

REFINITIV ROBUST FOUNDATION API V8.2.2.L1

DEVELOPERS GUIDE

.NET EDITION

© Refinitiv 2015 - 2017, 2020, 2022. 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

Chapter 1	Introduction	1
1.1	Purpose	1
1.2	Scope.....	1
1.3	Audience.....	1
1.4	Product Description	1
1.5	Using This Document	3
1.5.1	<i>Organization</i>	<i>3</i>
1.5.2	<i>Source Code.....</i>	<i>3</i>
1.6	Conventions.....	4
1.6.1	<i>Typographic.....</i>	<i>4</i>
1.6.2	<i>Notation</i>	<i>4</i>
1.6.3	<i>Activity Diagrams</i>	<i>4</i>
1.6.4	<i>Programming Examples</i>	<i>5</i>
1.7	Documentation Feedback.....	5
1.8	Acronyms and Abbreviations.....	6
1.9	References	7
Chapter 2	Product Description	8
2.1	What is RFA .NET API?	8
2.2	Features of RFA	9
2.3	Package Overview.....	12
Chapter 3	RFA Concepts	14
3.1	Refinitiv APIs	14
3.1.1	<i>Refinitiv Robust Foundation API (RFA).....</i>	<i>14</i>
3.1.2	<i>Enterprise Transport API (ETA).....</i>	<i>15</i>
3.1.3	<i>A Contrast of RFA and ETA</i>	<i>15</i>
3.1.4	<i>Performance</i>	<i>16</i>
3.1.5	<i>Fuctionality.....</i>	<i>16</i>
3.1.5.1	General Capability Comparison.....	16
3.1.5.2	Layer-Specific Capability Comparison	18
3.2	Concepts: Consumer/Provider	19
3.2.1	<i>Consumer</i>	<i>20</i>
3.2.1.1	Subscription: Request/Response	21
3.2.1.2	Watchlist	21
3.2.1.3	Batch.....	22

3.2.1.4	View	23
3.2.1.5	Symbol List	24
3.2.1.6	Enhanced Symbol List.....	25
3.2.1.7	Pause/Resume	25
3.2.1.8	Posting.....	27
3.2.1.9	Generic Message	28
3.2.1.10	Connection Recovery	28
3.2.1.11	Item Recovery and SingleOpen.....	28
3.2.1.12	Warm Standby	28
3.2.1.13	Private Stream	29
3.2.1.14	Connection Redirection (Load Balancing)	30
3.2.2	<i>Provider</i>	30
3.2.2.1	Interactive Provider.....	31
3.2.2.2	Non-Interactive Provider	32
3.2.3	<i>Hybrids</i>	34
3.3	Model Overview	35
3.3.1	<i>Open Message Model (OMM)</i>	35
3.3.2	<i>Domain Message Model (DMM)</i>	35
3.3.3	<i>Refinitiv Domain Model (RDM)</i>	35
3.3.4	<i>User Defined Domain Model</i>	35
3.3.5	<i>Refinitiv Wire Format (RWF)</i>	35
3.4	OMM Concepts.....	36
3.4.1	<i>OMM Messages</i>	36
3.4.1.1	Request Message.....	36
3.4.1.2	Response Message	36
3.4.1.3	Generic Message	37
3.5	Refinitiv Domain Models (RDM) Overview	38
3.5.1	<i>Service, Concrete Service, and Service Group</i>	39
3.6	OMM Data Constructs	40
3.6.1	<i>Data Types</i>	40
3.6.2	<i>Primitive Types</i>	41
3.6.3	<i>Container Types</i>	42
3.6.4	<i>Summary Data</i>	43
3.6.5	<i>Defined Data</i>	43
3.6.6	<i>Iterators</i>	43
Chapter 4	System View	44

4.1	Background.....	44
4.1.1	<i>RTDS Connection and Protocol Types</i>	44
4.1.2	<i>RTDS Components</i>	44
4.1.3	<i>RTDS Data Formats</i>	45
4.1.4	<i>Connections and Supported Data Dictionaries</i>	45
4.2	OMM Infrastructures	46
4.2.1	<i>Refinitiv Real-Time Advanced Distribution Server (ADS)</i>	47
4.2.2	<i>Refinitiv Real-Time Advanced Distribution Hub (ADH)</i>	47
4.2.3	<i>Refinitiv Real-Time Advanced Transformation Server (ATS)</i>	48
4.2.4	<i>Refinitiv Real-Time</i>	49
4.2.5	<i>Refinitiv Data Feed Direct (RDF Direct)</i>	50
4.2.6	<i>Direct-Connect</i>	51
4.2.7	<i>Internet Connectivity via HTTP and HTTPS</i>	52
Chapter 5	Common Package	53
5.1	Common Package Concepts.....	53
5.1.1	<i>Event Distribution Model</i>	53
5.1.1.1	Interest Specification	54
5.1.1.2	Events and Event Streams	54
5.1.1.3	Event Queues and Event Queue Groups	54
5.1.1.4	Opening and Closing Event Streams	55
5.1.1.5	Completion Events	55
5.1.1.6	Closures.....	55
5.1.1.7	Notification Clients	56
5.1.1.8	Example of the Event Distribution Model	56
5.1.2	<i>General Common Package Concepts</i>	57
5.2	Common Package Usage.....	58
5.2.1	<i>Event Distribution Model</i>	58
5.2.1.1	Using Event and Event Queue	60
5.2.1.2	Using Event Queue Group	63
5.2.1.3	Using Notification Client	65
5.2.1.4	Using Event Distribution Model in a Single-thread Context	67
5.2.1.5	Using Event Distribution Model in a Multiple-thread Context	67
5.2.1.6	Event Queue with Statistics	68
5.2.1.7	Event Distribution Usage Guideline.....	70
5.2.2	<i>Context</i>	71
5.2.2.1	RFA Version Info	71

5.2.3	<i>Buffer</i>	72
5.2.4	<i>RFA_String and RFA_WString</i>	73
5.2.4.1	<i>RFA_String Performance Consideration</i>	73
5.2.5	<i>RMTEConverter</i>	74
5.2.6	<i>Exception</i>	75
5.2.7	<i>Quality of Service</i>	78
5.2.7.1	<i>Quality of Service Request</i>	79
5.2.7.2	<i>Quality of Service Item Event</i>	79
5.2.7.3	<i>Quality of Service Recovery</i>	80
5.2.7.4	<i>Default Value</i>	80
5.2.7.5	<i>Quality of Service Scenario</i>	81
Chapter 6	Data Package	82
6.1	<i>Data Package Concepts</i>	82
6.1.1	<i>Logical Composite Pattern</i>	84
6.1.2	<i>Data Uniformity</i>	85
6.1.3	<i>Data Types and Data Containers</i>	85
6.1.3.1	<i>Data Type Classification</i>	86
6.1.3.2	<i>Data Containers</i>	89
6.1.4	<i>Standard Data and Defined Data</i>	90
6.1.5	<i>Container Property Breakdown</i>	90
6.1.6	<i>Leaf Properties</i>	91
6.1.6.1	<i>Blanking</i>	91
6.1.6.2	<i>Array Widths</i>	91
6.1.6.3	<i>Encoded Types</i>	91
6.1.7	<i>Entry Properties</i>	92
6.1.7.1	<i>Permission Data</i>	92
6.1.7.2	<i>Rippling</i>	92
6.1.7.3	<i>Entry Properties Comparison</i>	92
6.1.7.4	<i>Entry Actions</i>	93
6.1.8	<i>Data Types in Depth</i>	95
6.1.8.1	<i>DataBuffer</i>	95
6.1.8.2	<i>Array</i>	99
6.1.8.3	<i>FieldList</i>	99
6.1.8.4	<i>FieldList Info</i>	102
6.1.8.5	<i>ElementList</i>	102
6.1.8.6	<i>Series</i>	104

