer` and `examples.com.reuters.rfa.example.omm.postingProvider`.

## 13.7 Private Streams

### 13.7.1 Overview

Refer to Sections 3.3.1.3 and 3.2.12 for introduction of the *Private stream* concept.

Using a Private Stream, a Consumer application can create a virtual private connection with an Interactive Provider. A virtual private connection is made up of the existing individual point-to-point and multicast connections in the system. Messages exchanged via a Private Stream flow between a Consumer and an Interactive Provider using these existing underlying connections. However, unlike a regular stream, these messages are not fanned out by the RFA or by any Refinitiv Real-Time Distribution System components to be shared with other consumers or providers. Refer to Concepts Section for details. This feature adds important functionality to the RFA suite.

### 13.7.2 OMM Interfaces Support

The OMM package provides an indication flag `OMMMsg.Indication.PRIVATE_STREAM`. This flag is set on request, refresh and status type messages that are part of this event stream. Refer to Table 14 for the OMM Message Indication Flags summary.

### 13.7.3 Details

The consumer application starts the Private Stream by registering with request message that has the `OMMMsg.Indication.PRIVATE_STREAM` flag set. The example code below shows how to set a Private Stream in an application.

```
Msg.setIndicationFlags(OMMMsg.Indication.PRIVATE_STREAM);
```

#### Example 148: Setting Private Stream

The private stream is established between one consumer and one interactive provider. Any type of requests, functionality, or Domain Models can flow across a Private Stream. The list includes:

- Streaming Requests
- Snapshot Requests
- Posting
- Generic Messages
- Batch Requests
- Views
- All RDMS & Custom Domain Models.

The RFA does not support recovery for a Private Stream. In the case of connection failure, the consumer application is notified and has to handle recovery by resending the request.

The Warm Standby does not support private streams. The items opened as a Private Stream are closed during failover.

#### 13.7.3.1 Normal to Private Stream Item-Based Recall

The public to private stream recall feature allows a consumer application to learn if a particular item may and should be opened as a private stream only.

The private stream only restriction is enforced by the provider application, which based on the user's credentials and its permission may decide to allow to open this item as requested (public or private) or inform the consumer application about the restriction if the item was requested as public. For this, the stream is closed and the provider application uses a status message with status of Closed Redirect with the `OMMMsg.Indication.PRIVATE_STREAM` flag set and additional text information.

In response to this status message, the consumer application may attempt to open this item as private stream. The status message will contain the information (in the `AttribInfo`) for the consumer application to re-request.

## 13.7.4 Application Example

Two application examples are provided to illustrate the Private Stream feature: **StarterConsumer\_PrivateStream** and **StarterProvider\_PrivateStream**.

## 13.8 Visible Publisher Identifier

Visible publisher identifier in OMM is stored in the `PublisherPrincipalIdentity` class. This class provides both the publisher ID and the publisher address. You can obtain the Visible Publisher Identifier (VPI) in OMM from the `OMMMsg` of `OMMMsg.MsgType.Post` type and `OMMMsg.MsgType` response types (such as `OMMMsg.MsgType.REFRESH_RESP` and `OMMMsg.MsgType.UPDATE_RESP`).

### 13.8.1 OMM Interfaces Support

OMMConsumer applications and OMM provider applications can both get the VPI from the `OMMMsg` interface. The OMM package provides the indication flag `OMMMsg.HAS_PUBLISHER_INFO` which can verify the presence of VPI.

### 13.8.2 Details

Using RFA you can set VPI only on the `OMMMsg` of `OMMMsg.MsgType` response types. The OMM publisher chooses whether to set the Visible Publisher Identifier on the response message. If the consumer gets data from an upstream publisher which is an intermediary device in turn getting data from an upstream source, then the intermediary device will route the VPI set on the `PostMsg` to the upstream source. Finally, the ultimate publisher in the upward chain decides whether to set the VPI on the responses that it publishes. While processing a post message received from the posting device, the OMM publisher application gets the VPI from the post message and can include the VPI in the response message, when sending the posted content to the downstream device.

RFA now has an interface to set VPI (Publisher ID and Publisher address) on `PostMsgs`. If this interface is not set, RFA internally populates the VPI on post messages submitted by the OMM consumer application and sends the post out to the network.

In addition to being able to access VPI via RFA post messages and response messages, OMM applications can also obtain the VPI via Field Identifiers defined from the publisher component. For further details, refer to the publishing component's documentation.

### 13.8.3 Application Example

RFA includes two application examples to illustrate the VPI feature: **StarterConsumer\_Post** and **StarterProvider\_Post**.

## 13.9 Enhanced Symbol List

For an introduction to the "Symbol List" concept, refer to Section 3.2.5.

By default, the Symbol List domain message model provides access to a list of symbol names, accessed via a request for a well-known symbol list name. However, the enhanced symbol list request allows the user to specify to open all item streams from the name list in user's symbol list request. This can be achieved by defining symbol list item stream behaviors through the symbol list request message payload.

RFA can advertise to the application whether it supports the symbol list enhancement feature through Login domain response messages.

The application example (`omm.symbollist`) illustrates the symbol list enhancement feature.

## 13.9.1 OMM and RDM Interface Support

The Client can use the symbol list request payload to define symbol list item stream behavior. The RDM package provides the `RDMUser.SymbolList` definition class. This class defines `RDMUser.SymbolList.SymbolListBehaviors` for symbol list behaviors containing `RDMUser.SymbolList.DataStreams` for item data stream definitions (for details, refer to the *RFA Java RDM Usage Guide*)

For example, `RDMUser.SymbolList.DataStreams` is an element name in the element list in the payload. The element's data can be `RDMUser.SymbolList.SYMBOL_LIST_NAMES_ONLY`, `RDMUser.SymbolList.SYMBOL_LIST_DATA_STREAMS`, or `RDMUser.SymbolList.SYMBOL_LIST_DATA_SNAPSHOTS`. If this element is absent or the element's data is set to `RDMUser.SymbolList.SYMBOL_LIST_NAMES_ONLY`, the default behavior occurs, and the application receives only the symbol list response with a list of item names and no item data stream is opened. If the element data is set to `RDMUser.SymbolList.SYMBOL_LIST_DATA_STREAMS`, the application will receive a list of item names plus all item streaming data from the list of names. If the element data is set to `RDMUser.SymbolList.SYMBOL_LIST_DATA_SNAPSHOTS`, the application will receive a list of item names plus all item non-streaming data from the list of names.

Consumers can know whether RFA supports the symbol list enhancement feature via the LOGIN response's `AttribInfo.Attrib` element `SupportEnhancedSymbolList` (for details, refer to the *RFA Java RDM Usage Guide*).

## 13.9.2 Details

If the consumer wants to open a symbol list request for item names plus item data, the consumer can specify item data stream definitions in the payload of a symbol list request. Upon receiving symbol list response messages from the provider with a list of item names, RFA will internally open all items from the list by sending Batch requests or individual item requests to the provider on behalf of the consumer. The consumer application will expect to receive symbol list responses and individual item responses. The consumer application can retrieve the individual item handle from the first unsolicited refresh or the first status response message on the item stream like the Batch feature (for details on how to retrieve item handles, refer to Section 13.2.5.1). The enhancement assumes all item data streams opened by RFA will be Market Price domain. Refresh messages on item streams opened from the symbol name list are unsolicited refreshes.

After item data streams from the list of names are opened by RFA, all data streams act exactly the same as single item request streams opened by the consumer application. The consumer can reissue and/or close these streams as well. If there is a failure on the stream, RFA will recover the item stream in the same manner as it recovers any single open item stream.

If there are other updates received on the symbol list stream, RFA will open only those items which were never opened before. RFA ignores item names of "Delete" or "Update" action from updates on the symbol list stream.

RFA throws exceptions on invalid item data stream definitions or reissues on a symbol list stream with payloads.

### 13.9.2.1 Creating a Request Message for Item Names and Data

The process of submitting a symbol list request follows the general request-making process as outlined in Section 12.2.3. The consumer request message can specify a symbol list item stream definition in the payload encoded in an `ElementList`.

The code snippet below shows how to create a symbol list request with payload. For the complete example, refer to `omm.symbolList`.

```

OMMMsg msg = _pool.acquireMsg();
msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setModelType(RDMMsgTypes.SYMBOL_LIST);

//skip more code here...

//specify a symbol list item data definition below
_encoder.initialize(OMMTypes.MSG, 500);
_encoder.encodeMsgInit(msg, OMMTypes.NO_DATA, OMMTypes.ELEMENT_LIST);
_encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);

_encoder.encodeElementEntryInit(RDMUser.SymbolList.SymbolListBehaviors, OMMTypes.ELEMENT_LIST);
_encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);
_encoder.encodeElementEntryInit(RDMUser.SymbolList.DataStreams, OMMTypes.UINT);

if ( dataStreams == RDMUser.SymbolList.SYMBOL_LIST_DATA_STREAMS )
    _encoder.encodeUInt(RDMUser.SymbolList.SYMBOL_LIST_DATA_STREAMS);
else if ( dataStreams == RDMUser.SymbolList.SYMBOL_LIST_DATA_SNAPSHOTS )
    _encoder.encodeUInt(RDMUser.SymbolList.SYMBOL_LIST_DATA_SNAPSHOTS);
else if ( dataStreams == RDMUser.SymbolList.SYMBOL_LIST_NAMES_ONLY )
    _encoder.encodeUInt(RDMUser.SymbolList.SYMBOL_LIST_NAMES_ONLY);
else // not support
    return null;

_encoder.encodeAggregateComplete();
_encoder.encodeAggregateComplete();
OMMMsg newMsg = (OMMMsg)_encoder.getEncodedObject();

_spec.setMsg(newMsg);

//skip more code here...

Handle handle = _ommConsumer.registerClient(queue, _spec, client, null);

```

### Example 149: Encoding a Symbol List Request

#### 13.9.2.2 Handling Response Messages

An OMM Consumer application expects to receive response messages on a symbol list stream. If the symbol list request requires item names plus item data, the OMM Consumer application also expects response messages on individual item streams opened from the list of names.

The code snippet below demonstrates the consumer handling response messages on individual item streams. For a complete example, refer to [omm.symbollist](#).

```

// check the presence of view flag in the received request message
void processOMMMsg(Event event)
{
    OMMMsg msg = ((OMMItemEvent)event).getMsg();
    switch (msg.getMsgType())
    {
        case OMMMsg.MsgType.REFRESH_RESP:
        case OMMMsg.MsgType.UPDATE_RESP:

            if ( msg.getMsgModelType() == RDMMMsgTypes.MARKET_PRICE )
                _itemManager.processEventFromSymbolListClient(event);

        //skip more code here
    }
}

public void processEventFromSymbolListClient(Event event)
{
    //skip more code here

    OMMItemEvent ommItemEvent = (OMMItemEvent)event;
    Handle itemHandle = event.getHandle();
    String itemName = _handleMap.get(itemHandle);
    OMMMsg msg = ommItemEvent.getMsg();

    byte type = msg.getMsgType();
    switch ( type )
    {
        case OMMMsg.MsgType.REFRESH_RESP:
        {
            if (itemName == null)
            {
                if (msg.getMsgType() == OMMMsg.MsgType.REFRESH_RESP)
                {
                    if (msg.has((OMMMsg.HAS_ATTRIB_INFO)))
                        itemName = msg.getAttribInfo().getName();

                    // match individual item name with item handle from item unsolicited refresh
                    handleMap.put(itemHandle, itemName);
                    nameMap.put(itemName, itemHandle);
                }
            }

            //skip more code here
        }
    }
}

```

### Example 150: Handling an Item Response Message Opened From the Name List

## 13.10 Service Groups

Refer to Section 5.9 for an introduction to the *Service Group* concept.

There are two types of Services: Concrete Services and Service Groups. A **Concrete Service** is a named grouping of data items. A **Service Group** is a named grouping of Concrete Services. A Concrete Service may reside in multiple Service Groups or may reside in no Service Group. The Infrastructure defines the number of potentially available Concrete Services, whereas the number of Service Groups is purely an API administrative decision.

Concrete Services and Service Groups have similar characteristics. For example, both offer similar naming, request semantics and event semantics. Concrete Services differ in that they ultimately are supplied by a single data vendor and have a single infrastructure-induced Quality of Service. Service Groups may be supplied from one or multiple data vendors and may have multiple Qualities of Service.

An OMM Consumer application may choose to interact with Service Groups, Concrete Services or both. Interacting with Service Groups allows the application to manage Concrete Services collectively. For example, an application may interact with a Service Group containing Concrete Services from a single vendor, across vendors or across several Qualities of Service.

The purpose of Service Group is to provide greater coverage and robustness to RFA clients. Consumer applications requesting an item from a service group may be served by any of the services included in it, whereas a request to a concrete service limits the offering to one service. A consumer may be connected to multiple providers and configure a service group that includes services from the multiple providers.

The Service Group feature requires a specific configuration as shown in the following section.

### 13.10.1 Configuration

The configuration support for Service Group is shown below:

```

consSession
  |
  connectionList = consConnection1, consConnection2
  serviceGroupList = SG1
consConnection1
  |
  serviceList = S1
consConnection2
  |
  serviceList = S2, S3
SG1
  |
  serviceList = S1, S2, S3
S1
  |
  feedName = RDF1
S2
  |
  feedName = RDF1
S3
  |
  feedName = RDF2

```

A concrete service can be configured as an alias (i.e., as a service with a different name). An application may turn-in use the alias when making requests to this service. Also note that the configuration will use the alias when adding the service to a service group, as shown in the configuration example above. Using a configuration with aliases allows the application to differentiate between two identical services offered by different providers. This concept is useful when the consumer is connected to multiple providers through multiple connections.

In the above example, service Refinitiv Data Feed Direct1 from consConnection1 has an alias S1, and the same service Refinitiv Data Feed Direct1 from the consConnection2 has an alias S2. The two services are added to the SG1 service group by using the alias names S1 and S2. The service Refinitiv Data Feed Direct 2, offered by the provider connected through consConnection2, is unique for this provider and does not need to be added to the service group as an alias. However, for the consistency, it is a good practice to use aliases, when configuring a service group.

## 13.10.2 Details

A user can make a request for an item to the service group by specifying the configured service group name instead of a service in Attribute Information, as shown in example below:

```
String itemName = "TRI.N";
reqMsg.setAttribInfo("SG1", itemName, RDMInstrument.NameType.RIC);
```

### Example 151: Requesting Data From Service Group

When RFA receives this request, it retrieves the SG1 service group and finds a service (assume S1) that is able to serve the request, i.e., the service status is UP and the Quality of Service is adequate to support the request.

- For details on Service Status and Quality of Service, refer to Sections 13.10.2.1 and 13.10.2.2.
- For definitions of QualityOfService and QualityOfServiceRequest, refer to Section 7.8.

RFA sends the request to the consConnection1 on service S1. If this request can't be fulfilled by the provider application, the consumer application will receive a status response message with appropriate Status. RFA will retrieve the next service belonging to the SG1 and will resend the request message to this service.

RFA will continue sending the request until the item is served, i.e., it receives a refresh response with the Status indicating an OPEN stream, or until it tried the last service in the service group. When the RFA determines whether or not the item can be served, it sends a response message to the application. Specifically, a refresh response will be sent if the item is served, or a status response if cannot be.

The benefit of requesting to a service group instead of to the concrete service, is apparent during a service outage. In this above scenario, if the consumer requested a service from SG1, and the serving service S1 changes state to DOWN, the RFA can recover the item on another available service that belongs to this service group.

### 13.10.2.1 Service Status

**Service Status** reflects the state of a service (i.e., Up, Down). The Session layer of RFA receives Service Status in events containing Directory domain messages. The application can register to receive the events by sending a Directory request. Assuming the application registered interest in Service Status, RFA sends events on a change in status of a Concrete Service. Additionally, RFA sends events for each Service Group that contains the Concrete Service.

A Service Group Service Status is a summary relating the status of all services that belong to a group. Note: If at least one service is UP, the service status group is UP.

### 13.10.2.2 Quality of Service

The Quality of Service feature combined with the Service Group feature provides a support for routing requests to services. The routing is determined by the services' Status and Quality of Service. The two attributes can change over time. The RFA has the capability to connect to the right service initially or when the service attributes change by utilizing the two features. Scenarios where Quality of Service is combined with the Service Group are described below.