6.1.8.7	Vector	106
6.1.8.8	Map	109
6.1.8.9	FilterList	112
6.1.9	<i>Data Definition</i>	114
6.1.9.1	ElementList Definition	115
6.1.9.2	FieldList Definition	116
6.1.10	<i>Masks</i>	117
6.1.11	<i>Advanced Distribution Concepts</i>	117
6.1.11.1	Fragmentation.....	117
6.1.11.2	PartialUpdateReadIterator.....	117
6.1.11.3	Working with Encoded Buffers	118
6.1.11.4	Versioning Support	118
6.1.11.5	Reusable Objects	121
6.2	Data Package Usage.....	122
6.2.1	<i>Data Encoding</i>	122
6.2.1.1	General Data Encoding Process	122
6.2.1.2	Encoding Data Definition	124
6.2.1.3	Encoding DataBuffer	126
6.2.1.4	Encoding Array	127
6.2.1.5	Encoding ElementList.....	128
6.2.1.6	Encoding FieldList	130
6.2.1.7	Encoding FilterList	131
6.2.1.8	Encoding Map.....	132
6.2.1.9	Encoding Series	136
6.2.1.10	Encoding Vector	136
6.2.2	<i>Data Decoding</i>	138
6.2.2.1	General Data Decoding Process	138
6.2.2.2	Data Type Identifying.....	140
6.2.2.3	Decoding Data Definition.....	141
6.2.2.4	Decoding Data Buffer	142
6.2.2.5	Decoding Array	144
6.2.2.6	Decoding ElementList.....	145
6.2.2.7	Decoding FieldList	147
6.2.2.8	Decoding FilterList	150
6.2.2.9	Decoding Map.....	151
6.2.2.10	Decoding Series	154

6.2.2.11	Decoding Vector	155
6.2.2.12	Decoding Permission Data	156
Chapter 7	Message Package	157
7.1	Message Package Concepts	157
7.1.1	<i>Message Structure Concepts</i>	159
7.1.1.1	AttribInfo	159
7.1.1.2	Manifest	161
7.1.1.3	Payload	162
7.1.1.4	Message Type	163
7.1.2	<i>Symmetric Messaging Paradigm</i>	173
7.1.3	<i>Common Message Inheritance</i>	173
7.1.4	<i>Message Payload and Attribute Data</i>	173
7.1.5	<i>Refresh Fragmentation</i>	174
7.1.5.1	Final Refresh Completeness	174
7.1.6	<i>Priority</i>	175
7.1.7	<i>Interface Forwarding</i>	175
7.1.8	<i>Masks</i>	175
7.1.8.1	Interaction Type	176
7.1.8.2	Hint Mask	176
7.1.8.3	Indication Mask	177
7.1.9	<i>Batching Items in a Request Message</i>	177
7.1.10	<i>Dynamic View</i>	178
7.1.10.1	Views in RDM/DMM	178
7.1.10.2	Enhanced Symbol List (RDM SymbolList)	178
7.1.11	<i>Item Groups</i>	179
7.1.11.1	Provider Processing	179
7.1.11.2	Consumer Processing	179
7.1.11.3	Item GroupId Scenario	180
7.1.12	<i>Private Streams</i>	180
7.1.13	<i>Visible Publisher Identifier (VPI)</i>	181
7.1.14	<i>Response Status (Stream States and Data States)</i>	182
7.2	Message Package Usage	185
7.2.1	<i>Message Encoding</i>	186
7.2.1.1	Encoding Request Message	187
7.2.1.2	Encoding Response Message	194
7.2.1.3	Encoding Post Message	195

7.2.1.4	Encoding Ack Message	198
7.2.1.5	Encoding Generic Message	199
7.2.1.6	Message Validation	200
7.2.2	<i>Message Decoding</i>	201
7.2.2.1	Usage of Masks	202
7.2.2.2	Decoding Message Type.....	202
7.2.2.3	Decoding Request Message	202
7.2.2.4	Decoding Response Message.....	204
7.2.2.5	Decoding Post Message.....	208
7.2.2.6	Decoding Ack Message.....	210
7.2.2.7	Decoding Generic Message	212
Chapter 8	Configuration Package	215
8.1	Configuration Package Concepts	215
8.1.1	<i>Hierarchy Representation, Config Trees, Config Values, and Namespaces</i>	215
8.1.2	<i>RFA Namespaces</i>	219
8.1.2.1	RFA Component Names and the Default Namespace	219
8.1.2.2	RFA Component Configuration	219
8.1.3	<i>Configuration Sharing and Component Instance Sharing</i>	220
8.1.4	<i>Types of Configuration Values</i>	220
8.2	Configuration Package Usage.....	221
8.2.1	<i>Populating the Config Database from the Windows Registry</i>	221
8.2.1.1	Softlinks and Hardlinks	222
8.2.2	<i>Populating the Config Database from a File</i>	223
8.2.3	<i>Populate Config Database</i>	224
8.2.3.1	Initializing and Populating the RFA Config Database	225
8.2.3.2	Shutting down Config Database	227
8.2.3.3	Populating an Application Specific Config Database	227
8.2.4	<i>Query Configuration Information</i>	228
8.2.4.1	Retrieving RFA Configuration Nodes	228
8.2.4.2	Iterating RFA Configuration Nodes	229
8.2.4.3	Iterating RFA Configuration Nodes by using Foreach Loop.....	230
8.3	Configuration Package: Putting it All Together.....	231
8.3.1	<i>Populating and Querying a Config Database</i>	231
8.3.2	<i>Merging Multiple Namespaces from a Windows Registry Configuration</i>	233
Chapter 9	SessionLayer Package	236
9.1	SessionLayer Package Concepts	236

9.1.1	<i>General Session Layer Concepts</i>	236
9.1.1.1	Consumers and Providers	238
9.1.1.2	Application Threading Model	238
9.1.1.3	Low Latency and High Throughput	239
9.1.1.4	Horizontal Scaling.....	240
9.1.1.5	Pending Request Queue	240
9.1.1.6	Session Sharing and Multiple Event Sources	240
9.1.1.7	Event Distribution with Multiple Threads	241
9.1.2	<i>OMM Concepts</i>	241
9.1.2.1	OMM Item State	242
9.1.3	<i>Events and Cmds</i>	243
9.2	SessionLayer Package Usage	244
9.2.1	<i>Session</i>	244
9.2.1.1	Session Initialization	244
9.2.1.2	Session Cleanup	244
9.2.2	<i>Events and Cmds</i>	245
9.2.2.1	Event Registration	245
9.2.3	<i>OMMConsumer</i>	246
9.2.3.1	OMMConsumer Initialization	246
9.2.3.2	Event Registration	247
9.2.3.3	Submit Message	247
9.2.3.4	Modify Event Stream	247
9.2.3.5	Event Unregistration	248
9.2.3.6	OMMConsumer Cleanup.....	248
9.2.4	<i>Interactive OMMPProvider</i>	248
9.2.4.1	Interaction Type	249
9.2.4.2	Client Session Handles and Request Tokens	249
9.2.4.3	OMMPProvider Initialization.....	251
9.2.4.4	Event Registration	251
9.2.4.5	Submit Message	251
9.2.4.6	Unregistration Event	252
9.2.4.7	CleanUp.....	253
9.2.5	<i>Non-Interactive OMMPProvider</i>	253
9.2.5.1	Login Request.....	253
9.2.5.2	Item Tokens	254
9.2.5.3	OMMPProvider Initialization.....	254

9.2.5.4	Event Registration	255
9.2.5.5	Submit Message	255
9.2.5.6	Unregistration Event	256
9.2.5.7	Cleanup	256
Chapter 10	Logger Package	257
10.1	Logger Package Concepts	257
10.1.1	<i>Threading Model</i>	259
10.2	Logger Package Usage	259
10.2.1	<i>Configuring the Logger Package</i>	259
10.2.2	<i>Logging in Application</i>	260
10.2.3	<i>Application Logger</i>	261
10.2.3.1	Initializing Application Logger	261
10.2.3.2	Shutting Down Application Logger	262
10.2.4	<i>Application Logger Monitor</i>	263
10.2.4.1	Initializing Application Logger Monitor	264
10.2.4.2	Registering for Log Events	264
10.2.4.3	Closing an Event Stream for the Application Logger Monitor	266
10.2.4.4	Shutting Down Application Logger Monitor	266
10.2.5	<i>Component Logger</i>	267
10.2.5.1	Initializing Component Logger	268
10.2.5.2	Submitting Log Events	268
10.2.5.3	Shutting Down Component Logger	269
10.2.5.4	Using a Message File	270
10.2.5.5	Using a LogMsgMap	272
10.2.5.6	Using the Windows Event Logger	272
10.2.6	<i>Logger Internationalization</i>	272
10.3	Logger Usage Guidelines	273
10.3.1	<i>Control Log Event Fanout</i>	273
10.3.2	<i>State and Status Code in Status Interface</i>	273
10.3.3	<i>RFA Application Logger Name</i>	273
10.3.4	<i>Mixed Component Logger Configuration</i>	273
10.3.5	<i>RFA Application Logger Shutdown</i>	273
10.3.6	<i>Default Namespace</i>	273
Chapter 11	Implementing OMM Consumer	274
11.1	Application Lifecycle	274
11.2	Initialization OMM Consumer	276

11.2.1	<i>Acquire Session</i>	276
11.2.2	<i>Create Event Source</i>	276
11.2.3	<i>Create Event Queue</i>	277
11.3	<i>Process for Creating Request Message</i>	278
11.3.1	<i>Login</i>	279
11.3.2	<i>Directory</i>	280
11.3.3	<i>Dictionary</i>	280
11.3.4	<i>Level 1 Data (Market Price)</i>	280
11.3.5	<i>Level 2 Data (Market by Order, Market by Price, Market Maker and Symbol List)</i>	281
11.3.6	<i>Create a Stream for Sending Generic Message</i>	281
11.3.7	<i>Create a Stream for Sending Post Message</i>	281
11.4	<i>Registering for Events from OMM Consumer</i>	282
11.4.1	<i>Registering for Item Events</i>	282
11.4.2	<i>Registering for Connection Events</i>	282
11.4.3	<i>Registering for Connection Statics Events</i>	283
11.4.4	<i>Registering for Error Events</i>	284
11.5	<i>Dispatching Event Queue</i>	284
11.6	<i>Event Processing</i>	285
11.6.1	<i>Processing Item Event</i>	286
11.6.2	<i>Processing Connection Event</i>	287
11.6.3	<i>Processing Error Event</i>	288
11.6.4	<i>Processing Response Message</i>	288
11.6.4.1	<i>Processing Login Response Message</i>	288
11.6.4.2	<i>Processing Directory Response Message</i>	289
11.6.4.3	<i>Processing ConsumerStatus/SourceMirroring Data from the Directory Response Message</i>	290
11.6.4.4	<i>Processing Response Messages Containing Market Information</i>	290
11.6.5	<i>Processing Generic Message</i>	291
11.6.6	<i>Processing Ack Message</i>	291
11.7	<i>Process for Sending Messages using OMM Consumer</i>	292
11.7.1	<i>Generic Message</i>	292
11.7.1.1	<i>Creating Generic Message</i>	292
11.7.1.2	<i>Submitting Generic Message</i>	293
11.7.2	<i>Post Message</i>	294
11.7.2.1	<i>Creating Post Message</i>	294
11.7.2.2	<i>Submitting Post Message</i>	295
11.8	<i>Unregistering Events from OMM Consumer</i>	296

11.8.1	<i>Unregister Connection Events</i>	296
11.8.2	<i>Unregister Market Information Events</i>	297
11.8.3	<i>Unregister All Event Streams at Once</i>	297
11.9	<i>Cleaning Up</i>	298
11.9.1	<i>Deactivate Event Queue</i>	298
11.9.2	<i>Destroy Event Queue</i>	298
11.9.3	<i>Destroy Event Source</i>	299
11.9.4	<i>Release Session</i>	299
Chapter 12	Implementing OMM Provider	300
12.1	<i>Application Lifecycle</i>	300
12.2	<i>Interactive Provider</i>	301
12.2.1	<i>Initialization OMM Provider</i>	302
12.2.1.1	<i>Acquire Session</i>	302
12.2.1.2	<i>Create Event Source</i>	302
12.2.1.3	<i>Initializing Event Queue</i>	303
12.2.2	<i>Registering Events for Interactive Provider</i>	304
12.2.2.1	<i>Initial Steps for Registering for Events</i>	306
12.2.2.2	<i>Registering to Accept a Client Session</i>	307
12.2.3	<i>Dispatching Event Queue</i>	308
12.2.4	<i>Event Processing for Interactive Provider</i>	309
12.2.4.1	<i>Handling ProcessEvent() in the application</i>	309
12.2.4.2	<i>Handling Consumer Client Session Events (Requests)</i>	310
12.2.4.3	<i>Handling Consumer Solicited Item Events: Request Message</i>	312
12.2.4.4	<i>Handling Consumer SolicitedItemEvents: Generic Messages</i>	316
12.2.4.5	<i>Handling Consumer SolicitedItemEvents: Post Message</i>	316
12.2.4.6	<i>Handling CmdError Events</i>	317
12.2.4.7	<i>Handling Connection Events</i>	317
12.2.4.8	<i>Handling LoggerNotify Events</i>	318
12.2.4.9	<i>Handling Completion Events</i>	318
12.2.4.10	<i>Handling Connection Statistics Events</i>	318
12.2.5	<i>Sending Messages for Interactive Provider</i>	319
12.2.5.1	<i>Response Message</i>	319
12.2.5.2	<i>Generic Message</i>	321
12.2.5.3	<i>Ack Message</i>	322
12.2.6	<i>Unregistering Events for Interactive Provider</i>	324
12.2.6.1	<i>Closing items and Handling Closes</i>	324

12.2.6.2	Logging out a Client Session	325
12.2.6.3	Disconnecting a Client Session	325
12.2.6.4	Closing a Client Session	325
12.2.6.5	Disabling a Client Session Request	326
12.2.6.6	Disabling Other Interest Spec	326
12.2.7	<i>Cleaning up</i>	326
12.3	Non-Interactive Provider	329
12.3.1	<i>Initialization OMM Provider</i>	330
12.3.2	<i>Registering Events for Non-Interactive Provider</i>	330
12.3.2.1	Initial Steps for Registering for Events	331
12.3.3	<i>Dispatching Event Queue</i>	332
12.3.4	<i>Event Processing for Non-Interactive Provider</i>	332
12.3.4.1	Handling ProcessEvent() in the application	332
12.3.4.2	Handling Item Events (Login Events)	333
12.3.4.3	Login Events: Login Success	334
12.3.4.4	Login Events: Other Login States	334
12.3.4.5	Handling CmdError Events	336
12.3.4.6	Handling Connection Events	337
12.3.4.7	Handling LoggerNotify Events	337
12.3.4.8	Handling Connection Statistics Events	338
12.3.5	<i>Sending Messages for Non-Interactive Provider</i>	338
12.3.5.1	Response Message	338
12.3.5.2	Generic Message	341
12.3.6	<i>Unregisters Events for Non-Interactive Provider</i>	342
12.3.6.1	Logging out	342
12.3.6.2	Disabling others Interest Spec	342
12.3.7	<i>Cleaning up</i>	342
Chapter 13	RFA Feature in Details	343
13.1	Batch	343
13.2	Dynamic View	346
13.3	Generic Message	347
13.4	Posting	348
13.5	Private Stream	349
13.5.1	<i>Private Streams Rules and Constraints</i>	349
13.6	Service Group	350
13.7	Connection and Item Recovery	351

13.8	Item Group Management.....	352
13.9	Connection Redirection (Load Balancing)	353
13.10	Tunneling	353
13.11	Pause and Resume	354
13.11.1	Optimized Pause and Resume.....	354
13.11.2	Pause and Resume Message Flow.....	355
13.11.3	Handling Handle	355
13.11.4	Pause All and Resume All.....	355
13.12	Warm Standby	357
13.12.1	Effects in the Consumer	358
13.12.2	Effects in the Provider	360
13.13	Horizontal Scaling.....	361
13.13.1	Effects in the Consumer	361
13.13.2	Effects in the Provider	363
13.14	Service ID vs. Service Name	364
13.15	RDMFieldDictionary Utility	365
13.15.1	Consumer Handling	365
13.15.2	Provider Handling	366
13.16	News Headlines	368
13.17	Using the DACSLOCK API	368
13.18	Parsing AnsiPage Data	369
13.19	Application Signing	370
Chapter 14	Performance Consideration	371
14.1	Thread	371
14.2	RFA Consumer Queue	371
14.2.1	Response Queue Parameter.....	372
14.2.1.1	responseQueueMaxBatchSize	372
14.2.1.2	responseQueueBias	372
14.2.1.3	responseQueueBatchInterval	372
14.3	Threading Model.....	373
14.3.1	Callback Model	373
14.3.2	Client Model.....	373
14.3.3	Notification Client Model.....	373
14.4	Configuring RFA Consumer for Performance	374
14.4.1	Single Threaded with No Event Queue (Callback Model).....	374
14.4.2	Single Threaded with an Event Queue (Client Model)	375