#### 13.10.2.2.1 Routing for Quality of Service

When a consumer application request a range Quality of Service, the session layer of RFA will determine whether there is an available service that can provide data with this quality. If the user specifies a concrete service in the request, the session layer checks whether the specifically named service can abide to the request. If the request is done to the service group, the RFA will try all the services that are included in the specified service group. Thus, configuration with a service group give the user greater coverage.

#### 13.10.2.2.2 Change in Quality of Service

When the system changes the stream quality, a new image is sent, and the Status Code is set to indicate **QUALITY\_OF\_SERVICE\_CHANGE**.

When specifying interest, an application can specify the desired Quality of Service (e.g., Best Rate = **TickByTick**, Worst Rate = **SlowestRate**, Best Timeliness = **RealTime**, Worst Timeliness = **Delayed**). In addition to these attributes, the application can also specify whether the Stream will be **Static** or **Dynamic**. The difference between static and dynamic streams is described below.

Because of the following scenarios, there may be times when the connected Quality of Service is downgraded from the requested Quality of Service:

- The desired Quality of Service is not available on the network.
- The desired Quality of Service is available, but the user is not entitled to it.
- The initial Quality of Service provided in the original response can no longer be provided due to availability, entitlements or network congestion.

After the initial Concrete Service is chosen from a Service Group based on the requested Quality of Service, RFA may have to switch to another Concrete Service within the same Service Group. The Concrete Service that RFA switches to is dependent on the Stream property of the requested Quality of Service. If the Stream property was `STATIC_STREAM`, RFA only connects to any Concrete Service within the same Service Group that provides the exact Quality of Service that it was originally connected to, regardless of the requested Quality of Service.

If the Stream property was `DYNAMIC_STREAM`, RFA will connect to any Concrete Service within the same Service Group that can provide a Quality of Service that satisfies the requested Quality of Service. The following sections describe the triggers that cause RFA to switch.

#### 13.10.2.2.3 Recovery

When a service in use becomes unavailable, a new one must be found. In this case, the data stream can be switched over to another service currently available that can satisfy the Quality of Service request. RFA will attempt to reconnect to another Concrete Service within the same Service Group that will continue providing an item at a specific Quality of Service. For a static stream, the service's Quality of Service will exactly match the Quality of Service of the service to which the item was initially connected. For a dynamic stream, the Concrete Service's Quality of Service will match if it is a "best fit" with the originally requested Quality of Service. If the item is served from an inferior Quality of Service, then the stream will be changed to a "better" Quality of Service when the respective service becomes available.

When RFA receives a trigger signaling that it needs to perform recovery logic, the item will be placed in a recovering state. Periodically, RFA will check to see if a Concrete Service within the same Service Group is available and able to provide the Quality of Service needed. The Quality of Service needed is based on the Stream property of the requested Quality of Service. When a Concrete Service becomes available, the item is moved to the requested state, and item events are again sent to the user.

#### 13.10.2.2.4 Discovery

Discovery is the process in which RFA determines that another service in a service group can provide a better Quality of Service for a subscribed item, and automatically transfers the item to said service. Discovery is only applicable to dynamic streams. For static streams, the Quality of Service can't change, so nothing needs to be done.

If after receiving the initial image for an item, another Concrete Service within the same Service Group changes its state to 'OK' (and can provide a more favorable Quality of Service), RFA attempts to transfer the item to the new Concrete Service. If the new Concrete Service is in the same Service Group, RFA compares the Quality of Service of the new Concrete Service to the Quality of Service of the current Concrete Service. If the new Quality of Service is more favorable, RFA attempts to connect to the new Concrete Service. If it successfully connects, it closes the item on the old Concrete Service. If it is unsuccessful, RFA will stay with the original Concrete Service. If the item is served from an inferior Quality of Service, then the stream will be changed to a "better" Quality of Service when the respective service becomes available.

#### 13.10.2.2.5 Re-permission: Lock Change

After receiving a Refinitiv Data Access Control System lock change for an item on a Concrete Service, RFA will re-permission the item with the new lock. If a permission check fails on the Concrete Service that is currently being used, RFA performs the Recovery procedure to recover the item using another Concrete Service in the Service Group.

For a static stream, if the permission for the stream is still valid, RFA does nothing. If the permission is taken away from the stream, RFA closes the stream.

For a dynamic stream, RFA checks to see if the new lock grants permission for a better Concrete Service by performing the Discovery procedure. If permission is taken away for the current Concrete Service, RFA performs the Recovery procedure.

#### 13.10.2.2.6 Re-permission: Profile Change

Refinitiv Data Access Control System profile changes can potentially expand or shrink the set of Qualities of Service available to the user. After receiving a profile change from Refinitiv Data Access Control System, RFA will perform the Discovery procedure triggered by a re-permission, as opposed to the service's state changing to 'OK'.

#### 13.10.2.2.7 Stale Notification

The Stale Notification Procedure is followed when the infrastructure sends a status indicating that a watched item is stale. In this case, RFA may switch the item to another service, but only if it is available with the same Quality of Service for the item in question. If not, the item remains stale until the infrastructure automatically recovers the item.

#### 13.10.2.2.8 Closed Recover

When the infrastructure sends a closed recover event for an item, it indicates that the item is closed, but the application may attempt to reopen the item sometime later. RFA automatically tries to recover the item. In this case, the data stream is switched over only if another service is available with the same Quality of Service that can service the request. If not, RFA attempts to reopen the item from the same service that sent a closed recover.

## 13.11 Recovery

RFA supports Connection Recovery and Item Recovery. Refer to Sections 3.2.9 and 3.2.10 for an introduction to the **Recovery** concept.

When the RFA Consumer or RFA Provider connection goes down, the RFA Session layer attempts to reconnect without application intervention. This feature is known as Connection Recovery. Several configuration parameters define how often and how long RFA will attempt to recover connection. Refer to Section 13.11.1 for configuration details.

When a provider application becomes incapable of providing data for an open item stream, the data may be recovered using a process called Item Recovery. RFA may attempt to recover the item stream by resending the open request, or the data may be recovered by the provider, or the stream may be closed. The OMM Interface provides support for setting up the item stream recovery routine. Refer to Section 13.11.2 for details.

### 13.11.1 Configuration

The configuration for the Connection Recovery feature is outlined below:

#### consConnection, niProvConnection

```
|
  |
  |   connectionTimeout
  |   connectRetryInterval
  |   maxRetryCount
```

The RSSL and RSSL\_CPROV connection types define the `connectionTimeout`, `connectRetryInterval` and `maxRetryCount` elements.

- The `connectionTimeout` element defines the time in milliseconds to wait for a connection attempt to succeed. If the connection attempt times out, RFA performs recovery, if applicable.
- The `connectRetryInterval` element defines the time in milliseconds RFA will wait before attempting to recover a failed connection.
- The `maxRetryCount` element defines the maximum number of times RFA will attempt to recover failed connection. If this number is set to `-1`, the RFA tries to reconnect indefinitely. If this number is set to `0`, the RFA does not attempt to reconnect.

The configuration of a single connection may include multiple host names or IP addresses in its `serverList`. If the connection defines multiple hosts in the `serverList`, RFA attempts to reconnect to them in a round-robin fashion, and may (eventually) retry the first server in the list. This process will continue until a connection is established, or `maxRetryCount` is reached. If a connection is established, the count of connection attempts will reset to `0` (for the connection).

### 13.11.2 OMM Interfaces Support

The OMM package provides support for item stream recovery. The `OMMState` interface defines the health of an item stream. Refer to Section 10.3.6 for details.

`OMMState` contains the `OMMState.Stream` and `OMMState.Data` properties. Relevant combinations for item recovery are highlighted in the table below:

STREAM STATE	DATA STATE	DESCRIPTION
<code>OMMState.Stream.OPEN</code>	<code>OMMState.Data.OK</code>	Data on the stream is up to date.
<code>OMMState.Stream.OPEN</code>	<code>OMMState.Data.SUSPECT</code>	The stream is open, but the data is stale.
<code>OMMState.Stream.CLOSED_RECOVER</code>	<code>OMMState.Data.SUSPECT</code>	The stream is closed, but recovery will be attempted.
<code>OMMState.Stream.CLOSED</code>	<code>OMMState.Stream.SUSPECT</code>	The stream is closed, no recovery will be attempted.

**Table 92: Item Stream State**

RFA always communicates the most recent state of each item stream to a consumer application. For example, the initial state of the item is included in the initial refresh message, and any change to the state is communicated via status messages. If the data state changes to `SUSPECT`, the consumer application should take action that depends on the stream state and on the initial setting of the Login client.

Initially, RFA uses the `OMMAttribInfo` `Attrib` `Singleopen` and `AllowSuspectData` elements to establish item stream recovery routine. The attributes are included in the Login request.

The `SingleOpen` attribute communicates upstream (to RFA, which in turn communicates to the server, and so on) whether the consumer expects the system to recover data. If this flag is set to value `1`, then the system should drive the recovery. If this flag is set to value `0`, then the client application handles the recovery.

The `AllowSuspectData` attribute communicates upstream whether the consumer application accepts suspect data. If this flag is set to value `1`, then images and updates can be stale, i.e. the provider will send a status on an OPEN stream and SUSPECT data and will keep sending updates, even if the data is stale. If this flag is set to the value `0`, then the consumer application will not receive stale data. Instead, the provider will send status with CLOSED\_RECOVER stream.

### 13.11.3 Details

#### 13.11.3.1 Connection Recovery

The Connection Recovery procedure is determined by the configuration parameters described in Section 13.11.1.

When an RFA Consumer connection is recovered, the item streams that were open before the connection went down, will be recovered according to the recovery routine that the application set up. The Session Layer of RFA retains a watchlist of all open items. When a connection is restored, RFA will automatically re-register interest for all the items in the watchlist.

The application is notified of each connection attempt through the logger.

#### 13.11.3.2 Item Stream Recovery For Consumers

An active item stream may cease to be served for any of the following reasons:

- The connection goes down
- The provider capacity was exceeded
- The service becomes unavailable
- The item group becomes unavailable
- The Quality of Service of the service decreased below the quality requested by the consumer application

In all of the cases, RFA internally receives an event, which includes changes to stream state or service status. RFA processes these events and in-turn generates and delivers an appropriate event to the consumer application for each affected stream. Refer to Section 12.2.6.9 for details.

If a consumer requests an item from a service group, RFA attempts to recover the item using other services that belong to the service group.

If the stream can't be recovered through the service group, for each stream that received a status with a stream state CLOSED\_RECOVER, RFA applies the item recovery logic, based on the `SingleOpen` attribute. If `SingleOpen` is set to `1`, RFA provides recovery, otherwise the application should handle the CLOSED\_RECOVER status.

#### 13.11.3.3 Item Stream Recovery For Providers

A Provider application receives the `SingleOpen` and `AllowSuspectData` attributes in a Login request. If the consumer requested recovery (i.e., `SingleOpen` set to `1`), but the provider can't support it, the provider returns the attribute set to `0` in the Login response.

The `AllowSuspectData` attribute affects how the provider handles stale data. If this attribute is set to `1`, the provider will send status messages with a stream state OPEN and SUSPECT data for a stale stream, and will keep sending updates. Otherwise it will send a CLOSED\_RECOVER stream status.

## 13.12 Quality of Service

### 13.12.1 Overview

Quality of Service is an attribute associated with a Service that characterizes how current is the data provided by this Service. An RFA provider may present the available services and associated quality information to consumers in a Service Directory response. If the provider does not offer this information, the best quality of service is assumed by the receiving consumer. The consumer is aware of the quality of the services that are available to it.

A consumer application requesting an item can specify the range of Quality of Service it desires from a service, and can specify whether the stream quality can change. Refer to Section 7.8.3 for details about the `QualityOfServiceReq` definition. A provider application may provide the quality attribute in response messages that define the quality of service the consumer is receiving. Refer to Section 7.8.2 for details about the `QualityOfService` definition.

The Session Layer of RFA internally maintains the state of services and their Quality of Service details, which are received during the consumer application's initial phase (i.e., in Directory Response). When the consumer application requests an item with a specific level of Quality of Service, the Session Layer determines whether the request can be fulfilled, and if it can, how to route it.

## 13.12.2 OMM Interface

The OMM package provides the `OMMQos` and `OMMQosReq` interfaces. These interfaces are adapters to the `QualityOfService` and `QualityOfServiceReq` definitions respectively.

The OMM request type message optionally includes the `OMMQosReq` element. The `OMMMsg` interface also provides the `HAS_QOS_REQ` flag, which indicates presence of this element.

The OMM refresh response or update response type messages optionally include the `OMMQosReq` element. The `OMMMsg` interface also provides the `HAS_QOS` flag, which indicates presence of this element.

## 13.12.3 Details

### 13.12.3.1 Provider Applications

RFA uses the Directory domain to exchange information about services. A provider application sends a Directory Response message when the Service Directory is requested. If the request includes the `RDMSERVICE_FILTER_INFO` flag in the Attribute Information Attributes element, the provider may include the quality of service information in the response message. The quality of each service is expressed as array containing the of qualities the service supports.

The example below shows how the provider application encodes Quality of Service:

```
if ((attributeInformation.getFilter() & RDMSERVICE_FILTER_INFO) != 0)
{
    ...
    encoder.encodeElementEntryInit(RDMSERVICE_INFO_QOS, OMMTypes.ARRAY);
    encoder.encodeArrayInit(OMMTypes.QOS, 0);
    encoder.encodeArrayEntryInit();
    encoder.encodeQos(OMMQos.QOS.QOS_REALTIME_TICK_BY_TICK);
    ...
}
```

#### Example 152: Encoding Quality of Service In a Directory Response

The complete encoding of Service Directory messages can be found in the example applications, i.e., `session.ommprov..` Quality of Service information can be included in the refresh responses and update response messages. The refresh response gives the information about the services at the time the provider received a request. If the service characteristics change, the provider may send update response with the current information.

A Service Directory message may be routed to the consumer through infrastructure nodes with different capabilities. If any of the nodes is not capable of providing the quality of service the provider advertises, the quality that consumer receives is downgraded to take account of the node responsible for the degradation.

The provider may optionally include the `OMMQos` element in the response message to an item request.

### 13.12.3.2 Consumer Applications

A consumer application may include the `OMMQosReq` element when requesting an item. When a range is specified, the system will try to find the best Quality of Service available to the user. This can be based on entitlements and/or service availability. When using this type of "best available" request, the actual Quality of Service provided will be specified in a parameter contained within the image response. If this element is not specified, RFA uses the default value, which is a complete range of QualityOfService. The Session Layer of RFA determines whether the service the item was requested to is capable to provide with the quality specified by the `OMMQosReq` element. As stated in Section 13.12.3.1, RFA has the service quality information to determine the capability. If no service can provide the item with the requested quality, RFA returns a Status Response to the application, and no request is sent to the provider.

An application can specify a service as a concrete service or as a Service Group when requesting an item. Refer to Section 13.10.2.2 for the description of Service Group support of Quality of service.

### 13.12.3.3 Stream Property

The `QualityOfServiceReq` attribute defines the Stream Type. The Stream Type specifies whether the Quality of Service should change following the initial item image. The Stream Property could be one of the following:

- Static Stream Property: Implies that the Quality of Service will not change following the initial image.
- Dynamic Stream Property: Implies the Quality of Service may change following the initial item image.

The Dynamic Stream Property leverages Service Group, thus the consumer application typically uses the property when requesting an item to the Service Group. For details, refer to Section 13.10.2.2.

## 13.13 Item Group Management

### 13.13.1 Overview

Item groups can be used to efficiently update the state of many item streams through the use of a single group status message, instead of many individual item status messages. Each open data stream is assigned an item group.

### 13.13.2 OMM Interfaces Support

The OMM package defines the `OMMItemGroup` class. Refer to Section 10.9.1 for details. `OMMItemGroup` is a member of refresh response and status response messages.

### 13.13.3 Details

An item is assigned to an item group by a provider application. The group is conveyed to the consumer application in the initial refresh response or status response message. If the item migrates to a new group on a provider, the provider application sends a status message for this item with the new group. If the entire group of items migrates into another group (i.e., group merge), the provider sends a status message informing of group merge.

Item groups are defined on a per-service basis. It is possible to have two item groups that have matching values, but once the group's service ID is also considered, the values should be unique.

A consumer application should track the service ID-group pairings to ensure that only affected items are modified when group status messages are received. A provider can establish item group assignments on any basis that makes sense to the application's needs, but should ensure that each item group is unique 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 service being aggregated. This would allow the provider to mark all of the items from an upstream service that has become unavailable as being suspect, while all items from any other upstream services remain in their prior state.

A following diagram shows an example of utilizing item groups:

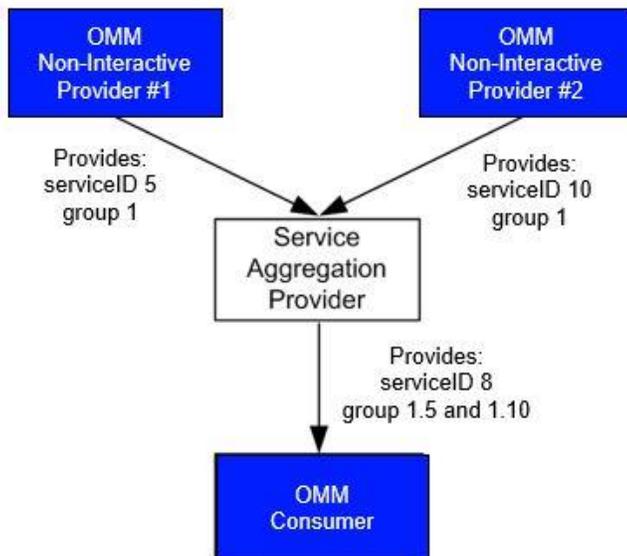


Figure 53: Item Group Example

This figure depicts two Non-Interactive Provider applications, each publishing item streams belonging to specific services and specific item groups. Both providers are communicating with an application that consumes data from all services, aggregates all data together into a single service, and allows distribution of this information to consumer applications. To ensure uniqueness to downstream components, the service aggregation provider appends additional identifiers to the group information it is receiving from the provider applications. In this example, the aggregation device has modified service ID **5**, group **1** into a group of **1.5** and service ID **10**, group **1** into a group of **1.10**. If Non-Interactive Provider #1's connection or service becomes unavailable, the aggregation device can send a single group status message to inform the consumer that all items belonging to group **1.5** are suspect. This would have no impact to any items belonging to group **1.10**.

### 13.13.3.1 Item Group Buffer Contents

The data contained in the group buffer should be treated as opaque by a consuming application, and may vary in length. The actual data contained in the group 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 group buffer. Providers that combine multiple data sources must ensure that the item groups in the resulting service are unique. This can be accomplished by appending an additional two byte value to each group that is on-passed to ensure uniqueness.

### 13.13.3.2 Group Status Message Information

Indications of state change and the merging of item groups occur via group status messages. A group status message is communicated to RFA via the Source Directory domain message model. For more specific information, refer to *RFA Java RDM Usage Guide*. A consumer application may receive the same information if it subscribes to Source Directory events. This is optional, since RFA processes the information and always sends a status event to the consumer application when a group merges or a group status changes.

## 13.14 Warm Standby

For an introduction to the *Warm Standby* concept, refer to Section 3.2.11.

The warm Standby feature provides a mechanism that enables OMM Consumers to establish standby connections in addition to the active connection. The standby server will not provide data to the consumer, but will have the data available. When the active connection fails, the standby server will start to service connected clients, immediately providing the data so the failover has a minimal impact on the consumer. An OMM Provider that supports Warm Standby is capable of serving in active and standby modes.

The application example [StarterProvider\\_warmStandby](#) illustrates the Warm Standby feature.

## 13.14.1 Configuration

The configuration for the Warm Standby feature is shown below:

### consConnection

```
|
  serverList: SBL1
  disableDownloadConnectionConfig
```

### StandbyLists

```
|
  SBL1
  |
  serverList
  startingActiveServer
```

The configuration above shows a configured standby list named SBL1. The list contains the **serverList** and **startingActiveServer** parameters. The serverList contains the active, and all standby servers the consumer will be connected to. The **startingActiveServer** parameter indicates a server to which the consumer will initially connect. If this parameter is not specified, RFA will select the first server from the serverList.

The Refinitiv Source Sink Library type connection may use a standby list as one of the servers in the connection serverList parameter. If the standby list is specified, the consumer will connect to the server configured in startingActiveServer parameter. If this is not configured, the consumer will connect to the first server from the standby list.

The Refinitiv Source Sink Library type connection may also configure the **disableDownloadConnectionConfig** parameter. The default for this parameter is false, which means the RFA consumer may be redirected (refer to Section 13.15 for details on the Connection Redirect feature). The Connection Redirection takes priority over the Warm Standby configuration. However, if the parameter is set for redirection, but the network does not supply the necessary redirection information, the Warm Standby configuration applies. If the **disableDownloadConnectionConfig** parameter is set to **true**, the redirection will not be relevant, and the Warm Standby configuration will apply.

## 13.14.2 OMM and RDM Interface Support

In the OMM package, the `AttribInfo.Attrib` element defines the `SupportsStandby` attribute. This attribute is set by the Provider to communicate to the OMM Consumer whether or not Warm Standby is supported. A value of **1** indicates the provider's support, and a value of **0** indicates otherwise.

The RDM defines a Generic type message in the Login domain. This message is sent automatically by RFA from consumers to providers informing the provider application which mode, active or standby, applies to the consumer. This message is sent initially to all standby servers, and is sent to the new active server during failover. Refer to Section **Error! Reference source not found..**

For further details, refer to the *RFA Java RDM Usage Guide*.

## 13.14.3 Details

### 13.14.3.1 OMM Consumer

To set up warm standby behavior, the consumer application must be configured as described in Section 13.14.1. The consumer application will connect to the active/standby group of servers, with RFA internally requesting the same items from both active and standby servers. However, the OMM Consumer application receives updates only on item streams opened to the primary/active server, and does not get updates, status, acknowledgement, and unsolicited refreshes messages from the standby/backup server(s). RFA does not provide standby service for private streams or for generic messages on data domain streams (i.e., these messages are not sent to standby servers).

If the active server fails, RFA switches to one of the standby servers. That standby server then becomes the new active server and continues sending data, without the application re-requesting the items. Any server that goes active after a failover should begin sending conflated updates with changed data for conflatable domains, and send unsolicited refresh messages followed by updates for any non-conflated domains. This process brings data streams back to their original pre-failover state.

When RFA switches to a new active server, the consumer application will be affected as follows:

- A log message, notifying of the switch to a new active server, will be written to a console. RFA also sends an `OMMConnectionEvent` event with `ConnectionStatus State UP` and `Status Code SERVER_SWITCHED` to the consumer application.
- It will receive a status message for each service that is not provided by the new provider, but was offered by the previously active provider.
- A status message with a stream state `ClosedRecover` for any item open on the failed server, but no longer open on the newly active server.

### 13.14.3.2 RFA Internal Processing on OMM Consumer

After establishing a connection to the standby server, RFA internally ensures that the standby server provides similar services, to the services offered by the active server by verifying the following criteria:

The Standby server's Login response must match:

- The `AttribInfo.Attrib`'s `ApplicationId`, `PositionId`, and `ProvidePermissionProfile` (If the user requested the data in Login request message, the values from all servers must match what was requested.)
- The `ProvidePermissionExpressions` (If requested by the user, values from all servers must match what was requested.)
- The `SingleOpen`, `AllowSuspectData`, `SupportBatchRequests`, `SupportOMMPost`, `SupportViewRequests` elements.

If any of the elements do not match, RFA disconnects the standby server.

The Standby server's Directory response must match all attributes except `Vendor` and `IsSource` of the `SERVICE_INFO_FILTER` for services common to both active and standby servers, otherwise, RFA disconnects the standby server.

The Standby server's Dictionary response must match the major version otherwise, RFA disconnects the standby server.

### 13.14.3.3 OMM Provider

An RFA providing application must notify the consumer application if the server supports the warm standby feature using the Login response. The consuming side of RFA notifies the providing application if the provider should operate in Standby mode or Active mode through a `ConsumerConnectionStatus` generic message on a LOGIN stream. Providing applications operating in Standby mode must send a refresh with an empty payload in response to an item request for data domains.

When a providing application in Standby mode receives a `ConsumerConnectionStatus` generic message on a Login stream indicating that it should operate in active mode, the providing application must do the following:

- Send a full update on the directory.
- If data is being conflated, send updates containing all changed data and then continue sending update messages as normal.
- If data was not conflated or conflatable, send an unsolicited refresh message to provide all data and bring the state back to OK. After this, the providing application should send update messages as normal.

## 13.15 Connection Redirect

For an introduction to "Load Balancing (Connection Redirect)," refer to Section 3.2.13. The Connection Redirect feature enables the load on RFA providers to be spread across multiple servers on the platform.

An RFA consumer's connection can receive a providers current utilization statistics (referred to as it's current load) from the platform during login processing.

The platform can provide a list of provider servers with a low load in the Login response message. Based on the information, the connection serving the consumer application can disconnect from the initial server, and connect to one provided in the list. In addition to the target server, the list may also include standby providers. Multiple redirections are possible, if the subsequent login responses contain server lists.

The process of redirection is transparent to the application.

## 13.15.1 Configuration

Configuration support for Connection Redirect feature is shown below:

consConnection

|

**disableDownloadConnectionConfig**

**maxNumRedirections**

Each Refinitiv Source Sink Library type connection has a **disableDownloadConnectionConfig** configuration parameter that enables or disables the redirection feature. The default for this parameter is false, which means the feature applies to Refinitiv Source Sink Library type connections by default.

If the **disableDownloadConnectionConfig** configuration parameter is set to false (i.e., the feature applies), an additional configuration parameter applies. The Refinitiv Source Sink Library type connection has a **maxNumRedirection** configuration parameter, which sets the limit on the number of redirections. The default for this parameter is 1. Acceptable values range from 0 to the maximum integer. If the value is 0, the RFA consumer stays connected to the initial server, and no redirection occurs.

## 13.15.2 Details

Any process of redirection is transparent to the user. The redirection happens in the connection layer. The following description is informational only, and not related to writing an application.

If a redirection occurs, the initial login response (from the first provider) is not forwarded by the RFA to the consumer application; the final login response processed is sent to the consumer. For details on the Login response payload, refer to the *RFA Java RDM Usage Guide*.

If an application is configured to use the redirection feature, the RFA consumer internally requests the list of servers by specifying **DownloadConnectionConfig** in the wire format of a Login request using the **AttribInfo.Attrib** element (Refer to Login domain in the *RFA Java RDM Usage Guide*). When a **ConnectionConfig** is included in Login response, RFA redirects to the first server on the list. The RFA consumer stays connected to the server (i.e., the redirection process is completed) in the following cases:

- When the server from which RFA receives configuration information is listed first in the downloaded connection list.
- When RFA detects that the maximum number of redirections is reached.
- When the server does not return **ConnectionConfig** in the payload, or sends a response without a payload.

## 13.16 Tunneling

For an introduction to **Tunneling**, refer to Section 4.7.

The use of tunneling provides Internet connectivity via HTTP and HTTPS. OMM Consumer applications can establish connections via HTTP and HTTPS tunneling. OMM Interactive Provider applications can accept incoming Refinitiv Source Sink Library connections tunneled via HTTP. This functionality is supported across all platforms.

## 13.16.1 Configuration

The configuration for Tunneling feature is shown below:

Each Refinitiv Source Sink Library type connection, i.e., a consumer connection, defines the following parameters supporting the Tunneling feature:

### consConnection

```
|
  tunnelingType
  tunnelingKeystoreType
  tunnelingKeystoreFile
  tunnelingKeystorePassword
  tunnelingSecurityProvider
  tunnelingKeyManagerAlgorithm
  tunnelingTrustManagerAlgorithm
  tunnelingHTTPproxy
  tunnelingHTTPproxyHostname
  tunnelingHTTPproxyPort
  tunnelingObjectName
```

The [tunnelingType](#) configuration parameter defines type of tunneling for this connection. The accepted values are: None (non-tunneling), http (HTTP type connection), and https (encrypted HTTP connection).

The [tunnelingHTTPproxy](#) configuration parameter enables connecting to an HTTP Proxy. If set to true, RFA Java uses an 'HTTP CONNECT' to tunnel through an HTTP Proxy. If [tunnelingHTTPproxy](#) is true, the config parameters [serverList](#) and [portNumber](#) are still used to specify the final destination of the Provider you want to connect to.

The [tunnelingHTTPproxyHostname](#) configuration parameter defines the hostname of the HTTP Proxy.

The [tunnelingHTTPproxyPort](#) configuration parameter defines the port number of the HTTP Proxy.

The [tunnelingObjectName](#) configuration parameter defines an object name for load balancing to the various ADSs that are part of a hosted solution.

The following parameters apply when the tunneling type is [https](#):

- The [tunnelingKeystoreType](#) configuration parameter defines the type of Java keystore. If empty, the property [keystore.type](#) in the JDK [java.security](#) file is used.
- The [tunnelingKeystoreFile](#) configuration parameter defines a filename of the Java keystore.
- The [tunnelingKeystorePassword](#) configuration parameter is the password that was used when creating the Java keystore file. Password may be empty.
- The [tunnelingSecurityProvider](#) configuration parameter specifies the Java Security Provider to use.
- The [tunnelingKeyManagerAlgorithm](#) configuration parameter specifies the key management algorithm used. The Key Manager determines which authentication credentials to send to the remote host. If empty, then the property [ssl.KeyManagerFactory.algorithm](#) in the JDK file [java.security](#) is used.
- The [tunnelingTrustManagerAlgorithm](#) configuration parameter specifies the trust management algorithm used. The Trust Manager determines the remote authentication (and thus the connection) should be trusted. If empty, then the property [ssl.TrustManagerFactory.algorithm](#) in JDK file [java.security](#) is used.

Each `RSSL_PROV` type connection, i.e., an interactive provider connection, defines the following parameters supporting the Tunneling feature:

#### provConnection

|

#### tunnelingType

The `tunnelingType` configuration parameter defines the types of tunneling this server can accept.

- If `tunnelingType` is `http`, the server can accept both normal non-tunneling connections and http connections.
- If `tunnelingType` is `None`, the server accepts only normal non-tunneling connections.

---

**NOTE:** A provider cannot directly accept HTTPS connections. HTTPS traffic encrypted by a Consumer must be decrypted by a 3<sup>rd</sup> party hardware or software device and forwarded to the provider as HTTP traffic.

---

## 13.16.2 Details

An application utilizing the tunneling feature needs to configure the parameters highlighted in Section 13.16.1.

An interactive provider application simply specifies that it accepts http traffic, by setting the `tunnelingType` configuration parameter to `http`.

A consumer application needs to configure additional parameters, in addition to `tunnelingType`. By setting the `tunnelingHTTPProxy` configuration parameter to true, the application will be connected via a proxy. The following configuration parameters specify the proxy host name and port: `tunnelingHTTPProxyHostname` and `tunnelingHTTPProxyPort`. A client connection that is leveraging a `connectionType` `http` or `https` may be connecting through proxy devices as it tunnels through the Internet.

If the consumer application utilizes connection configured for `https`, additional configuration parameters apply. The parameters specify security settings, such as: `tunnelingKeystoreType`, `tunnelingKeystoreFile`, `tunnelingKeystorePassword`, `tunnelingSecurityProvider`, `tunnelingKeyManagerAlgorithm`, `tunnelingTrustManagerAlgorithm`. RFA utilizes JDK `java.security` package. Some of the parameters (i.e., : `tunnelingKeystoreType`, `tunnelingSecurityProvider`, `tunnelingKeyManagerAlgorithm`, `tunnelingTrustManagerAlgorithm`) if not specified, are supported by the JDK `java.security` package.

## 13.16.3 Proxy Authentication

You can configure some proxy servers to authenticate client applications before they pass through the proxy to their destination. The list of supported HTTP/HTTPS proxy servers and associated authentication schemes (e.g., Negotiate/Kerberos, NTLM) are specified in the `README` file.