14.4.3	<i>“Full” Dual-Threaded Configuration (Client Model)</i>	376
14.4.4	<i>Dual-Threaded with No Event Queue (Callback Model)</i>	376
14.4.5	<i>Multiple Event Queues (Client Model)</i>	377
14.4.6	<i>Horizontal Scaling (Client Model)</i>	378
14.5	Configuring RFA Provider for Performance	379
14.5.1	<i>Direct Write Model</i>	379
14.5.1.1	forceFlushOnWrite	379
14.5.1.2	tcp_nodelay	379
14.5.1.3	outBoundMessagePacking	379
14.6	Memory Management	380
14.7	Object Reuse	381
14.8	OMM Message Pool	381
14.9	OMM Consumer Request Throttling	382
14.10	Using Service Groups: Service Renaming / Aliasing	383
Chapter 15	Deprecated Functionality	384
15.1	OMMPerfMode Session Configuration	384
Appendix A	Deployment	385
Appendix B	Implementation	387
B.1	Using IDisposable Interface	387
B.2	Data Scope Handling	387
B.3	IEnumerable and IEnumerator	387
B.4	Setting Reference Type	388
Appendix C	Application Design	389
C.1	Typical Consumer Application	389
C.2	Complex Consumer Application	390
C.3	Typical Provider Application	391
C.4	Hybrid Application	392

List of Examples

EXAMPLE 1: INITIALIZE EVENT QUEUE	60
EXAMPLE 2: DISPATCH FROM EVENT QUEUE WITH TIMEOUT	61
EXAMPLE 3: PROCESS EVENT IN CLIENT	61
EXAMPLE 4: SHUTDOWN EVENT QUEUE	62
EXAMPLE 5: INITIALIZE EVENT QUEUE GROUP	64
EXAMPLE 6: SHUTDOWN EVENT QUEUE GROUP	64
EXAMPLE 7: PROCESS NOTIFICATION CLIENT	67
EXAMPLE 8: INITIALIZE CONTEXT	71
EXAMPLE 9: UNINITIALIZE CONTEXT	71
EXAMPLE 10: GET RFA PRODUCT VERSION	71
EXAMPLE 11: GET PACKAGE VERSIONS	72
EXAMPLE 12: BUFFER USAGE	72
EXAMPLE 13: RFA_STRING USAGE	73
EXAMPLE 14: RMTESCONVERTER USAGE	74
EXAMPLE 15: EXCEPTION HANDLING IN RFA	77
EXAMPLE 16: STARTERPROVIDER_NONINTERACTIVE PASSING VERSION INFORMATION	120
EXAMPLE 17: STARTERPROVIDER_INTERACTIVE PASSING VERSION INFORMATION	120
EXAMPLE 18: STARTERCONSUMER_GENERICMSG PASSING VERSION INFORMATION	120
EXAMPLE 19: REUSABLE OBJECT	121
EXAMPLE 20: ENCODING ELEMENTENTRYDEF	124
EXAMPLE 21: ENCODING ELEMENTLISTDEF	125
EXAMPLE 22: ENCODING FIELDLISTDEF	125
EXAMPLE 23: ENCODING DATABUFFER	126
EXAMPLE 24: ENCODING DATABUFFER	126
EXAMPLE 25: ENCODING DATABUFFER	126
EXAMPLE 26: ENCODING ARRAY WITH ARRAYWRITEITERATOR	127
EXAMPLE 27: ENCODING ARRAY WITH SINGLEWRITEITERATOR	127
EXAMPLE 28: ENCODING ELEMENTLIST WITH ELEMENTLISTWRITEITERATOR	128
EXAMPLE 29: ENCODING ELEMENTLIST WITH SINGLELISTWRITEITERATOR	128
EXAMPLE 30: ENCODING ELEMENTLIST AND ELEMENTLISTDEF WITH ELEMENTLISTWRITEITERATOR	129
EXAMPLE 31: ENCODING ELEMENTLIST AND ELEMENTLISTDEF WITH SINGLELISTWRITEITERATOR	129
EXAMPLE 32: ENCODING FIELDLIST WITH FIELDLISTWRITEITERATOR	130
EXAMPLE 33: ENCODING FIELDLIST WITH SINGLEWRITEITERATOR	130
EXAMPLE 34: ENCODING FIELDLIST AND FIELDLISTDEF WITH FIELDLISTWRITEITERATOR	131

EXAMPLE 35: ENCODING FIELDLIST AND FIELDLISTDEF WITH SINGLEWRITEITERATOR	131
EXAMPLE 36: ENCODING FILTERLIST WITH FILTERLISTWRITEITERATOR	132
EXAMPLE 37: ENCODING FILTERLIST WITH SINGLEWRITEITERATOR	132
EXAMPLE 38: ENCODING MAP'S DATA DEFINITION WITH DATADEFWRITEITERATOR	133
EXAMPLE 39: ENCODING MAP'S DATA DEFINITION WITH SINGLEWRITEITERATOR	133
EXAMPLE 40: ENCODING MAP'S SUMMARYDATA WITH FIELDLISTWRITEITERATOR	134
EXAMPLE 41: ENCODING MAP'S SUMMARYDATA WITH SINGLEWRITEITERATOR	134
EXAMPLE 42: ENCODING MAP WITH MAPWRITEITERATOR	135
EXAMPLE 43: ENCODING MAP WITH SINGLEWRITEITERATOR	135
EXAMPLE 44: ENCODING SERIES WITH SERIESWRITEITERATOR	136
EXAMPLE 45: ENCODING SERIES WITH SINGLEWRITEITERATOR	136
EXAMPLE 46: ENCODING VECTOR WITH VECTORWRITEITERATOR	137
EXAMPLE 47: ENCODING VECTOR WITH SINGLEWRITEITERATOR	137
EXAMPLE 48: USE FOREACH LOOP TO ITERATE THROUGH FIELDENTRY IN FIELDLIST	138
EXAMPLE 49: DATA TYPE IDENTIFYING	141
EXAMPLE 50: DECODING ELEMENTLISTDEF	141
EXAMPLE 51: DECODING FIELDLISTDEF	142
EXAMPLE 52: DECODING DATABUFFER	142
EXAMPLE 53: DECODING DATABUFFER STRING TYPE	143
EXAMPLE 54: DECODING DATABUFFER ANSI, XML, OPAQUE TYPE	143
EXAMPLE 55: DECODING DATABUFFER TIME TYPE	143
EXAMPLE 56: DECODING ARRAY WITH ARRAYREADITERATOR	144
EXAMPLE 57: DECODING ARRAY WITH SINGLEREADITERATOR	144
EXAMPLE 58: DECODING ARRAYENTRY	144
EXAMPLE 59: DECODING ELEMENTLIST WITH ELEMENTLISTREADITERATOR	145
EXAMPLE 60: DECODING ELEMENTLIST WITH SINGLEWRITEITERATOR	145
EXAMPLE 61: DECODING DATADEF WITH ELEMENTLISTREADITERATOR	146
EXAMPLE 62: DECODING DATADEF WITH SINGLEREADITERATOR FOR ELEMENTLIST	146
EXAMPLE 63: DECODING ELEMENTENTRY	146
EXAMPLE 64: DECODING FIELDLIST WITH FIELDLISTREADITERATOR	147
EXAMPLE 65: DECODING FIELDLIST WITH SINGLEREADITERATOR	147
EXAMPLE 66: DECODING FIELDLIST WITH ITERATORMASK BY FIELDLISTREADITERATOR	148
EXAMPLE 67: DECODING FIELDLIST WITH ITERATORMASK BY SINGLEREADITERATOR	148
EXAMPLE 68: DECODING FIELDENTRY	149
EXAMPLE 69: DECODING FILTERLIST WITH FILTERLISTREADITERATOR	150
EXAMPLE 70: DECODING FILTERLIST WITH SINGLEREADITERATOR	150