---

**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:

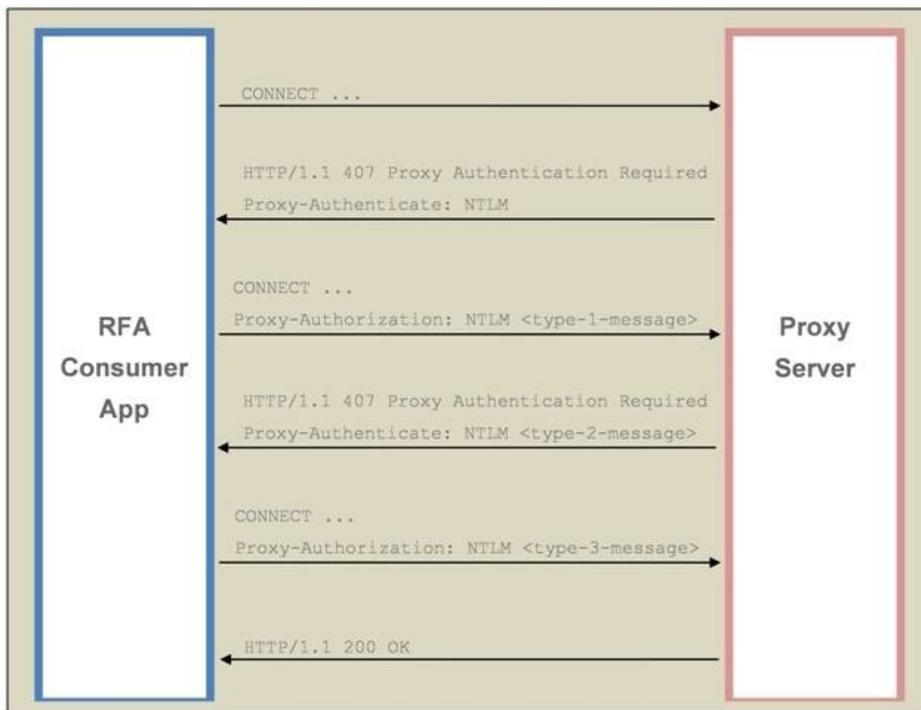
- Establish the type of credentials an application must provide to the proxy server.
- Define how the credentials required for authentication are encoded.
- Determine the “handshaking” process in which messages will be exchanged between the proxy and the application during the authentication process.

This section:

- Provides an overview of the proxy authentication process
- Details how to programmatically supply RFA Consumer applications the user credentials required to authenticate with a proxy
- Illustrates what the console output of an RFA Consumer application successfully authenticating with an HTTP/HTTPS proxy server looks like
- Provides tips for troubleshooting Proxy Authentication failures

### 13.16.3.1 An Overview of the Proxy Authentication Process

The proxy authentication process follows the pattern shown in the diagram below. The diagram illustrates an RFA Consumer application successfully authenticating with a proxy server configured to use Windows Authentication (i.e., NTLM):



**Figure 54: An RFA Consumer Application Authenticating with a Proxy Server Using NTLM**

In this diagram:

- Using HTTP or HTTPS protocol, the RFA Consumer application (or applet) attempts to connect to a provider (e.g., a ADS; which is 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.

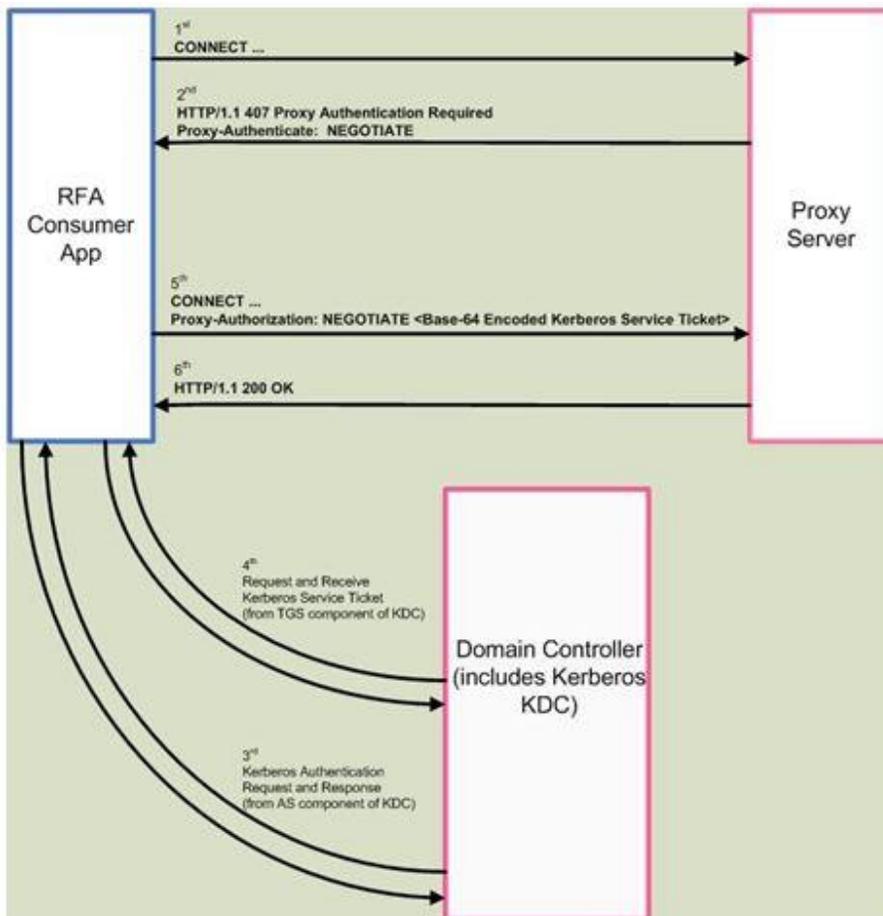
The initial response sent from the proxy server may also indicate that the connection between it and the consumer application will be closed.

- If RFA 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.
  - 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 that the application provides. 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 RFA with the HTTP “OK” error code# 200, indicating a successful authentication.
  - If the authentication scheme is NTLM (as illustrated in the diagram above), then it requires negotiation (i.e., multiple messages sent back and forth) to complete the authentication process, after the initial message, the proxy server again sends a message to the RFA containing HTTP error code #407, and (typically) additional handshaking details to be processed by the application. RFA would then use this information to send a message back to the proxy server to continue the authentication process until authentication ultimately succeeds (or fails). If authentication is ultimately successful, the proxy sends a response to RFA with HTTP “OK” error code# 200.

- If the authentication scheme is Negotiate/Kerberos, (as illustrated in Figure 55) then it requires additional handshaking with a Domain Controller to complete the authentication process. After the proxy server sends a message to RFA containing HTTP error code #407, RFA will do 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. RFA then uses this ticket to attempt to authenticate and if it is now successful, the proxy sends a response to RFA with HTTP “OK” error code #200.

If authentication fails, the proxy server sends a response containing an HTTP error code and text/HTML indicating that authentication failed.

If an RFA Consumer’s connection is configured (via the RFA Configuration Editor) to connect to a provider via a proxy server (using HTTP or HTTPS tunneling), and the proxy server requires authentication, RFA will automatically participate in the authentication process. The application must supply RFA-valid credentials as described in the next section. Specifically, RFA will automatically parse the list of supported authentication schemes (sent by the proxy), select the most appropriate scheme, re-connect to the proxy (if necessary), and exchange the messages required by the selected authentication scheme.



**Figure 55: An RFA Consumer Application Authenticating with a Proxy Server using Negotiate/Kerberos**

**NOTE:** When an RFA Java Consumer application connects to an RFA Java Provider via HTTP or HTTPS tunneling, RFA establishes a single connection between the consumer and provider. When an RFA Java Consumer application connects to a non-Java Provider (e.g., ADS) via HTTP or HTTPS tunneling, RFA establishes two connections to the Provider. After successfully connecting and logging in to a non-Java provider, the console output from RFA will display the details of this second connection attempt. Because the second connection must also authenticate with the proxy, the console output will show the authentication process repeating a 2nd time.

From the perspective of the RFA Consumer application, the only additional “work” required to support proxy authentication is to programmatically supply RFA the credentials required for authentication. Section 13.16.3.2 describes this code in detail.

### 13.16.3.2 Supplying RFA Credentials for Proxy Authentication

This section describes the credentials required for proxy authentication and explains how to supply them to RFA.

When RFA connects to a proxy server that requires authentication, the proxy server sends a response to RFA containing HTTP error code# 407, indicating authentication is required and includes a list of authentication schemes enabled on the proxy server.

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
```

#### Example 153: Sample 407 Proxy Response Listing the Authentication Schemes Enabled on the Proxy

RFA supports Negotiate (Kerberos), Kerberos, NTLM and Basic authentication schemes (highlighted in the output above). These authentication schemes are listed from ‘most secure’ (Negotiate/Kerberos) to ‘least secure’(Basic). RFA would first attempt to use the first provided authentication scheme (i.e., Negotiate) and if that fails it attempts to use the next authentication scheme in the order provided. So in the above sample the order of authentication schemes attempted 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 as described below.

**NOTE:** You must determine how your Consumer application or applet obtains credentials (e.g., usernames and passwords) from end users.

#### 13.16.3.2.1 Providing Credentials to RFA

The following procedure describes how to provide the required authentication credentials to RFA. The following procedure illustrates how to modify the **StarterConsumer** example. For further details on this procedure, refer to Section 13.16.3.2.2.

- Open **StarterConsumer.java** located in **Examples/com/reuters/rfa/example/omm/cons**.
- Add the following import statements:

```
import com.reuters.rfa.config.ConfigDb;
import com.reuters.rfa.config.CredentialConfigKeys;
```

- Directly below the call `Context.initialize()`, insert the following code. For every invocation of the `configDb.addVariable()` method, replace the string literals (i.e., `domainname`) with the details / credentials obtained by your application:

```
// set the proxy credentials, which will only be used if the proxy requires authentication
final ConfigDb configDb = new ConfigDb();
final String connectionPath = "myNamespace.Connections.proxyConsumerConnection"; // the fully-qualified
name of the connection used to run this application / applet

//for all authentication schemes:
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyUsername,
    "Firstname.Lastname"); // the last parameter is the username (e.g. Robert.Smith)
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyPasswd, "MyPassword"); //
the last parameter is the password

//for Negotiate, Kerberos, or NTLM:
```

```

configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyDomain, "domainname"); //
the last parameter is the name of the windows domain(e.g. amers)
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyLocalHostname,
"workstationname.domainname.example.com"); // the last parameter is the local hostname of the
workstation (e.g. workstationname.amers.example.com)

//for Negotiate, Kerberos:
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyKRB5configFile,
"C:\\WINDOWS\\krb5.ini"); // the last parameter is the Kerberos5 configuration file

Context.initialize(configDb);

```

### 13.16.3.2.2 More Details on Providing Credentials

The following section describes in further detail each of the steps in Section 13.16.3.2.1.

First, an instance of `ConfigDb` (representing the RFA configuration database) is created:

```
final ConfigDb configDb = new ConfigDb();
```

The next line specifies the connection (as found in the RFA Java Configuration Editor) used to run the application (or applet). You must specify the full path to the connection (the namespace + `Connections` + connection name). Replace `myNamespace` and `proxyConsumerConnection` below with the namespace and connection name (respectively) used by your application:

```
final String connectionPath = "myNamespace.Connections.proxyConsumerConnection";
```

The next line specifies the username. Presuming a login name `amers\Robert.Smith`, replace `Firstname.Lastname` with `Robert.Smith`:

```
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyUsername, "Firstname.Lastname");
```

The next line specifies the password associated with the login. In the sample code below, replace `MyPassword` with the actual password:

```
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyPasswd, "MyPassword");
```

The next line is required only if you use Negotiate/Kerberos, or NTLM. It specifies the name of the Windows domain to which the user belongs. Consider the following sample Windows login name: `amers\Robert.Smith`. In this sample, the backslash separates the Windows domain name `amers` from the username `Robert.Smith`. Presuming a login name of `amers\Robert.Smith`, replace `domainname` in the sample code below with `amers`:

```
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyDomain, "domainname");
```

The next line is required only if you use NTLM. It specifies the hostname of the workstation from which the client logs in. Thus, replace `workstationname.domainname.example.com` in the sample code below with the actual local hostname on which your application (or applet) runs:

```
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyLocalHostname,
"workstationname.amers.example.com");
```

The next line is required only if you use Negotiate/Kerberos. It specifies the Kerberos5 configuration file needed to communicate with the Domain Controller (or KDC) to obtain the Kerberos service ticket needed to complete the Kerberos authentication.

```
configDb.addVariable(connectionPath, CredentialConfigKeys.tunnelingHTTPproxyKRB5configFile,
"C:\\WINDOWS\\krb5.ini");
```

A custom sample example of a KRB5 config file is the following:

```
[libdefaults]
  default_realm = DOMAINNAME.COM
  default_tkt_enctypes = aes128-cts rc4-hmac des3-cbc-sha1 des-cbc-md5 des-cbc-crc
  default_tgs_enctypes = aes128-cts rc4-hmac des3-cbc-sha1 des-cbc-md5 des-cbc-crc
  permitted_enctypes = aes128-cts rc4-hmac des3-cbc-sha1 des-cbc-md5 des-cbc-crc

[realms]
  DOMAINNAME.COM = {
    kdc = myDomainController
    default_domain = DOMAINNAME.COM
  }

[domain_realm]
  .DOMAINNAME.COM = DOMAINNAME.COM
```

In the above Kerberos config file, the domain controller is “myDomainController” and the user domain is “DOMAINNAME.COM”.

Finally, the last line adds the `configDb` instance which contains the user credentials to RFA’s `Context`:

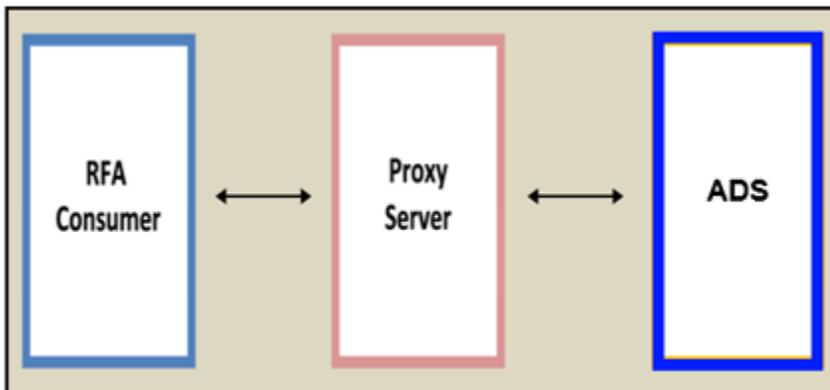
```
context.initialize(configDb);
```

After this code is in place, RFA automatically uses the above credentials for proxy authentication.

**NOTE:** If a proxy server does not require authentication, credentials are not used for any other purpose. Supplying them to RFA has no affect on the execution of your application. Additionally, if your application provides RFA details required for NTLM authentication, (i.e., the user’s domain, and application’s hostname) but RFA authenticates with the proxy server using another scheme (e.g. Basic), the additional details specified for NTLM authentication will not be utilized.

### 13.16.3.3 Console Output from a successful Proxy Authentication attempt

This section illustrates the console output of a Consumer application which successfully authenticates with an HTTP proxy server, ultimately connecting to a provider (and requesting an item). The following diagram describes the scenario:



**Figure 56: Proxy Authentication Architecture**

The following RFA Configuration Editor screenshot shows the Consumer configuration used in this scenario:

Key	Value
connectionType	RSSL
logFileName	console
portNumber	14002
serverList	oaklsun140
tunnelingHTTPproxy	true
tunnelingHTTPproxyHostName	oaklpc104
tunnelingHTTPproxyPort	8080
tunnelingType	http

**Figure 57: RFA Configuration Editor**

In the Consumer configuration above:

- **portNumber**: specifies the port number on which the ADS is listening for HTTP connections.
- **serverList**: specifies the hostname where the ADS is running.
- **tunnelingHTTPProxy** is set to **true**, indicating that the Consumer should connect to the Provider via a proxy server.
- **tunnelingHTTPProxyHostName** specifies the hostname of the machine on which the proxy server runs.
- **tunnelingHTTPProxyPort** specifies the port on which the proxy server listens for (client) HTTP connections.
- **tunnelingType** is set to **http**, indicating that the Consumer should utilize the HTTP protocol.

The following console output from the **StarterConsumer** example (modified to supply valid credentials) illustrates successful NTLM authentication with a proxy server. It also shows the consumer requesting an item from the provider, and the provider sending a refresh message containing data. For brevity, some of the proxy-sent HTML error messages have been shortened and some empty lines removed.