EXAMPLE 71: DECODING FILTERENTRY	150
EXAMPLE 72: DECODING MAP WITH MAPREADITERATOR	151
EXAMPLE 73: DECODING MAP WITH SINGLEREADITERATOR	151
EXAMPLE 74: DECODING MAP'S DATADEF WITH DATADEFREADITERATOR	152
EXAMPLE 75: DECODING MAP'S DATADEF WITH LIST OF DATADEF	152
EXAMPLE 76: DECODING SUMMARY DATA	152
EXAMPLE 77: DECODING KEY DATA	153
EXAMPLE 78: DECODING MAPENTRY	153
EXAMPLE 79: DECODING SERIES WITH SERIESREADITERATOR	154
EXAMPLE 80: DECODING SERIES WITH SINGLEREADITERATOR	154
EXAMPLE 81: DECODING SERIESENTRY	155
EXAMPLE 82: DECODING VECTOR WITH VECTORREADITERATOR	155
EXAMPLE 83: DECODING VECTOR WITH SINGLEREADITERATOR	155
EXAMPLE 84: DECODING VECTORENTRY	156
EXAMPLE 85: CHECKING HINT MASK	176
EXAMPLE 86: SETTING INDICATIONMASK	177
EXAMPLE 87: ENCODING REQUEST MESSAGE	187
EXAMPLE 88: ENCODING REQUEST MESSAGE FOR VIEW	190
EXAMPLE 89: ENCODING REQUEST MESSAGE FOR BATCH	192
EXAMPLE 90: ENCODING REQUEST MESSAGE WITH DATASTREAMS FOR SYMBOLLIST ENHANCEMENT	193
EXAMPLE 91: ENCODING RESPONSE MESSAGE	195
EXAMPLE 92: ENCODING POST MESSAGE	197
EXAMPLE 93: CREATING THE ACK MESSAGE	198
EXAMPLE 94: CREATING GENERIC MESSAGE FOR CONSUMER	199
EXAMPLE 95: VALIDATING A LOGIN MESSAGE	200
EXAMPLE 96: USAGE OF MASKS	202
EXAMPLE 97: DECODING MESSAGE TYPE	202
EXAMPLE 98: DECODING REQUEST MESSAGE	203
EXAMPLE 99: DECODING RESPONSE MESSAGE	205
EXAMPLE 100: PROCESSING BATCH ITEM RESPONSE	207
EXAMPLE 101: DECODING POST MESSAGE	210
EXAMPLE 102: DECODING ACK MESSAGE	211
EXAMPLE 103: DECODING GENERIC MESSAGE IN A CONSUMER	212
EXAMPLE 104: DECODING GENERIC MESSAGE IN A PROVIDER	214
EXAMPLE 105: REGISTRY FILE FOR THE SAMPLE CONFIGURATION	221
EXAMPLE 106: CONFIGURATION FLAT FILE FORMATATABASE	223

EXAMPLE 107: INITIALIZE RFA CONFIG DATABASE	226
EXAMPLE 108: SHUTDOWN CONFIGURATION	227
EXAMPLE 109: RETRIEVE ROOT CONFIG TREE	228
EXAMPLE 110: RETRIEVE CONFIG NODE WITH KNOWN TYPE	228
EXAMPLE 111: RETRIEVE CONFIG NODE WITH UNKNOWN TYPE	229
EXAMPLE 112: ITERATE OVER CONFIG TREE	230
EXAMPLE 113: ITERATE OVER CONFIG TREE BY USING FOREACH LOOP	230
EXAMPLE 114: POPULATING AND QUERYING A CONFIG DATABASE	233
EXAMPLE 115: MERGING MULTIPLE NAMESPACES FROM A WINDOWS REGISTRY CONFIGURATION	235
EXAMPLE 116: ACQUIRE SESSION	244
EXAMPLE 117: RELEASE SESSION	244
EXAMPLE 118: REGISTER OMMCONNECTIONEVENT	245
EXAMPLE 119: CREATE EVENT SOURCE	246
EXAMPLE 120: SUBMIT GENERIC MESSAGE	247
EXAMPLE 121: REISSUE THE INTEREST SPECIFICATION	247
EXAMPLE 122: UNREGISTER EVENT	248
EXAMPLE 123: DESTROY EVENT SOURCE	248
EXAMPLE 124: INITIALIZE EVENT SOURCE	251
EXAMPLE 125: SUBMIT RESPONSE MESSAGE	251
EXAMPLE 126: SUBMIT GENERIC MESSAGE	252
EXAMPLE 127: SUBMIT ACK MESSAGE	252
EXAMPLE 128: UNREGISTER EVENT	252
EXAMPLE 129: DESTROY EVENT SOURCE	253
EXAMPLE 130: INITIALIZE EVENT SOURCE	254
EXAMPLE 131: SUBMIT RESPONSE MESSAGE	255
EXAMPLE 132: SUBMIT GENERIC MESSAGE	255
EXAMPLE 133: UNREGISTER EVENT	256
EXAMPLE 134: DESTROY EVENT SOURCE	256
EXAMPLE 135: IMPLEMENT LOGMSGMAP.GETMSG() METHOD	258
EXAMPLE 136: INITIALIZE APPLICATION LOGGER	262
EXAMPLE 137: SHUTDOWN APPLICATION LOGGER	262
EXAMPLE 138: INITIALIZE APPLICATION LOGGER MONITOR	264
EXAMPLE 139: APPLICATION LOGGER MONITOR CLIENT	264
EXAMPLE 140: REGISTER FOR APPLICATION LOGGER EVENTS	265
EXAMPLE 141: UNREGISTER FOR APPLICATION LOGGER EVENTS	266
EXAMPLE 142: SHUTDOWN APPLICATION LOGGER MONITOR	266

EXAMPLE 143: INITIALIZE COMPONENT LOGGER	268
EXAMPLE 144: SUBMIT LOG EVENTS	268
EXAMPLE 145: SHUTDOWN COMPONENT LOGGER	269
EXAMPLE 146: FORMATTING IN A MESSAGE COMPILER (.MC) FILE	271
EXAMPLE 147: SUBMITTING A LOG EVENT	272
EXAMPLE 148: USING A LOGMSGMAP	272
EXAMPLE 149: ACQUIRE SESSION	276
EXAMPLE 150: CREATE EVENT SOURCE	276
EXAMPLE 151: CREATE EVENT QUEUE	277
EXAMPLE 152: CREATE LOGIN REQUEST MESSAGE	279
EXAMPLE 153: CREATE DIRECTORY REQUEST MESSAGE	280
EXAMPLE 154: CREATE MARKET PRICE REQUEST MESSAGE	281
EXAMPLE 155: SET MESSAGE MODEL TYPE FOR GENERIC MESSAGE	281
EXAMPLE 156: REGISTER ITEM EVENT	282
EXAMPLE 157: REGISTER CONNECTION EVENT	282
EXAMPLE 158: REGISTER ERROR EVENT	284
EXAMPLE 159: DISPATCH EVENT QUEUE	284
EXAMPLE 160: PROCESS ITEM EVENT CALLBACK	285
EXAMPLE 161: PROCESS MESSAGE FROM ITEM EVENT CALLBACK	286
EXAMPLE 162: PROCESS MESSAGE FROM CONNECTION EVENT CALLBACK	287
EXAMPLE 163: PROCESS MESSAGE FROM ERROR EVENT CALLBACK	288
EXAMPLE 164: PROCESS LOGIN RESPONSE MESSAGE	289
EXAMPLE 165: PROCESS DIRECTORY RESPONSE MESSAGE	290
EXAMPLE 166: PROCESS MARKET INFORMATION RESPONSE MESSAGE	291
EXAMPLE 167: PROCESS GENERIC MESSAGE	291
EXAMPLE 168: PROCESS ACK MESSAGE	291
EXAMPLE 169: CREATE GENERIC MESSAGE	293
EXAMPLE 170: SUBMIT GENERIC MESSAGE	293
EXAMPLE 171: CREATE POST MESSAGE	295
EXAMPLE 172: SUBMIT POSTING MESSAGE	295
EXAMPLE 173: UNREGISTER CONNECTION EVENT	296
EXAMPLE 174: UNREGISTER MARKET INFORMATION EVENT	297
EXAMPLE 175: DEACTIVATE EVENT QUEUE	298
EXAMPLE 176: DESTROY EVENT QUEUE	298
EXAMPLE 177: DESTROY EVENT SOURCE	299
EXAMPLE 178: RELEASE SESSION	299

EXAMPLE 179: INITIALIZE SESSION	302
EXAMPLE 180: INITIALIZE EVENT SOURCE	302
EXAMPLE 181: INITIALIZE EVENT QUEUE	303
EXAMPLE 182: REGISTER FOR OMMLISTENERCONNECTIONINTSPEC, OMMCLIENTSESSIONLISTENERSPEC AND OMMERRORINTSPEC	306
EXAMPLE 183: REGISTER FOR DIFFERENT LISTENER NAMES OF TWO OMMPROVIDER	307
EXAMPLE 184: ACCEPT A CLIENT SESSION	307
EXAMPLE 185: DISPATCH EVENT QUEUE	308
EXAMPLE 186: HANDLING PROCESSEVENT()	310
EXAMPLE 187: ACCEPT ACTIVE CLIENT SESSION EVENT	311
EXAMPLE 188: REJECT ACTIVE CLIENT SESSION EVENT	312
EXAMPLE 189: PROCESS INACTIVE CLIENT SESSION EVENT	312
EXAMPLE 190: PROCESS OMMSOLICITEDITEMEVENT FOR REQUEST MESSAGES	314
EXAMPLE 191: PROCESS OMMSOLICITEDITEMEVENT FOR GENERIC MESSAGES	316
EXAMPLE 192: PROCESS OMMSOLICITEDITEMEVENT FOR POST MESSAGE	317
EXAMPLE 193: HANDLING CMDERROR EVENTS	317
EXAMPLE 194: HANDLING CONNECTION EVENTS	318
EXAMPLE 195: SUBMIT RESPONSE MESSAGE	321
EXAMPLE 196: SUBMIT GENERIC MESSAGE	322
EXAMPLE 197: SUBMIT ACK MESSAGE	324
EXAMPLE 198: CLOSING ITEMS	324
EXAMPLE 199: DISCONNECTING A CLIENT SESSION	325
EXAMPLE 200: CLOSING A CLIENT SESSION	326
EXAMPLE 201: DISABLING A CLIENT SESSION REQUEST	326
EXAMPLE 202: CLEANING UP RFA RESOURCES	328
EXAMPLE 203: REGISTER FOR OMMCONNECTIONINTSPEC, OMMERRORINTSPEC AND OMMITEMINTSPEC	332
EXAMPLE 204: HANDLING PROCESSEVENT()	333
EXAMPLE 205: HANDLING LOGIN EVENTS	336
EXAMPLE 206: HANDLING CMDERROR EVENTS	336
EXAMPLE 207: HANDLING CONNECTION EVENTS	337
EXAMPLE 208: SUBMIT RESPONSE MESSAGE	340
EXAMPLE 209: SUBMIT GENERIC MESSAGE	341
EXAMPLE 210: LOGGING OUT	342
EXAMPLE 211: SENDING A BATCH REISSUE	344
EXAMPLE 212: SENDING A BATCH CLOSE	344
EXAMPLE 213: TWO SERVICE GROUP CONFIGURATION EXAMPLE	350

EXAMPLE 214: PAUSE ALL AND RESUME ALL	356
EXAMPLE 215: WARM STANDBY CONFIGURATION EXAMPLE	357
EXAMPLE 216: CONFIGURATION FOR ENABLE HORIZONTAL SCALING IN RSSL_CONS_ADAPTER	361
EXAMPLE 217: CONFIGURATION FOR ENABLE HORIZONTAL SCALING IN RSSL_PROV_ADAPTER	363
EXAMPLE 218: LOADING DICTIONARY FROM FILE	365
EXAMPLE 219: LOADING DICTIONARY FROM FILE	366
EXAMPLE 220: DICTIONARY FOR NOTIFYING CONSUMER	366
EXAMPLE 221: DICTIONARY HANDLING IN PROVIDER	367
EXAMPLE 222: AUTHORIZATION TOKEN ENCODING EXAMPLE	370
EXAMPLE 223: CONFIGURATION FOR RENAME SERVICE	383
EXAMPLE 224: UNEXPECTED ERROR IN C++/CLI	387
EXAMPLE 225: SETTING REFERENCE TYPE	388
EXAMPLE 226: SETTING REFERENCE TYPE WHICH LEADS UNEXPECTED RESULT	388

List of Figures

FIGURE 1: SYSTEM VIEW DIAGRAM NOTATION	4
FIGURE 2: RFA – CORE DIAGRAM	14
FIGURE 3: ETA – CORE DIAGRAM	15
FIGURE 4: OMM-BASED PRODUCT OFFERINGS	15
FIGURE 5: RTDS INFRASTRUCTURE	19
FIGURE 6: RFA.NET AS A CONSUMER	20
FIGURE 7: REQUEST/RESPONSE	21
FIGURE 8: BATCH REQUEST	22
FIGURE 9: VIEW REQUEST	23
FIGURE 10: SYMBOL LIST: BASIC SCENARIO	24
FIGURE 11: SYMBOL LIST: WILDCARDING WITH AN ADS CACHE	24
FIGURE 12: SYMBOL LIST REQUEST FOR NAMED DATA	25
FIGURE 13: POSTING INTO A CACHE	28
FIGURE 14: PRIVATE STREAM SCENERIOS	29
FIGURE 15: PROVIDER ACCESS POINT	30
FIGURE 16: INTERACTIVE PROVIDER	31
FIGURE 17: NON-INTERACTIVE PROVIDER	32
FIGURE 18: PUBLISHING TO MULTIPLE ADH CACHES	33
FIGURE 19: HYBRID APPLICATION	34
FIGURE 20: TYPICAL COMPONENTS IN AN RTDS NETWORK	46
FIGURE 21: REFINITIV REAL-TIME ADVANCED DISTRIBUTION SERVER	47
FIGURE 22: REFINITIV REAL-TIME ADVANCED DISTRIBUTION HUB	47
FIGURE 23: REFINITIV REAL-TIME ADVANCED TRANSFORMATION SERVER	48
FIGURE 24: REFINITIV REAL-TIME	49
FIGURE 25: REFINITIV DATA FEED DIRECT (RDF DIRECT)	50
FIGURE 26: DIRECT-CONNECT	51
FIGURE 27: INTERNET CONNECTIVITY	52
FIGURE 28: SESSION LAYER’S USAGE OF THE EVENT DISTRIBUTION MECHANISM	56
FIGURE 29: GENERAL EVENT DISTRIBUTION	59
FIGURE 30: EVENT DISTRIBUTION: EVENT QUEUE GROUP	63
FIGURE 31: NOTIFICATION CLIENT	66
FIGURE 32: HIGH/LOW THRESHOLD BEHAVIOR	69
FIGURE 33: HIGH-LEVEL DATA RELATIONSHIPS	82
FIGURE 34: COMPOSITE PATTERN	84

FIGURE 35: LOGICAL COMPOSITE SUBSYSTEM	84
FIGURE 36: ARRAY STRUCTURE	99
FIGURE 37: FIELDLIST STRUCTURE	100
FIGURE 38: ELEMENTLIST STRUCTURE	102
FIGURE 39: SERIES STRUCTURE	104
FIGURE 40: VECTOR STRUCTURE	106
FIGURE 41: MAP STRUCTURE	109
FIGURE 42: FILTERLIST STRUCTURE	112
FIGURE 43: DEFINED DATA DEFINITIONS	114
FIGURE 44: ELEMENTLISTDEF STRUCTURE	115
FIGURE 45: FIELDLISTDEF STRUCTURE	116
FIGURE 46: USING CONTAINER-SPECIFIC WRITE ITERATOR TO ENCODE DATA	123
FIGURE 47: USING SINGLEWRITEITERATOR TO ENCODE DATA	123
FIGURE 48: DECODING DATA WITH CONTAINER-SPECIFIC READ ITERATOR	139
FIGURE 49: DECODING DATA WITH SINGLEREADITERATOR	139
FIGURE 50: STRUCTURE OF MESSAGE	159
FIGURE 51: ATTRIBINFO	160
FIGURE 52: MANIFEST	161
FIGURE 53: PAYLOAD162	
FIGURE 54: REQUEST MESSAGE STRUCTURE	163
FIGURE 55: RESPONSE MESSAGE STRUCTURE	165
FIGURE 56: POST MESSAGE STRUCTURE	167
FIGURE 57: ACK MESSAGE STRUCTURE	169
FIGURE 58: GENERIC MESSAGE STRUCTURE	171
FIGURE 59: ITEM GROUP EXAMPLE	180
FIGURE 60: OMM ITEM STATE DIAGRAM	182
FIGURE 61: MESSAGE FLOW BETWEEN SENDER AND RECEIVER	185
FIGURE 62: ENCODING PROCESS	186
FIGURE 63: DECODING PROCESS	201
FIGURE 64: CONFIGURATION DATABASE AND STAGING CONFIGURATION DATABASE	216
FIGURE 65: TERMS AND CONCEPTS OF THE CONFIG PACKAGE BY EXAMPLE	218
FIGURE 66: RFA CONFIGURATION DATABASE AND NAMESPACE	219
FIGURE 67: RFA CONFIGURATION DATABASE LAYOUT	219
FIGURE 68: CONFIG DATABASE: POPULATE CONFIGURATION	224
FIGURE 69: CONFIG DATABASE: CONFIGURATION RESPOSITORY EXAMPLE	225
FIGURE 70: CONSUMER AND PROVIDER APPLICATIONS USING A SESSION	238