```
*****
*           Begin RFA Java StarterConsumer Program           *
*****
RFA Version: 7.2.1.L1.all
field dictionary read from RDMFieldDictionary file
enum dictionary read from enumtype.def file
LoginClient: Sending login request...
LoginClient.processEvent: Received Login Response...
LoginClient: Received Login Response - MsgType.REFRESH_RESP
```

A

MESSAGE

```
Msg Type: MsgType.REFRESH_RESP
Msg Model Type: LOGIN
Indication Flags: REFRESH_COMPLETE
Hint Flags: HAS_ATTRIB_INFO | HAS_ITEM_GROUP | HAS_RESP_TYPE_NUM | HAS_STATE
State: OPEN, SUSPECT, NONE, "All connections pending"
Group: 0000
RespTypeNum: 0 (RespType.SOLICITED)
AttribInfo
    Name: firstname.lastname
    NameType: 1 (USER_NAME)
    Attrib
        ELEMENT_LIST
            ELEMENT_ENTRY ApplicationId: 256
            ELEMENT_ENTRY Position: 10.10.10.59/OAKXAGOLD1
            ELEMENT_ENTRY AllowSuspectData: 1
            ELEMENT_ENTRY SingleOpen: 1
            ELEMENT_ENTRY SupportBatchRequests: 1
            ELEMENT_ENTRY SupportOptimizedPauseResume: 1
```

```

ELEMENT_ENTRY SupportPauseResume: 1
ELEMENT_ENTRY SupportViewRequests: 1
ELEMENT_ENTRY SupportOMMPost: 1

```

Payload: None

**B**

```

CONNECT oaklsun140:14002 HTTP/1.1
User-Agent: RFA/Java
Proxy-Connection: Keep-Alive
Content-Length: 0
Host: oaklsun140:14002
Pragma: no-cache

```

-- begin response -

**C**

```

HTTP/1.1 407 Proxy Authentication Required
Server: FreeProxy/4.50
Date: wed, 25 Apr 2012 16:12:13 GMT
Content-Type: text/html
Transfer-Encoding: Chunked
Proxy-Authenticate: NTLM
Proxy-Authenticate: Basic realm="somerealm"
Proxy-Connection: Close

```

7E5

<html>

<head>

<title>Error reported by FreeProxy</title>

</head>

<body>

<p style="margin-top: 6px; margin-bottom: 0px"><font face="Verdana" size="2">

Access denied owing to authentication failure.</font></p>

<p style="margin-top: 6px; margin-bottom: 0"><font face="Verdana" size="2">Retry

the action, using a valid userid and password.</font></p>

</body>

</html>

-- end response -

**D**

```

*** The proxy requested a close (during authentication). Reconnecting. ***

```

**E**

```

CONNECT oaklsun140:14002 HTTP/1.1
User-Agent: RFA/Java
Proxy-Connection: Keep-Alive
Content-Length: 0
Host: oaklsun140:14002
Pragma: no-cache
Proxy-Authorization: NTLM T1RMTVNTUAABAAAFAFEJKALJDKALJFDALKJDALKJFDALKJFDALKJFDALDAKLJFAALKJFALKJAFIAUWA=

```

-- begin response -

F

```

HTTP/1.1 407 Proxy Authentication Required
Server: FreeProxy/4.50
Date: wed, 25 Apr 2012 16:12:15 GMT
Content-Type: text/html
Transfer-Encoding: Chunked
Proxy-Authenticate: NTLM
TlRMTVNTUAACAAAACgAQKMNFDJHDHFIFKFFJFKIFLKDJSYFDJJJEWJSLKSLKQDCADZFSFDGVDGGBDGDGLJAZJMCHNCBWRDSJFODYDKT
UIXTGXFVKGKHLGAG0AZQAUAHIAZQB1AHQAZQByAHMALgBjAG8AbQAAAAAA
Proxy-Connection: Keep-Alive

```

```

7E5
<html>

<head>
<title>Error reported by FreeProxy</title>
</head>

<body>

<p style="margin-top: 6px; margin-bottom: 0px"><font face="Verdana" size="2">
Access denied owing to authentication failure.</font></p>
<p style="margin-top: 6px; margin-bottom: 0"><font face="Verdana" size="2">Retry
the action, using a valid userid and password.</font></p>

</body>

</html>
-- end response -

```

G

```

CONNECT oaklsun140:14002 HTTP/1.1
User-Agent: RFA/Java
Proxy-Connection: Keep-Alive
Content-Length: 0
Host: oaklsun140:14002
Pragma: no-cache
Proxy-Authorization: NTLM
TlRMTVNTUAADAAAAGAAAFADFDAaAEAAAAAAAAAB8AQAAANQIIKOZAQMSIJSwdKXNBLsBAQAAAAAANBAAI50BVMNBXTWfU00QUPOEQA
LKJFDALKDFJALFJAMZZNKLFACLKJFAFAJ;LKAcwAuAGMabwZZZZZZZBtAAMAAZZIIIIIIWmWYvNBXVwGFDJFJFGDNDZUUUUUYANZIj
NBXZRWTsWGBXGOOZKZIZIIZRYQXIHIAAAQXAA=

```

```

-- begin response --
-- end response --
-- begin response --
0
-- end response --
Ignoring a response from the proxy that did not contain a response code (1/10)
-- begin response --

```

H

```

HTTP/1.1 200 OK
Server: FreeProxy/4.50
Date: wed, 25 Apr 2012 16:12:17 GMT
Content-Type: application/octet-stream

```

```

-- end response --
Connection established to oaklpc104:8080

```

```
LoginClient.processEvent: Received Login Response...
```

```
LoginClient: Received Login STATUS OK Response
```

I

MESSAGE

```
Msg Type: MsgType.STATUS_RESP
```

```
Msg Model Type: LOGIN
```

```
Indication Flags:
```

```
Hint Flags: HAS_ATTRIB_INFO | HAS_STATE
```

```
State: OPEN, OK, NONE, "Login accepted by host oaklsun140."
```

```
AttribInfo
```

```
  Name: firstname.lastname
```

```
  NameType: 1 (USER_NAME)
```

```
  Attrib
```

```
    ELEMENT_LIST
```

```
      ELEMENT_ENTRY ApplicationId: 256
```

```
      ELEMENT_ENTRY ApplicationName: ADS
```

```
      ELEMENT_ENTRY Position: 10.10.10.59/OAKXAGOLD1
```

```
      ELEMENT_ENTRY ProvidePermissionExpressions: 1
```

```
      ELEMENT_ENTRY ProvidePermissionProfile: 0
```

```
      ELEMENT_ENTRY AllowSuspectData: 1
```

```
      ELEMENT_ENTRY SingleOpen: 1
```

```
      ELEMENT_ENTRY SupportBatchRequests: 1
```

```
      ELEMENT_ENTRY SupportOptimizedPauseResume: 1
```

```
      ELEMENT_ENTRY SupportPauseResume: 1
```

```
      ELEMENT_ENTRY SupportViewRequests: 1
```

```
      ELEMENT_ENTRY SupportOMMPost: 1
```

```
Payload: None
```

```
Consumer Login successful...
```

J

```
ItemManager.sendRequest: Sending item request...
```

```
ItemManager: Subscribing to TRI.N
```

```
ItemManager.processEvent: Received Item Event...
```

K

MESSAGE

```
Msg Type: MsgType.STATUS_RESP
```

```
Msg Model Type: MARKET_PRICE
```

```
Indication Flags:
```

```
Hint Flags: HAS_ATTRIB_INFO | HAS_STATE
```

```
State: OPEN, SUSPECT, NONE, "waiting for service IDN_RDF UP. Item recovery in progress..."
```

```
AttribInfo
```

```
  ServiceName: IDN_RDF
```

```
  Name: TRI.N
```

```
  NameType: 1 (RIC)
```

```
Payload: None
```

L

```
CONNECT 132.88.252.142:14002 HTTP/1.1
```

```
User-Agent: RFA/Java
```

```
Proxy-Connection: Keep-Alive
```

```
Content-Length: 0
```

```
Host: 132.88.252.142:14002
```

```
Pragma: no-cache
```

```

-- begin response --
HTTP/1.1 407 Proxy Authentication Required
Server: FreeProxy/4.50
Date: Wed, 25 Apr 2012 16:12:19 GMT
Content-Type: text/html
Transfer-Encoding: Chunked
Proxy-Authenticate: NTLM
Proxy-Authenticate: Basic realm="somerealm"
Proxy-Connection: Close

-- end response --
*** The proxy requested a close (during authentication). Reconnecting. ***

CONNECT 132.88.252.142:14002 HTTP/1.1
User-Agent: RFA/Java
Proxy-Connection: Keep-Alive
Content-Length: 0
Host: 132.88.252.142:14002
Pragma: no-cache
Proxy-Authorization: NTLM TlRMTVNTUAABAAAFAFEJKALJDKALJFDALKJFDALKJFDALKJFDALKJFDALDAKLJFAALKJFALKJAFIAUWA=

-- begin response --
HTTP/1.1 407 Proxy Authentication Required
Server: FreeProxy/4.50
Date: Wed, 25 Apr 2012 16:12:20 GMT
Content-Type: text/html
Transfer-Encoding: Chunked
Proxy-Authenticate: NTLM
    TlRMTVNTUAACAAAACgAQKMNFDJHDHFIFKFFJFKIFLKDJSYFDJJJEWJSLKSLKQDCADZFSFDGVDGGBDGDGDLJAZJMCHNCBWRDSJFODYDKT
    UIXTGXFVKGKHGLGAG0AZQAUAHIAZQB1AHQAZQBYAHMALgBjAG8AbQAAAAAA
Proxy-Connection: Keep-Alive

7E5
<html>

<head>
<title>Error reported by FreeProxy</title>
</head>

<body>

<p style="margin-top: 6px; margin-bottom: 0px"><font face="Verdana" size="2">
Access denied owing to authentication failure.</font></p>
<p style="margin-top: 6px; margin-bottom: 0"><font face="Verdana" size="2">Retry
the action, using a valid userid and password.</font></p>

</body>

</html>

-- end response --
CONNECT 132.88.252.142:14002 HTTP/1.1
User-Agent: RFA/Java
Proxy-Connection: Keep-Alive
Content-Length: 0
Host: 132.88.252.142:14002
Pragma: no-cache
Proxy-Authorization: NTLM
    TlRMTVNTUAADAAAAGAAAFADAFDAaEAAAAAAAAAB8AQANQIIKOZAQQMSIJSwdKXNBLsBAQAAAAAAAAANBAIs0BVMNBXTWfU00QUPOEQA
    LKJFDALKDFJALFJAMZZNKLFACLKJFAFAJ;LKAcwAuAGMabwZZZZZZZbtAAMAAZZIIIIIIWMWYVNBXVWGFDFJFGDNDZUUUUUYANZIJ
    NBXZRWTswGBXGOOZKIZIZRYQXIHIIAAQXAA=

-- begin response --

```

```

-- end response --
-- begin response --
0

-- end response --
Ignoring a response from the proxy that did not contain a response code (1/10)
-- begin response --
HTTP/1.1 200 OK
Server: FreeProxy/4.50
Date: wed, 25 Apr 2012 16:12:23 GMT
Content-Type: application/octet-stream

-- end response --
Connection established to oaklpc104:8080
ItemManager.processEvent: Received Item Event...

```

**M**

MESSAGE

```

Msg Type: MsgType.REFRESH_RESP
Msg Model Type: MARKET_PRICE
Indication Flags: REFRESH_COMPLETE | CLEAR_CACHE
Hint Flags: HAS_ATTRIB_INFO | HAS_ITEM_GROUP | HAS_PERMISSION_DATA | HAS_QOS | HAS_RESP_TYPE_NUM |
HAS_STATE
State: OPEN, OK, NONE, "All is well"
Qos: (RT, TbT)
Group: 00010acb00c671b0
PermissionData: 030acb6562c0 ( 0x03,0x0a,0xcb,0x65,0x62,0xc0 )
RespTypeNum: 0 (RespType.SOLICITED)
AttribInfo
  ServiceName: _RDF
  ServiceId: 2763
  Name: TRI.N
Payload: 1198 bytes
  FIELD_LIST
    FIELD_ENTRY 1/PROD_PERM: 6562
    FIELD_ENTRY 2/RDNDISPLAY: 64
    FIELD_ENTRY 3/DSPLY_NAME: REFINITIV
    FIELD_ENTRY 4/RDN_EXCHID: NYS (2)
    FIELD_ENTRY 6/TRDPRC_1: 29.1900
    FIELD_ENTRY 7/TRDPRC_2: 29.1800
    FIELD_ENTRY 8/TRDPRC_3: 29.1700
    FIELD_ENTRY 9/TRDPRC_4: 29.1600
    FIELD_ENTRY 10/TRDPRC_5: 29.1600
    FIELD_ENTRY 11/NETCHNG_1: 0.2500
    FIELD_ENTRY 12/HIGH_1: 29.2900
    FIELD_ENTRY 13/LOW_1: 28.9800

```

**Figure 58: Consumer Console Output Indicating a Successful Proxy Authentication attempt**

In the console output above:

- The Login Refresh Response from RFA, labeled (A), indicates that the login process has started.
- The highlighted (HTTP) CONNECT message, labeled (B), indicates RFA is attempting to connect to the proxy server.
- The highlighted response from the proxy server, labeled (C), contains an HTTP 407 error code (indicating authentication is required) and lists the authentication schemes enabled on the proxy. The message also indicates that the connection between RFA and the proxy will be closed.

The response is followed by an HTML-formatted error message (for display in web browsers, but ignored by RFA).

- The next highlighted output, labeled (D), which reads **\*\*\* The proxy requested a close (during authentication). Reconnecting. \*\*\*** indicates that RFA is automatically reconnecting to the proxy server to continue the authentication process.
- The next highlighted HTTP CONNECT message (sent from RFA to the proxy server), labeled (E), contains an NTLM type-1 message informing the proxy that RFA wants to use the NTLM authentication scheme.
- The highlighted response from the proxy server, labeled (F), contains an HTTP 407 error code with an NTLM type-2 message issuing a “challenge”.
- The response is followed by some HTML text (for display in a web browser), which is ignored by RFA.
- The next highlighted HTTP CONNECT message (sent from RFA to the proxy server), labeled (G), contains an NTLM type-3 message.

The “response” output following the CONNECT message is actually from the proxy server and was sent immediately after the NTLM type-2 message from Step  (i.e., it is not a response to the NTLM type-3 message). Because the HTML text is not relevant to the authentication process, RFA ignores the text.

- The next highlighted HTTP response from the proxy server, labeled (H), contains an **HTTP/1.1 200 OK** error code which indicates that the authentication was successful.
- After proxy authentication succeeds, RFA is able to connect through the proxy to its ultimate destination, the ADS. RFA then sends to the application a Login Status response message, labeled ( I ) with a state of OPEN / OK, indicating that the ADS accepted the login attempt.
- RFA then sends an item request for **TRI.N** to the ADS, labeled (J).
- RFA immediately delivers to the application a Status Response, labeled (K), with a state of OPEN / SUPECT indicating that it is waiting for the ADS to reply with the requested item **TRI.N**.
- As noted earlier, when an RFA Java Consumer connects to a non-Java RFA Provider via HTTP or HTTPS tunneling, it actually establishes two connections between the Consumer and the Provider. The (HTTP) CONECT message, labeled (L), shows the authentication process once-again occurring (and ultimately succeeding) for the second connection.
- Finally, RFA sends the application a Refresh Response (that was received from the ADS), labeled (M), with a state of OPEN / OK, containing market data for the requested item **TRI.N**.

### 13.16.3.4 Negotiate/Kerberos

As we see in Figure 78, the following sequence occurs when an RFA Java Consumer attempts to do Negotiate/Kerberos authentication:

1. The Consumer sends a HTTP CONNECT message to the Proxy Server attempting to connect to it.
2. The Proxy Server replies with an HTTP error code #407, which indicates that authentication is required. This message also lists the authentication schemes enabled on the proxy. In this case it "Negotiate".
3. The Consumer starts the first handshake with the Domain Controller (KDC) to request from the Kerberos Authentication Server (part of the KDC) a ticket to the Ticket Granting Server (another part of the KDC). The Authentication Server looks up the client in its database, then generates an encrypted session key for use between the client and the Ticket Granting Server.
4. The Consumer uses the encrypted session key to create an authenticator to send to the Ticket Granting Server (in KDC), which then decrypts the authenticator and sends the Kerberos service ticket needed for the "HTTP" service on the Proxy Server. This Kerberos ticket can now be used in authenticating with the Proxy Server.
5. The Consumer sends a new HTTP CONNECT message to the Proxy Server, but this time it is using an encrypted Kerberos service ticket just received from the Domain Controller (KDC).
6. The Proxy Server sends an HTTP message containing [HTTP/1.1 200 OK](#) error code which indicates that the Negotiate/Kerberos authentication was successful.

On Windows platforms the following registry key should be added:

- For Windows XP and Windows 2000, the registry key and value should be:
  - HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos
  - Value Name: allowtgsessionkey
  - Value Type: REG\_DWORD
  - Value: 0x01
- For Windows 2003 and Windows Vista, the registry key and value should be:
  - HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters
  - Value Name: allowtgsessionkey
  - Value Type: REG\_DWORD
  - Value: 0x01

### 13.16.3.5 Failed Proxy Authentication / Connection Attempts

Regardless of whether the proxy server requires authentication, RFA behaves the same way. If an RFA consumer cannot connect to a Provider due to a proxy authentication failure (or general connectivity issues), RFA sends the consumer application a Login Status Response message with a OPEN / SUSPECT state and any error text explaining the failure.

RFA's recovery logic also behaves the same, regardless of whether a proxy requires authentication.

If a failure requires that RFA reconnect to a provider, RFA automatically authenticates with the proxy server, if required.

### 13.16.3.6 Tips for Troubleshooting Proxy Authentication

When a proxy server fails to authenticate credentials, it typically closes the connection. In this case, RFA automatically reconnects to the proxy and retries the authentication process until either:

- The (configured) RFA connection timeout is reached, or
- An "unrecoverable" error occurs.

In both cases, RFA will send the application a login status message.

During the authentication process, some proxy servers send additional responses containing HTML or plain text to be displayed to users. RFA will ignore up to ten of these responses before considering the accumulated responses (containing non-parseable HTTP response codes) to be an unrecoverable error.

In this case, the error text in the login response read: [Unable to parse HTTP response code. \(Authentication errors may have occurred.\)](#). If you observe this error, ensure your application provides valid credentials applicable to the authentication scheme in use.

The following additional tips may be helpful when troubleshooting proxy authentication:

- If your consumer connects to an RFA Java Provider, be sure the RFA Java Provider has enabled HTTP tunneling.

The Provider's `tunnelingType` configuration parameter enables HTTP connections.

A provider cannot directly accept HTTPS connections. HTTPS traffic encrypted by a Consumer must be decrypted by a 3<sup>rd</sup> party hardware or software device and forwarded to the provider as regular, unencrypted HTTP traffic.

- To rule out proxy-related connectivity / authentication issues, if possible, confirm that a Consumer and Provider can successfully establish a direct HTTP connection to each other, with no intermediate proxy server.
- When the server (e.g. a Provider) listens on a port not typically utilized for HTTP/HTTPS traffic (e.g. 14002), some HTTP/HTTPS proxy servers (by default) prevent clients (e.g. an RFA Consumer) from establishing connections to servers (e.g. a Provider).

In this case, you may be tempted to work around the proxy configuration issue by configuring the Provider to listen for connections on a port typically used for HTTPS connectivity (e.g. 443). However, such a workaround might fail because by default, some operating systems will prevent a Provider from binding to this specially "privileged" port.

- Providers support HTTP, but not HTTPS. If you use a 3<sup>rd</sup>-party hardware or software device that decrypts HTTPS traffic (and in-turn forwards the non-encrypted HTTP traffic to the Provider), the consumer's `serverList` configuration parameter should specify the hostname or IP address of the hardware or software device, with the consumer's `portNumber` parameter set to the port on which the device listens (e.g. 443).
- If the proxy server uses an authentication scheme (e.g. NTLM) and RFA cannot authenticate, consider testing the proxy authentication scheme by:
  - Requesting a document (such as a web page hosted on a web server) using a browser (e.g. Firefox) via the proxy server.
  - Observing the authentication process using a "packet sniffer" (e.g. Wireshark).

When running on Microsoft Windows, some browsers including Firefox may automatically transmit the currently logged-in user's credentials to the proxy server without ever displaying a prompt to the user (to ask for the credentials).

- If NTLM authentication is failing:
  - If possible, temporarily disable NTLM authentication (Windows Authentication) on the proxy server, and enable Basic authentication. (For basic authentication, you will likely also need to add a username and password to the proxy's datastore of login credentials.) This technique can be used to isolate proxy configuration issues (such as the proxy server being unable to authenticate Windows users.)
  - Microsoft ActiveDirectory may be configured to automatically "lock out" an account if the username associated with the account has too many failed authentication attempts. (This includes too many failed attempts to authenticate with a proxy server.) If a user becomes "locked out", a call to the company helpdesk is often required to "unlock" the account.

## 13.17 Non-Interactive Provider to Reliable Multicast

For an introduction to the "Non-Interactive Providers" concept, refer to Section 3.3.2.

This feature extends the functionality of Non-Interactive Providers by letting Non-Interactive Providers publish simultaneously to multiple ADHs (ADH 2.2 or higher). To utilize this feature the non-interactive provider application must use a `RSSL_MCAST_NIPROV`-type connection.

### 13.17.1 Configuration

Configuration support for Non\_Interactive Provider to Reliable Multicast feature is shown below:

**multicastConnection**