FIGURE 71: LOGGER PACKAGE	258
FIGURE 72: LOGGER PACKAGE CONFIGURATION	259
FIGURE 73: APPLICATION LOGGER	261
FIGURE 74: APPLICATION LOGGER MONITOR: REGISTER CLIENT	263
FIGURE 75: COMPONENT LOGGER LIFECYCLE	267
FIGURE 76: MESSAGE COMPILER BUILD PROCESS	270
FIGURE 77: RFA CONFIGURATION DATABASE “DEFAULT” NAMESPACE	273
FIGURE 78: THE LIFECYCLE OF STARTERCONSUMER EXAMPLE	274
FIGURE 79: OMM CONSUMER	275
FIGURE 80: INITIALIZATION OMM CONSUMER	276
FIGURE 81: PROCESS FOR CREATING REQUEST MESSAGE	278
FIGURE 82: REGISTERING FOR EVENTS FROM OMM CONSUMER	282
FIGURE 83: EVENT PROCESSING	285
FIGURE 84: PROCESS FOR SENDING MESSAGE USING OMM CONSUMER	292
FIGURE 85: UNREGISTERING EVENTS FROM OMM CONSUMER	296
FIGURE 86: CLEANING UP	298
FIGURE 87: STARTERPROVIDER_INTERACTIVE LIFE CYCLE	300
FIGURE 88: STARTERPROVIDER-NONINTERACTIVE LIFE CYCLE	300
FIGURE 89: HIGH LEVEL ACTIVITY DIAGRAM OF IMPLEMENTING INTERACTIVE PROVIDER	301
FIGURE 90: INITIALIZATION OMM PROVIDER	302
FIGURE 91: REGISTERING EVENTS FOR INTERACTIVE PROVIDER	304
FIGURE 92: THE EVENT PROCESSING FOR INTERACTIVE PROVIDER	309
FIGURE 93: RELATIONSHIP BETWEEN PROCESSING OMMSOLICITEDITEMEVENT AND SENDING MESSAGES FOR INTERACTIVE PROVIDER	319
FIGURE 94: UNREGISTERING PROCESSES FOR INTERACTIVE PROVIDER	324
FIGURE 95: CLEANING UP OMM PROVIDER	326
FIGURE 96: HIGH LEVEL ACTIVITY DIAGRAM FOR IMPLEMENTING NON-INTERACTIVE PROVIDER	329
FIGURE 97: REGISTERING EVENTS FOR NON-INTERACTIVE PROVIDER	330
FIGURE 98: PROCESSING EVENTS FOR NON-INTERACTIVE PROVIDER	332
FIGURE 99: SENDING MESSAGES FOR NON-INTERACTIVE PROVIDER	338
FIGURE 100: UNREGISTERING PROCESSES FOR NON-INTERACTIVE PROVIDER	342
FIGURE 101: BATCH REQUEST, REISSUE AND CLOSE	343
FIGURE 102: PAUSE AND RESUME MESSAGE FLOW (II = INITIALIMAGEFLAG, IAR= INTERESTAFTERREFRESHFLAG)	355
FIGURE 103: ORDER OF EVENTS WHEN WARM STANDBY IS ACTIVE	358
FIGURE 104: HORIZONTAL SCALING APPLICATION	362
FIGURE 105: SINGLE THREADED WITH NO EVENT QUEUE (CALLBACK MODEL)	374

FIGURE 106: SINGLE THREADED WITH AN EVENT QUEUE (CLIENT MODEL)	375
FIGURE 107: FULL DUAL-THREADED CONFIGURATION (CLIENT MODEL)	376
FIGURE 108: DUAL-THREADED WITH NO EVENT QUEUE (CALLBACK MODEL)	376
FIGURE 109: MULTIPLE EVENT QUEUES (CLIENT MODEL)	377
FIGURE 110: HORIZONTAL SCALING (CLIENT MODEL)	378
FIGURE 111: DIRECT WRITE MODEL	379
FIGURE 112: TYPICAL CONSUMER APPLICATION DESIGN	389
FIGURE 113: COMPLEX CONSUMER APPLICATION DESIGN	390
FIGURE 114: TYPICAL PROVIDER APPLICATION DESIGN	391
FIGURE 115: HYBRID APPLICATION DESIGN	392

List of Tables

TABLE 1: CHAPTER DESCRIPTIONS	3
TABLE 2: ACRONYMS AND ABBREVIATIONS	6
TABLE 3: FEATURES OF RFA .NET	11
TABLE 4: PACKAGE OVERVIEW	13
TABLE 5: PERFORMANCE COMPARISON	16
TABLE 6: CAPABILITIES BY API	17
TABLE 7: LAYER-SPECIFIC CAPABILITIES	18
TABLE 8: OVERVIEW OF REFINITIV DOMAIN MODELS (RDM)	38
TABLE 9: RFA PRIMITIVE TYPES	41
TABLE 10: RFA CONTAINER TYPES	43
TABLE 11: PROTOCOL TYPES	44
TABLE 12: REFINITIV REAL-TIME DISTRIBUTION SYSTEM COMPONENTS	44
TABLE 13: REFINITIV REAL-TIME DISTRIBUTION SYSTEM DATA FORMATS	45
TABLE 14: INFRASTRUCTURE COMPONENT AND TYPE OF DATA DICTIONARY SUPPORTED	45
TABLE 15: EVENT DISTRIBUTION MODEL CONCEPTS	54
TABLE 16: COMMON PACKAGE CONCEPTS	57
TABLE 17: EXAMPLE QUALITY OF SERVICE REQUEST	79
TABLE 18: QUALITY OF SERVICE REQUEST DEFAULTS	80
TABLE 19: QUALITY OF SERVICE: REQUEST PARAMETERS VS. BEHAVIOR	81
TABLE 20: DATA PACKAGE CONCEPTS	84
TABLE 21: DATA CONTAINER UNIFORMITY	85
TABLE 22: DATA TYPE CLASSIFICATION	88
TABLE 23: DATA TYPE CONTAINMENT	89
TABLE 24: CONTAINER PROPERTIES	90
TABLE 25: ENTRY TYPE PROPERTIES	92
TABLE 26: EXPLICIT ACTIONS	93
TABLE 27: IMPLICIT ACTIONS	93
TABLE 28: TIME FILTERING BASED ON ACTION	94
TABLE 29: DATA BUFFER PROPERTIES	98
TABLE 30: ARRAY PROPERTIES	99
TABLE 31: ARRAY ENTRY PROPERTIES	99
TABLE 32: FIELD LIST PROPERTIES	101
TABLE 33: FIELD ENTRY PROPERTIES	101
TABLE 34: ELEMENT LIST PROPERTIES	103

TABLE 35: ELEMENTENTRY PROPERTIES	103
TABLE 36: SERIES PROPERTIES	105
TABLE 37: SERIESENTRY PROPERTIES	105
TABLE 38: VECTOR PROPERTIES	107
TABLE 39: VECTORENTRY PROPERTIES	108
TABLE 40: MAP PROPERTIES	110
TABLE 41: MAPENTRY PROPERTIES	111
TABLE 42: FILTERLIST PROPERTIES	113
TABLE 43: FILTERENTRY PROPERTIES	113
TABLE 44: DATADEF PROPERTIES	114
TABLE 45: ELEMENTLISTDEF PROPERTIES	115
TABLE 46: ELEMENTENTRYDEF PROPERTIES	115
TABLE 47: FIELDLISTDEF PROPERTIES	116
TABLE 48: FIELDENTRYDEF PROPERTIES	116
TABLE 49: WRITE ITERATOR INTERFACE FUNCTIONS	123
TABLE 50: READ ITERATOR FUNCTIONS	139
TABLE 51: MESSAGE PACKAGE CONCEPTS	158
TABLE 52: ATTRIBINFO PROPERTIES	161
TABLE 53: MANIFEST PROPERTIES	162
TABLE 54: REQUEST MESSAGE PROPERTIES	165
TABLE 55: RESPONSE MESSAGE PROPERTIES	167
TABLE 56: POST MESSAGE PROPERTIES	169
TABLE 57: ACK MESSAGE PROPERTIES	170
TABLE 58: GENERIC MESSAGE PROPERTIES	172
TABLE 59: COMMON MESSAGE PROPERTIES	173
TABLE 60: INTERACTIONTYPE	176
TABLE 61: STREAM AND DATA STATE DESCRIPTIONS: CONSUMER	183
TABLE 62: STREAM AND DATA STATE DESCRIPTIONS: PROVIDER	183
TABLE 63: STATUS CODE DEFINITIONS	184
TABLE 64: GENERAL CONFIGURATION PACKAGE CONCEPTS	215
TABLE 65: DATA TYPES SUPPORTED BY THE CONFIG PACKAGE	220
TABLE 66: MAPPING BETWEEN CONFIG VALUES AND WINDOWS REGISTRY DATA TYPES	221
TABLE 67: GENERAL SESSION LAYER CONCEPTS	236
TABLE 68: OMM CONCEPTS	241
TABLE 69: EVENT AND CMD	243
TABLE 70: OMMCONSUMER INTERFACE	246