```
|
|
| connectionType           RSSL_MCAST_NIPROV
|
| disconnectOnGaps
|
| interfaceName
|
| receiveAddress
|
| receivePortNumber
|
| sendAddress
|
| sendPortNumber
|
| unicastPortNumber
```

The connection type is `RSSL_MCAST_NIPROV`. For this type the configuration parameters listed above are available.

- The `disconnectOnGaps` boolean parameter defaults to False, so if any multicast gap situation occurs 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.
- The `interfaceName` parameter specifies the IP address of the Network Interface Card.
- The `receiveAddress` and `receivePortNumber` parameters specify where the multicast data is received.
- The `sendAddress` and `sendPortNumber` parameters specify where the multicast data is sent.
- The `unicastPortNumber` parameter specifies port used for unicast traffic. It is used internally by RFA for acknowledgements and retransmission requests.

## 13.18 Provider Dictionary Download from an ADH

Publishing applications can request dictionaries from the ADH, which provides a central location for publishers to obtain a dictionary. A login message from the ADH can communicate whether the ADH supports the Dictionary Download feature. Only ADH of at least Version 2.6 support the dictionary download feature.

Two application examples illustrate this feature: `StarterProvider_Interactive` and `StarterProvider_NonInteractive`.

### 13.18.1 OMM Interfaces Support

The Existing `OMMProvider` interface from the OMM package supports the Dictionary Download feature in both Interactive Provider and Non-Interactive Provider.

A new interface (`OMMClientSessionItemIntSpec`) is added which the client uses to specify interest in a Dictionary Download on a specific client session in the OMM Interactive Provider client.

### 13.18.2 Details

To send a dictionary request, the provider must:

- Specify an item interest specification:
- For an OMM Non-Interactive Provider client, create an `OMMItemIntSpec`.
- For an OMM Interactive Provider client, create an `OMMClientSessionItemIntSpec`, and use `setClientSessionHandle()` to populate it with the client session handle.
- Acquire an `OMMMsg` of request type from pool.
- Use `setMsgModelType()` to set the message model type of the request to `RDMMMsgTypes.DICTIONARY`.
- Use `setIndicationFlags()` to set indication flags to `OMMMsg.Indication.NONSTREAMING` and `OMMMsg.Indication.REFRESH`.

---

**NOTE:** The Provider Dictionary Download feature supports only non-streaming requests.

---

- Acquire an attribute object from pool. Set the name of the dictionary using the `OMMAttribInfo`. Populate the request message with the `OMMAttribInfo`.

For details, refer to the *RFA Java Edition RDM Usage Guide*.

- Set the data filter (`setFilter()`) to specify the dictionary type.
- Set the service name (`setServiceName()`) to indicate the service for which it wants the dictionary.
- Set the name (`setName()`) of the dictionary it wants.

Standard dictionaries distributed by Refinitiv are named `RDMFieldDictionary` and `enumtype.def`.

- Set the message on the interest specification by calling `setMsg()`.
- Send a message using `registerClient()` with interest specification and retain the return handle.

### 13.18.3 Application Example

```
OMMMsg msg = pool.acquireMsg();
msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setMsgModelType(RDMMsgTypes.DICTIONARY);
msg.setIndicationFlags(OMMMsg.Indication.NONSTREAMING | OMMMsg.Indication.REFRESH);

OMMAttribInfo attribInfo = pool.acquireAttribInfo();
attribInfo.setName(_fieldDictName);
attribInfo.setServiceName(_servicename);
attribInfo.setFilter(RDMDictionary.Filter.NORMAL);
msg.setAttribInfo(attribInfo);

OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ommItemIntSpec.setMsg(msg);
_fieldDictHandle = _providerDemo._provider.registerClient(_providerDemo._eventQueue,
    ommItemIntSpec, this, null);
```

**Figure 59: Dictionary Download Example**

## 13.19 Application Signing

System administrators can configure Refinitiv Real-Time Distribution System systems such that they require applications to include a Refinitiv's application authorization token in their Login Request when connecting via TCP, HTTP, Encrypted (HTTPS), or Reliable Multicast. To obtain an application authorization token for inclusion in your application, contact your Refinitiv account representative. RFA encrypts the token for secure transmission in the OMM Login request.

On Refinitiv Real-Time Distribution System systems that require tokens:

- Applications without tokens can still connect but can only receive data.
- If an application includes an invalid authorization token, the application cannot access data.

The following code snippet illustrates how to encode an application authorization token in a login request:

```
// allocate a message from the pool; assume the pool has been created
OMMMsg msg = pool.acquireMsg();

// set the message elements
msg.setMsgType(OMMMsg.MsgType.REQUEST);
msg.setModelType(RDMMsgTypes.LOGIN);
ommmsg.setIndicationFlags(OMMMsg.Indication.REFRESH);
// set some elements in attribute information
msg.setAttribInfo(null, "userName", RDMInstrument.NameType.USER_NAME);

// get the encoder and initialize
OMMEncoder encoder = pool.acquireEncoder();
encoder.initialize(OMMTypes.MSG, 500);
encoder.encodeMsgInit(msg, OMMTypes.ELEMENT_LIST, OMMTypes.NO_DATA);
// Release the initial message
pool.releaseMsg(msg);

// encode the element list containing three standard elements
encoder.encodeElementListInit(OMMElementList.HAS_STANDARD_DATA, (short)0, (short)0);

// Encode attrib part of attribute information which is an element list
// Encode ApplicationId, Position & other attrib elements (as shown in other
// sample code)
:
:

// Encode Application Authorization Token.
encoder.encodeElementEntryInit(RDMUser.Attrib.ApplicationAuthorizationToken, OMMTypes.ASCII_STRING);
encoder.encodeString("appAuthToken1", OMMTypes.ASCII_STRING);

encoder.encodeAggregateComplete();

// get the encoded message from the encoder
OMMMsg encodedMessage = (OMMMsg)encoder.getEncodedObject();

// the encoder may be released
pool.releaseEncoder(encoder);
```

#### Example 154: Authorization Token Encoding Example

# Chapter 14 Application Performance Tuning and Considerations

## 14.1 Performance Tuning Overview

There are many performance tradeoffs that can be made when configuring RFA. Users can configure and tune RFA applications to perform best for their specific environment.

RFA applications are structured into three main components called the connection, Session Layer, and client. Depending on the configuration and the application needs, there can be one or more of each component. The client component is the part of the application written by the client or customer.

## 14.2 Threads

RFA is designed to be thread aware and thread safe. RFA can be configured to have a separate thread of execution for each component or share a thread of execution between components. For example, RFA can be configured to use the same thread of execution for the connection, Session Layer, and client. Or, it can be configured to use a separate thread for each component. However, some restrictions to this apply. If there are multiple components of a specific type, then each component may need to have a separate thread of execution.

Threads are used to handle both incoming and outgoing messages. For example, a connection thread would read response messages from the transport and write request messages to the transport.

The main advantage of using multiple threads is the work performed by RFA and/or the client can be divided up between multiple CPU cores. Each thread can be run in parallel on separate cores by the OS. However, if RFA is being run on a machine with a small number of cores, users may find it more efficient to reduce the number of threads used by RFA to be less than or equal to the number of CPU cores on the machine.

The number of threads used by RFA is configured indirectly and is based on the queue configuration, which is described in the following section.

## 14.3 RFA Consumer Queues

RFA threads communicate using queues. There are queues for both inbound and outbound messages, though this section focuses primarily on the inbound queues. The queue used between the connection and Session Layer is called the response queue and is used to queue response messages. The queue used between the Session Layer and the client is called the event queue. If RFA is configured without any queues then a single thread of execution is used for all components, and messages coming from the connection are passed directly to the application via a callback function.

Having an event queue implies that the Session Layer and client will be using separate threads. Setting the `shareConnections` configuration parameter to `true` specifies that a response queue should be used by RFA and also implies that a separate threads should be used for the connection and Session Layer.

The main advantages of using queues and multiple threads are to take advantage of multiple CPU cores, and to provide buffering for high-throughput situations. The disadvantage of using queues is that they introduce additional latency. Putting a message on a queue to pass it between threads and then taking it off requires additional machine resources. It also requires copying the message.

There are some restrictions on when queues may be used. For example, if there are multiple connections sending response messages to the same Session Layer component, a queue must be used. In this case both connections will put messages on the same response queue for the Session Layer to process.

RFA also supports multiple event queues and allows multiple event queues to be grouped together into an event queue group. Event queue groups are useful for prioritizing how messages are dispatched.

Event queues can also be configured to keep statistics. These statistics can be useful for applications to track their usage of the event queue. However RFA incurs additional overhead when keeping event queue statistics.

## 14.4 Threading Models

RFA uses diverse threading models which are discussed in details below. The complete RFA application comprises of three components: application, RFA Session, and RFA Connection. All of the components can run in separate threads, or some can share threads.

The application runs always in a separate thread. However it can be set programmatically whether the incoming messages are processed by application thread or by the RFA thread. The two models are referred to as **callback model** and **client model** and are described in following sections.

The thread sharing between RFA Session and Connection is determined by configuration parameter **shareConnection**. If it is set to false, which is a default configuration, the two components use the same thread, otherwise, they run in separate threads.

### 14.4.1 Callback Model

The Callback Model is used when no event queue is used. RFA will perform callbacks into the application-provided `processevent()` callback for each message received. The application/client code is run by the same thread of execution as the Session Layer component. This model offers the lowest latency because no queues are used. When using this model, the application should limit the amount of work done by the application callback functions so that the RFA thread can service incoming I/O in a timely manner. The callback model can only be used by OMM Consumers.

### 14.4.2 Client Model

The Client Model is used when an event queue is utilized. The application/client code has its own thread of execution. Messages are retrieved from the event queue by client calls to `EventQueue::dispatch()`. Typically, client code is structured in a “dispatch loop,” which is centered around calls to `dispatch()` to check the event queue for any incoming messages. This type of client is generally RFA-centric. The application thread is structured around reading messages from the event queue. This model uses multiple threads and is generally used for high throughput situations.

### 14.4.3 Notification Client Model

The notification client model is a variant of the client model. The application has its own thread of execution. However, the client is notified by a callback from the Session Layer when there is a message to be retrieved from the event queue. This type of client is less RFA-centric than the Client Model described earlier. It is generally assumed the application is focused on non-RFA activities and needs to be notified when a message is available to be read. This type of model is recommended for applications that are run on a separate thread and do not need high throughput or low latency. These types of clients focus primarily on non-RFA activities and only need to use RFA occasionally. This is considered the lowest performing RFA configuration. One disadvantage of this configuration is that the client receives a notification for every single response message received by RFA, which can be costly in high throughput situations. Another potential disadvantage is that it may take some time for the client to react to a notification if it was busy performing other task. Refer to Section 7.4 for the description of interfaces supporting Notification Client. Refer to Section 7.4 for details how to program for Notification Client.

## 14.5 Configuring the RFA Consumer for Performance

There are multiple configurations for configuring an RFA consumer for performance. Each has its own advantages and disadvantages. They are detailed in the sections below.

### 14.5.1 Low Latency with Callback Model

The low latency without an event queue configuration does not use any queues and processes response messages with a single thread, as shown in the diagram below.

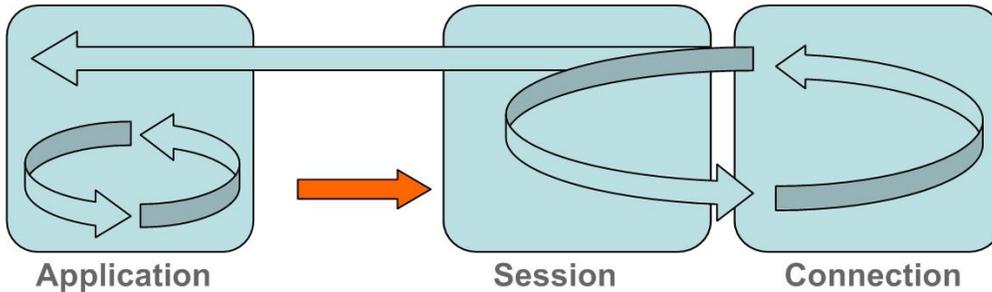


Figure 60: Low Latency with Callback Model

There are two threads in this configuration, one main application thread, and a single RFA thread. The application thread is used to send requests. The RFA thread is used for the connection, Session Layer, and for the application callbacks that process response messages using `processEvent()`. The model is enabled by passing `NULL` for the `EventQueue` argument of `registerClient()` and by setting `shareConnections` configuration parameter to `false`.

Advantages:

- Incurs the lowest possible latency since there is no overhead in queuing or message copying.

Disadvantages:

- Only one CPU core is used for processing response messages
- If the client callback does any significant processing on the single thread being used, then additional messages from the connection may not be processed in a timely manner. (An application can create additional thread to overcome this limitation).
- Since there is only a single thread being used, response messages may not be processed in a timely fashion if the application is continually sending a significant number of request messages. Conversely, request messages may incur latency if a large number of response messages is being received.

### 14.5.2 Low Latency with Client Model

The low latency with an event queue model has an event queue, an application thread, and an RFA thread, as shown in the diagram below.

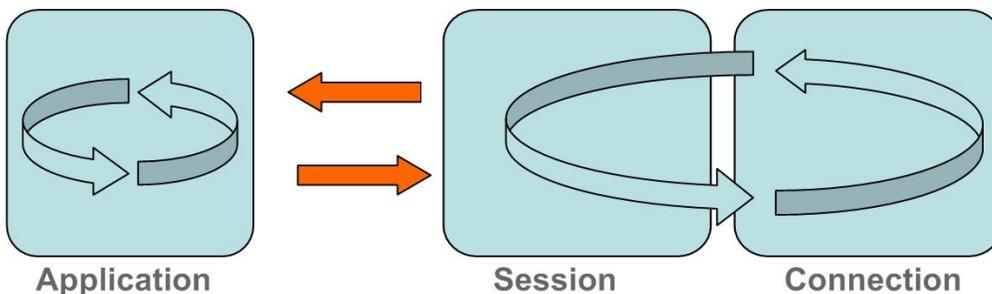


Figure 61: Low Latency with Client Model

There are a total of two threads in this configuration. The main application thread is used to send requests and to process response messages. The second thread is used by RFA for connection and Session Layer processing. The model is enabled by specifying an `EventQueue` argument in `registerClient()` and by setting `shareConnections` **configuration parameter to false**.

Advantages:

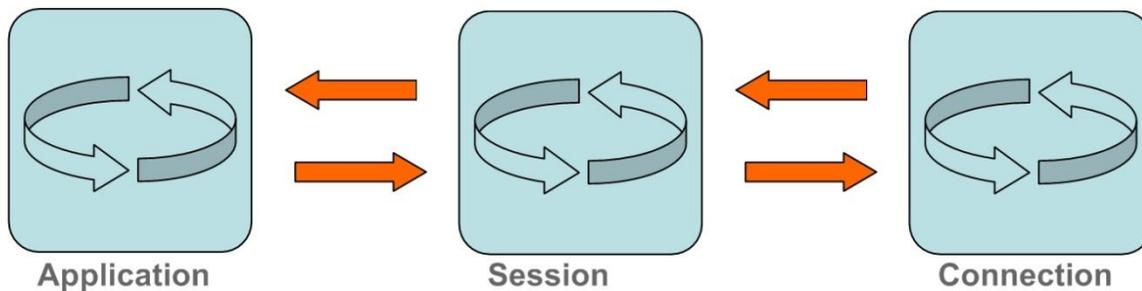
- Isolates RFA from the processing in `processEvent()`; i.e., there is one thread that is always ready to read from the connection even if the client is busy processing a previously received message.
- Offers protection against situations where there is a large burst of messages that need to be read from the connection and are queued up for processing.
- Can take advantage of up to two CPU cores because two threads are being used.

Disadvantages:

- More latency than the full low latency configuration due to the use of an event queue.

### 14.5.3 Full Throughput with Client Model

The full throughput configuration has an event queue, a response queue, an application thread, a Session Layer thread, and an connection thread, as shown in the diagram below.



**Figure 62: Full Throughput with Client Model**

There are a total of three threads in this configuration. The main application thread is used to send requests and to process response messages. The second thread is used by RFA for Session Layer processing. The third thread is used by the RFA connection to read and write messages. The model is enabled by specifying an `EventQueue` argument in `registerClient()` and by setting `shareConnections` **configuration parameter to true**.

Advantages:

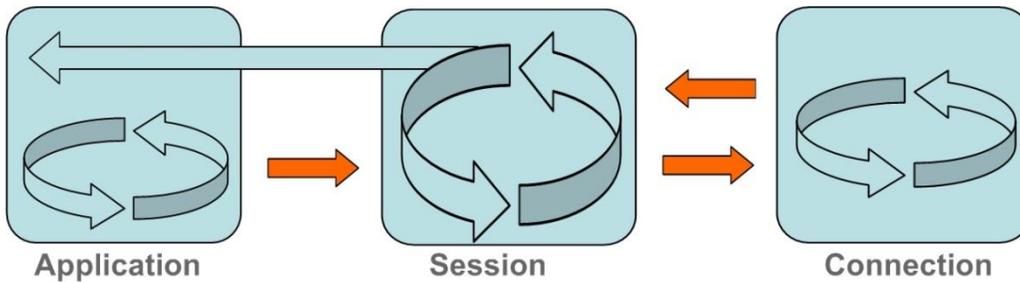
- The three threads allow the work to be spread across three CPU cores.
- Offers the best protection against situations where there is a large burst of messages received from the connection.
- Offers the best performance for applications that are both sending and receiving a significant number of messages at the same time because of the multiple threads.

Disadvantages:

- More potential latency because two queues are being used.

## 14.5.4 Throughput with Callback Model

The throughput with no event queue configuration has a response queue and two threads to process response messages, as shown in the diagram below.



**Figure 63: Throughput with Callback Model**

There are a total of three threads in this configuration. The main application thread is used to send requests. The second thread is used by RFA for Session Layer processing and for application callbacks. The third thread is used by the RFA connection to read and write messages. The model is enabled by specifying `NULL` as the `EventQueue` argument in `registerClient()` and by setting `shareConnections` configuration parameter to true.

Advantages:

- This configuration can use two cores to handle response messages.
- It has a thread that is dedicated to reading messages from the connection and therefore provides good buffering for large bursts of messages.

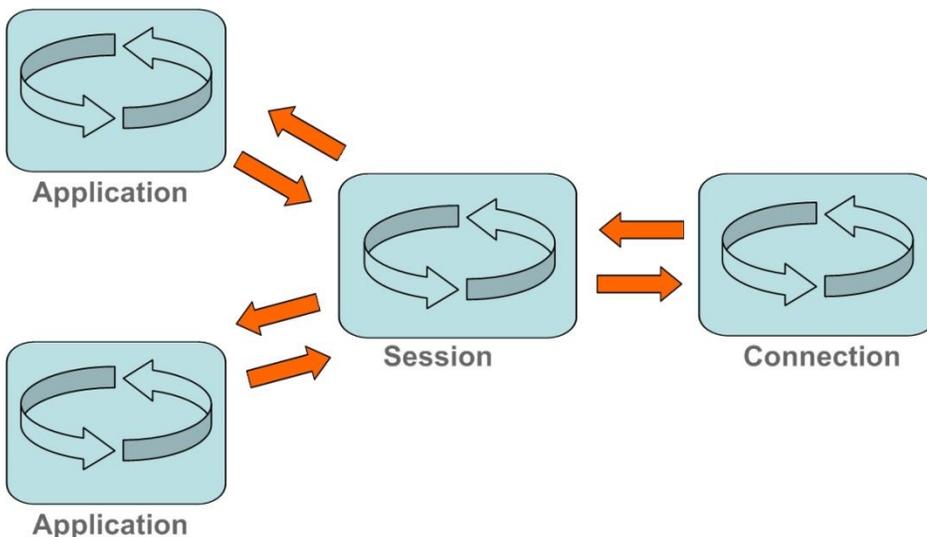
Disadvantages:

- Slightly higher latency than the latency configuration due to the use of a single queue

This model generally provides higher throughput than the latency configuration with an event queue.

## 14.5.5 Multiple Event Queues (Client Model)

The following diagram shows an application utilizing multiple event queues.



**Figure 64: Multiple Event Queues with Client Model**

The above diagram shows a total of four threads: two application threads, a session thread, and a connection thread. However, the number of threads and event queues can vary based on the design on the application.

Developers can create multi-threaded client applications that read from multiple event queues. In the diagram above, a full throughput configuration is shown with two application threads, each with their own event queue. This technique can also be used with a low latency configuration using an event queue.

In situations where the response messages being received are large and complex, the application may spend a significant amount of time decoding them. The advantage of this configuration is that the tasks of decoding and processing messages can be split between multiple application threads. In this case, each application would subscribe to a subset of the desired items and therefore each application thread would have to decode a subset of the messages.

This model is enabled by each application thread specifying a different `EventQueue` argument in `registerClient()` and by setting `shareConnections` **configuration parameter to true**. In this model, each application thread acquires (shares) the same session object.

Advantages:

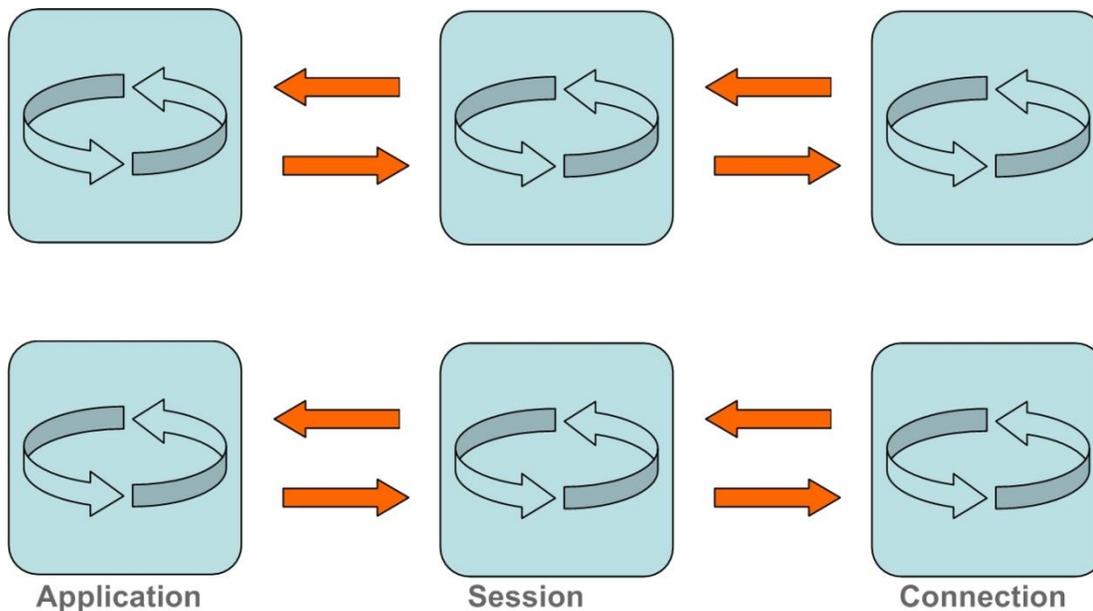
- Allows splitting the application workload between multiple threads, giving more machine resources to decoding and application-specific activities.
- The application can vary the number of event queues and threads to balance the load on the application threads and the available cores on the machine.

Disadvantages:

- Some applications may not lend themselves to this type of multi-threaded configuration.
- Somewhat higher latency due to the use of queues.

### 14.5.6 Horizontal Scaling

Horizontal scaling can be used to further distribute the processing of response messages between multiple CPU cores while running a single process on the machine.



**Figure 65: Multiple Event Queues with Client Model**

In the above diagram there are a total of six threads. There are two application threads, two session threads, and two connection threads. There are also two event queues and two response queues, all running within a single process. This configuration is intended for specialized situations requiring very high throughput.

In this situation, each of the application threads would subscribe to a subset of items so that each of the applications, sessions, and connections are required to process only a subset of the data. If the machine has six or more processor cores, then the OS would be able to distribute this work between all the cores.

Horizontal scaling is scalable and can be used to create applications with 3, 6, 9, 12, or more threads. However, it may not be beneficial to create applications with more threads than there are cores on the machine.

This technique could be accomplished by running multiple processes, each containing an application, Session Layer, and connection thread. However, in general, applications will find it more efficient to operate within a single process.

Horizontal Scaling allows the application to pair a session with a connection. This feature may be utilized by applications that run on multiprocessor cores. The purpose of this feature is increase their total throughput for both response and request messages.

## 14.5.7 Connection Sharing

Multiple Sessions used by the same consumer application can share a single connection (e.g., two different application components using two different Sessions but routing to the same ADS connection). The purpose of this feature is to minimize bandwidth and reduce memory usage.

To configure connection sharing between multiple sessions, the sessions should define the same connection in the `connectionList`. Additionally, the `shareConnection` parameter should be set to `true`.

A consumer application using multiple sessions with shared connections will be served by RFA the same way as if the sessions use dedicated connections. The only difference may be seen during initial consumer startup. When the two sessions return different response for a Login, both of the sessions will be denied.

The Login responses to the clients utilizing the sessions should have almost all of the Attribute Information elements matching. The exceptions are attributes indicating support for the Pause and Resume and Post features.

## 14.5.8 Throttling

If a client application requests a large number of items at the same time, the requests could overflow the outgoing channel. This could cause the channel to disconnect. Additionally, the item images could be received by the consumer application about the same time. If the subscribing application cannot process the images fast enough, the network channel will overflow, causing the Slow Consumer channel to be disconnected. When the subscriber tries to recover, it will again request all of the items, thereby repeating the problem. This situation is often called thrashing.

Consumer applications can use throttling to limit the rate of requests sent to a provider to reduce the probability of thrashing occurring.

The throttling configuration is as follows:

### Connections

```

|
consConnection
|
  throttleBatchCount
  throttleEnabled
  throttleMaxCount
  throttleRequestsPerInterval
  throttleTimerInterval
  throttleType

```

The `throttleEnabled` configuration parameter, if set to true, enables request throttling to control the rate at which requests are sent to the network.

The `throttleType` configuration parameter determines if the throttle queue is based on a timer or on a count of pending requests. Only used when `throttleEnabled = true`.

The `throttleMaxCount` configuration parameter defines the maximum number of pending requests at a time. Only used when `throttleType = count`. When the infrastructure provides an “openLimit”, the value will be the minimum of configured value and the value specified by the provider.

The `throttleBatchCount` configuration parameter specifies the maximum number of requests that are sent out at a time. When the `throttleMaxCount` is reached, then this number of responses must be received before any more requests are sent. Only used when `throttleType = count`.

The `throttleTimerInterval` configuration parameter defines an interval (in milliseconds) at which items (from config variable `throttleRequestsPerInterval`) go to infrastructure. Only used when `throttleType = timer`.

The `throttleRequestsPerInterval` configuration parameter defines how many items go to infrastructure at every interval (from config variable `throttleTimerInterval`). Only used when `throttleType = timer`.

The RFA uses request throttling to avoid thrashing. The throttling will apply when the `throttleEnabled` parameter is set to true. RFA Java has two request throttle queue implementations: one controlled by time interval, and the other controlled by rate. The throttling method is configured using `throttleType` configuration parameter.

If the type is set to **count**, RFA limits the maximum number of pending requests that have not received an image. The number of pending requests is determined by the `throttleMaxCount` configuration parameter. The RFA also limits the number of requests sent at a time. The limit is determined by the `throttleBatchCount` configuration parameter.

If the type is set to **timer**, RFA limits the maximum number of pending requests that have not received an image in a specified time interval. The number of pending requests and the interval are determined by the `throttleRequestsPerInterval` and `throttleTimerInterval` configuration parameters respectively.

RFA Java's throttle queues are configurable on a per connection basis. Most subscribing applications can use the defaults. However, depending on factors such as client machine speed, network bandwidth, and whether the distribution network is point-to-point or multicast, some applications may see better behavior by tuning the configuration.

When the throttling is enabled, all item requests and batch requests, that require refresh are throttled. The batch request can be divided into several subbatches, depending on the `throttleBatchCount` parameter.

## 14.6 Configuring the Provider for Performance

The programming to Client Model versus Callback Model applies to provider application as well. Alike, the configuring `shareConnection` parameter for single thread or separate threads between RFA session and connection, applies too.

## 14.7 Other Tuning Configuration Parameters

The RFA can utilize several other configuration parameters to tune the application, as described below. These parameters apply to consumer and provider applications, unless noted otherwise.

### 14.7.1 `tcp_nodelay`

Setting the `tcp_nodelay` parameter to true will disable the TCP/IP Nagle algorithm used by the operating system in order to provide lower latency. Setting this to false will cause the OS to buffer outgoing messages to increase overall throughput.

### 14.7.2 `initialWatchlistSize`

Initial size of internal watchlists. Applications that request a large number of items can set this to a larger value to avoid extra hash table resizing while the requests are being made. Both, session and connection may specify the watchlist size. However the value for connection may be configured to a different number than for a session, since the two watchlists are different containers.

### 14.7.3 `enableOMMEventAge`

This parameter applies to Refinitiv Source Sink Library type connection only, i.e., consumer connection. If set to true, it enables the methods `getEventAge()` and `getEventAgeNano()`, to return the time difference between the "response message creation time" and the current time.

If disabled, the methods `getEventAge()` and `getEventAgeNano()` returns -1.

## Appendix A Quick Reference

PACKAGE	CONCEPT	DESCRIPTION
LOGGER	Logging levels	A measure of severity for a message, it provides control over which messages will be displayed
LOGGER	Namespace	A name tree used to group and partition messages by their source
COMMON	Buffer	Provides simple encapsulation of raw data as an object that contains both the data and a data length. The Buffer supports typical operations such as copy and compare.
COMMON	Client	Application call-out code that receives dispatched Events. A Client always executes in the application's thread context.
COMMON	Closure	An application may supply a Closure on sending an Interest Specification. RFA returns the Closure in Events. A closure is a reference to an object supplied by the application.
COMMON	Command	An abstract specification that does not establish interest when sent to an Event Source.
COMMON	Concrete Service	A named grouping of data items supplied by a single data vendor. This has a single infrastructure-determined Quality of Service.
COMMON	Context	Integrates multiple RFA packages into a single application and coordinates the interaction between these packages. An application uses the context to manage initialization and shutdown of RFA. Additionally, the context provides versioning information for all RFA packages and underlying libraries.
COMMON	Dispatch	An application action that obtains the next available Event from an Event Queue or Event Queue Group.
COMMON	Event	An asynchronous response message sent to the application that has interest in a particular entity.
COMMON	Event Queue	A container of Events awaiting delivery to an application.
COMMON	Event Queue Group	A container of Event Queues that emulates a single Event Queue. An application typically uses an Event Queue Group to prioritize dispatching amongst different Event Queues or when dispatching from multiple Event Queues in a single thread context.
COMMON	Event Source	An RFA component that typically reacts to an Interest Specification by sending one or more Events. An Event Source provides access to a back-end system.
COMMON	Event Source Factory	An interface for creating Event Sources (e.g. the Session in the Session Layer package)
COMMON	Event Stream	The sequence of Events sent to the application that has interest in a particular entity.
COMMON	Exceptions	A set of RFA-specific exception classes used to convey Error conditions such as invalid configuration or invalid parameters.
COMMON	Handle	A unique identifier representing an open event stream.
COMMON	Interest Specification	A request sent from an application with the goal of opening interest in a particular entity.
COMMON	Message	An abstract container of header and raw data. Messages flow between service provider and service consumer applications.

PACKAGE	CONCEPT	DESCRIPTION
COMMON	Notification Client	Application call-out code used to notify the application of newly available Events on an Event Queue or Event Queue Group. A Notification Client always executes in the RFA's thread context.
COMMON	Principal Identity	Represents a user's identity to support entitlements. The Session Layer package and internal entitlement systems use this Principal Identity.
COMMON	Quality of Service	Specifies a categorical method of classifying Market Data Services provided by the Market Data delivery infrastructure. The Session Layer package and internal entitlement systems use Quality of Service.
COMMON	Response Status	Provides the states and conditions of an item.
COMMON	Service	A namespace that classifies a set of data sources on a network.
COMMON	Service Group	A named grouping of Concrete Services. Service Groups may be supplied from one or multiple data vendors and may have multiple Qualities of Service
COMMON	Service Load	Conveys the load status of Services and contains a <b>load factor</b> , which is a number specified by the providing application. The load factor indicates the relative workload endured by the Service. Both Concrete Services and Service Groups may use the same Service Load events.
COMMON	Service Status	Conveys the status of Services and contains State information (e.g., Up, Down). Both Concrete Services and Service Groups have similar Service Status.
COMMON	Status	Provides a simple Status interface.
CONFIG	Config Database	An in-memory hierarchal configuration store that is set up by the Preferences API and is populated by the user.
CONFIG	Config Node	A node is a hierarchical collection of configuration data. Each node is named in a similar fashion to directories in a hierarchical system. It follows the Java Preferences node naming scheme
CONFIG	Config Subtree	A specific concrete type of Config Node that typically contains a set of child Config Nodes. It is a combination of Java Preferences trees. RFA Java Edition first searches the user tree and then the system tree
CONFIG	Namespace	A name chosen to correspond to a Config Subtree (called a Namespace Tree) under the Config Root. Namespaces are used to logically group sets of configuration hierarchies in a Config Database
CONFIG	Config Root	The uppermost Config Node in a Config Database in which RFA Java stores all its configuration data. It must be: <b>"/com/reuters/rfa"</b>
CONFIG	Component Path	The Config Node(s) under the Namespace node and above the Instance Name node
CONFIG	Instance Name	The Config Node under the Component Path
CONFIG	Softlink	A specific type of Config Node used to point to another Config Node for configuration sharing using a relative path. The notation is [Namespace:]InstanceName
OMM	Actions	Operations on entries within a Container to manage update processing and fragment reconstruction. A Provider may specify actions. A Consumer needs to comply with the actions.
OMM	AttribInfo	Attributes describing the message; This is defined by the <a href="#">OMMAttribInfo</a> class.

PACKAGE	CONCEPT	DESCRIPTION
OMM	Changeable attributes	Attributes that may change during the lifetime of a stream.
OMM	Consumer	An application that uses services; the application sends requests and receives responses.
OMM	Container	An item, e.g. FilterList, Vector, Series, etc., containing <b>entries</b>
OMM	Data	The abstract interface for all data contained in the Data package. An application uses concrete realizations of this to access OMM Data. Represents the actual content.
OMM	Data Decoding	Process of obtaining the actual content (raw data) represented by the data structures.
OMM	Data Encoding	Converting actual content (raw data) into data structures provided by the Data package.
OMM	Def ID	The identifier referencing the definition.
OMM	Defined Data	<p>Data that separates content and definition. The content and definition are separated into discrete segments. Some Containers optionally support the definition segment (e.g. Map, Vector, Series) while other Containers optionally support the content segment (e.g. FieldList, ElementList).</p> <p>The content type always resides in the definition.</p> <p>The definition has either a local scope or global scope. Local scope implies the definition occurs within the same data instance as the content. Global scope implies the definition and content occur across different data instances.</p> <p>Defined data eliminates the need for an application to manage a dictionary of definitions and reduces bandwidth consumption through the ability to distribute <b>entry</b> definitions only once. Bandwidth consumption is further reduced in the case of fixed-width data, as distribution of the width need only occur once.</p>
OMM	Entry	The data contained within a Container, e.g. Field, Vector entry; the entry may contain another Container or a <b>leaf</b> .
OMM	Entry Content	Contained data within an entry.
OMM	Entry Definition	Description of the data (content) contained within an entry.
OMM	Identifier	The specific attribute that uniquely identifies the entry (e.g. the name portion of a name / value pair). The type of the identifier typically differs across entry types. Entries that have no identifier are implicitly identified by their order in the Container.
OMM	Iterators	Utility to step through the data and access individual entries.
OMM	Key Data	Data that typically identifies the associated <b>payload data</b> .
OMM	Leaf	<p>The simplest form of data; represents the base data types.</p> <p>Only one type of leaf is supported and is referred to as the <b>Data Buffer</b>. It may contain the following:</p> <ul style="list-style-type: none"> <li>• Simple types like Int, UInt, Date, Time, etc.</li> <li>• Data types requiring external parsers; e.g. XML, ANSI, etc.</li> </ul>
OMM	Message	<p>Interface that contains header information and possibly data interfaces. Messages flow between service provider and service consumer applications. This is defined by the abstract <b>Msg</b> class.</p> <p>The Message interface can contain another message as its payload.</p>
OMM	Message Model Type	The type identifying the specific message model for OMM .

PACKAGE	CONCEPT	DESCRIPTION
OMM	Multi-part refresh	The collective set of refreshes forming an image, in the case of refresh fragmentation. The final refresh is indicated by the Refresh Complete flag.
OMM	Nested Data	A Container contained within another Container to form a hierarchy.
OMM	OMM Connection Interest Specification	for a declaration of interest in Connection Events.
OMM	OMM Consumer	An Event Source capable of requesting information. It sends requests and receives responses.
OMM	OMM Item Event	An Event with information about the item.
OMM	OMM Item Interest Specification	Specification for interest in Item Events.
OMM	OMM Provider	An Event Source capable of making information available to another application or a back-end system. It receives requests and sends responses.
OMM	Payload Data	Data that satisfies the business purpose of the message. For example in Level 1 data, the payload data contains price discovery information.
OMM	Permission Data	Authorization information with respect to a login context.
OMM	Provider	An application that provides services; the application receives requests and sends responses.
OMM	Refresh fragmentation	The ability of an image to be split across multiple independently-distributed messages.
OMM	Request message	The message that flows from the Consumer to the Provider and carries the request details. The request message is encapsulated in the OMM Item Interest Specification ( <a href="#">OMMItemIntSpec</a> ) Attributes of the request message can be specified using the <a href="#">OMMAttribInfo</a> interface
OMM	Response message	The message that flows from the Provider to the Consumer. The response message is encapsulated in the OMM Item Event ( <a href="#">OMMItemEvent</a> ). The response message attributes can be retrieved using the <a href="#">OMMAttribInfo</a> interface.
OMM	Response status	Indicates stream State or data State
OMM	Response type	Indicates the type of response and is contained in the response message. The response type could be a refresh, update or status.
OMM	Standard data	Data that unites content and definition, represented as one entity. <b>Content</b> is the contained data within an entry. <b>Definition</b> is a content description and consists of the entry identifier and possibly other attributes. The content type may reside within the entry or in an independently distributed dictionary. Standard data is easy to use and is similar to conventional price discovery dataformats such as Marketfeed.
OMM	Summary data	Metadata that describes a Container's entries. For example, summary data could specify the currency of each entry's price or rules regarding the resorting of entries.
Session Layer	Concrete Service	A named grouping of Market Data Items provided from a single back-end system.
Session Layer	Connection	Encapsulates connectivity to a back-end system such as the Refinitiv Data Feed Direct, ADS, and TMF.

PACKAGE	CONCEPT	DESCRIPTION
Session Layer	Contribute	The process of submitting information to a head-end system about a particular entity (e.g., Market Data Items) <sup>1</sup> . A contributing application typically contributes Market Data images to a head-end system.
Session Layer	Dictionary	An entity object containing information for Market Data parsing.
Session Layer	Entitlements Mechanism	Controls the application's ability to subscribe, publish and contribute specific information. RFA implements entitlements through the Refinitiv Data Access Control System.
Session Layer	Event Source	A component that typically reacts to an Interest Specification by sending one or more events.
Session Layer	Market Data Dictionary	An entity object containing information for Market Data parsing.
Session Layer	Market Dataformat	The structure of Market Data. An application may obtain the Market Dataformat(s) to determine which data dictionaries it needs to subscribe to for Market Data parsing.
Session Layer	Market Data Group Status	Conveys status about a set of Market Data Items within a Market Data Service.
Session Layer	Market Data Item	An entity that contains information about a particular Market Data instrument.
Session Layer	Market Data Message Type	The data type associated with a Market Data when subscribing (e.g., image, unsolicited image, update, correction, closing run, status, rename, group change, and permission data).
Session Layer	Market Data Service	Either a Concrete Service or a Service Group.
Session Layer	Market Data Subscriber	An Event Source used for Market Data subscribing.
Session Layer	Market Data Unmanaged Publisher	An Event Source used for Market Data publishing.
Session Layer	Publish/Provide	The process of submitting information to the Refinitiv Real-Time Distribution System infrastructure about a particular entity (e.g., Market Data Items). A publishing application typically publishes a Market Data image and updates.
Session Layer	Quality of Service	A categorical method of classifying Market Data Services provided by the Market Data delivery infrastructure.
Session Layer	Service Info	The collective name for Service Name and ServiceId
Session Layer	Service Group	A named grouping of Concrete Services. Service Groups may be supplied from one or multiple data vendors and may have multiple Qualities of Service.
Session Layer	Session	An Event Source Factory that encapsulates one or more connections. Connections within a Session share the same resources such as a thread context.
Session Layer	Subscribe/Consume	The process of using an event stream to obtain Events having information about a particular entity (e.g., Market Data Items). A subscribing application typically subscribes to receive Market Data image and updates.

---

<sup>1</sup> Contributing is Market Data specific but described here for consistency with publishing.

PACKAGE	CONCEPT	DESCRIPTION
Session Layer	User Validation Mechanism	An application's way of identifying the user. For OMM Event Sources, the application provides the user credentials. For Market Data Event Sources, the application may provide the Principal Identity.

**Table 93: Quick Reference Table**

## Appendix B Deprecated Functionality

This section describes functionality that has been deprecated from RFA. Deprecated RFA classes are supported only for backward compatibility with existing applications. Do not use deprecated classes in new applications.

**NOTE:** Because support for deprecated classes might be removed in a future release, Refinitiv recommends that you actively migrate away from these interfaces.

### B.1 Deprecated Classes

No classes were deprecated in this release.

### B.2 Deprecated Fields

The following table lists the fields that are deprecated.

**NOTE:** If the current RDM or RDMDictionary still specifies deprecated OMMTypes, you will need to continue to use those deprecated OMMTypes until a replacement is available.

DEPRECATED FIELDS	
OMMTypes.ANY	OMMTypes.REAL32
OMMTypes.INT32	OMMTypes.REAL64
OMMTypes.INT64	OMMTypes.REAL32_RB
OMMTypes.INT32_1	OMMTypes.REAL64_RB
OMMTypes.INT32_2	OMMTypes.REAL_RB
OMMTypes.INT32_4	OMMState.UNSPECIFIED
OMMTypes.INT64_8	Event.LICENSING_EVENT
OMMTypes.UINT32	Event.ENTITLEMENTS_AUTHENTICATION_EVENT
OMMTypes.UINT64	OMMMsg.MsgType.STREAMING_REQ
OMMTypes.UINT32_1	OMMMsg.MsgType.NONSTREAMING_REQ
OMMTypes.UINT32_2	OMMMsg.MsgType.PRIORITY_REQ
OMMTypes.UINT32_4	OMMMsg.Indication.RESUME_REQ
OMMTypes.UINT64_8	OMMMsg.Indication.RESUME_WITH_REFRESH_REQ

**Table 94: Deprecated Fields**

## B.3 Deprecated Methods

The following table lists the deprecated methods and replacements.

DEPRECATED METHODS	EXISTING REPLACEMENT METHODS
OMMNumeric.getIntValue()	OMMNumeric.getLongValue()
OMMNumeric.toInt()	OMMNumeric.toLong()
OMMEncoder.encodeInt32( int )	OMMEncoder.encodeInt( long )
OMMEncoder.encodeInt64( long )	OMMEncoder.encodeInt( long )
OMMEncoder.encodeUInt32( int )	OMMEncoder.encodeUInt( long )
OMMEncoder.encodeUInt64( long )	OMMEncoder.encodeUInt( long )
OMMEncoder.encodeUInt64( BigInteger )	OMMEncoder.encodeUInt( BigInteger )
OMMEncoder.encodeReal32( int, byte )	OMMEncoder.encodeReal( long, byte )
OMMEncoder.encodeReal64( long, byte )	OMMEncoder.encodeReal( long, byte )
FieldDictionary.addEnumFidDef(short, String, String, short, short, short, String, short, short, Boolean)	

**Table 95: Deprecated Methods and Replacements**

© Copyright 2007 - 2017, 2019, 2020 - 2021 Refinitiv. All rights reserved.

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