TABLE 71: INTERACTIVE OMMPROVIDER INTERFACE	249
TABLE 72: NON-INTERACTIVE OMMPROVIDER INTERFACE	253
TABLE 73: LOGGER PACKAGE CONCEPTS	257
TABLE 74: LOGGING IN APPLICATIONS	260
TABLE 75: MESSAGE MODEL TYPES	281
TABLE 76: FURTHER READING: BATCH	345
TABLE 77: FURTHER READING: DYNAMIC VIEWS	346
TABLE 78: FURTHER READING: GENERIC MESSAGE	347
TABLE 79: FURTHER READING: POSTING	348
TABLE 80: FURTHER READING: PRIVATE STREAMS	349
TABLE 81: FURTHER READING: SERVICE GROUPS	350
TABLE 82: FURTHER READING: CONNECTION AND ITEM RECOVERY	351

Chapter 1 Introduction

1.1 Purpose

This Developers Guide describes application development using RFA. The purpose is to provide the concepts, typical activities, and examples in areas including access to market information, configuration, and logging.

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

1.2 Scope

This Developers Guide focuses on how to implement consumer and provider applications and concepts of the Session Layer Package, Message Package, Data Package, Configuration Package, Logger Package and Common Package. However, some usage of other packages is also included.

1.3 Audience

This Developers Guide targets .NET software programmers developing RFA-based applications for the financial marketplace. It is assumed that developers have a basic understanding of trading room infrastructure and concepts of object-oriented analysis and design.

1.4 Product Description

The Refinitiv Robust Foundation API (RFA) .NET edition provides data-neutral, thread-aware access for accessing market information, as well as some generic configuration and logging capabilities.

RFA .NET edition is a wrapper API on RFA C++ edition for clients wishing to develop applications easily on .NET programming languages and gain benefit of the entire .NET library. It provides the same functionality of RFA C++ edition on OMM interfaces. For the performance perspective, RFA .NET edition has the performance overhead for marshalling data and function calls between managed code and unmanaged code that makes its performance slightly worse than RFA C++ edition¹. The API supports connectivity to Refinitiv Real-Time, Refinitiv Data Feed Direct (RDF Direct) 1.x and the Refinitiv Real-Time Distribution System (RTDS), through OMM and optimized, hierarchical data representations. OMM provides extensible message header information and extensible payload information.

¹ It depends on the application type.

RFA's functionality is spread across the following packages:

- The **Message** Package defines the messages that flow between various applications. These messages adopt an OMM, which enables realization of new, comprehensive data models with seldom in need of changing the API or infrastructure.
- The **Data** Package encapsulates and manages comprehensive data structures capable of realizing the content portion of disparate data models. 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 **Session Layer** Package provides service-providing and service-consuming interfaces to support connectivity to RDF Direct 1.x and the RTDS. These interfaces offer a symmetric behavioral model for hybrid applications. It provides features including thread-awareness, service grouping, connecting grouping, controlled dispatching, event message routing, quality of service, entitlements, connection pooling, connection sharing, data quality notification, normalized state & events, automatic data recovery, queue grouping & prioritization, unrestricted time quantum on call-back, request timeout, request throttling, event fanout, session sharing, item group management, pending request queuing, item buffer pooling, and multithreaded implementation.
- The **Logger** Package provides the ability to log messages as well as to receive these messages for languages that do not have a standard logging interface. This functionality is used by other RFA packages, and is also available to applications that use RFA or other packages built on top of RFA. The Logger Package allows for message internationalization.
- The **Configuration** Package provides the ability to retrieve configuration information from different repositories for languages that do not have a standard configuration interface. This functionality is used by other RFA packages, and is also available to applications that use RFA or other packages built on top of RFA.
- The **Common** Package adds a few abstract interfaces to support the Message and Data Packages. It implements a common thread-safe and thread-aware Event Distribution mechanism used by the Session Layer and the Logger Packages, as well as some additional common mechanisms.
- The **RDM** Package provides Refinitiv Domain Model specific definitions that can be used by the application and Dictionary utility for decoding, storing and encoding data dictionary information.
- The **PrivateStream** Package allows RFA users to request and establish private streams between consumer and provider applications.

1.5 Using This Document

1.5.1 Organization

The material represented in this guide is divided into sections as follows:

CHAPTER	TOPIC
Chapter 1, "Introduction"	Document description
Chapter 2, "Product Description"	Brief description of RFA .NET edition
Chapter 3, "RFA Concepts"	Concepts of consumer and provider applications
Chapter 4, "System View"	RFA application in system environment
Chapter 5, "Common Package "	Common package concept and usage
Chapter 6, "Data Package"	Data package concept and usage
Chapter 7, "Message Package"	Message package concept and usage
Chapter 8, "Configuration Package"	Configuration package concept and usage
Chapter 9, "SessionLayer Package"	SessionLayer package concept and usage
Chapter 10, "Logger Package"	Logger package package concept and usage
Chapter 11, "Implementing OMM Consumer"	How to implement OMM Consumer
Chapter 12, "Implementing OMM Provider"	How to implement OMM Provider
Chapter 13, "RFA Feature in Details"	A more detailed discussion of RFA features
Chapter 14, "Performance Consideration"	A more detailed discussion of RFA performance
Chapter 15, "Deprecated Functionality"	Details about Deprecated Functionality
Appendix A, "Deployment"	How to deploy the software package to a machine
Appendix B, "Implementation"	Additional implementation details for .NET languages
Appendix C, "Application Design"	Examples of application design

Table 1: Chapter Descriptions

1.5.2 Source Code

The RFA packages include several examples in the C#, VB .NET, C++/CLI and F# programming languages including Windows Communication Foundation (WCF) , Windows Presentaion Foundation (WPF) and ASP.NET web framework. The example programs in this document do not show exception handling for purpose of brevity. See the *RFA Reference Manual .NET Edition* for exception specifications defined on particular methods.

1.6 Conventions

1.6.1 Typographic

- C# classes, methods, properties and types shown in `blue, Lucida Console` font.
- Parameters are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples(one or more lines of code) are shown in `Lucida console` font against a light grey background.

1.6.2 Notation

In the diagrams, the following notation is used:



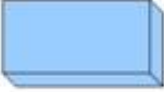

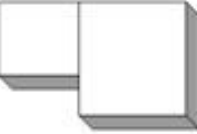



	Feed Handler, RTDS server, or other application		Network of multiple servers
	RFA application		Point-to-point connection showing direction of primary data flow
	Application with local daemon		Point-to-point connection showing direction of client connecting to server
	Multicast network		Data from external source (e.g. consolidated network or exchange)

Figure 1: System View Diagram Notation

1.6.3 Activity Diagrams

Activity diagrams follow the Unified Modeling Language (UML) standard.

1.6.4 Programming Examples

- Examples do not show inclusion of RFA namespaces.
- Source examples shown in this document are for reference only and highlight specific usage when clarification is necessary.
- Examples code shown in the document does not imply use within examples distributed with the RFA package.
- Examples favour clarity and brevity over flexibility (e.g., minimal objectification, minimal optional parameters)

1.7 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.
- 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 address.

1.8 Acronyms and Abbreviations

ACRONYM	DEFINITION
API	Application Programming Interface
ADH	Refinitiv Real-Time Advanced Distribution Hub
ADS	Refinitiv Real-Time Advanced Distribution Server
ATS	Refinitiv Real-Time Advanced Transformation Server
CBR	Criteria-Based Requests
DACS	Refinitiv Data Access Control System
DDM	Domain Message Model
EED	Refinitiv Real-Time Edge Device
ETA	Enterprise Transport API
IDN	Refinitiv Real-Time
OMM	Open Message Model
QoS	Quality of Service
EDF Direct	Refinitiv Data Feed Direct
RDM	Refinitiv Domain Model
RFA	Refinitiv Robust Foundation API
RTDS	Refinitiv Real-Time Distribution System
RSSL	Refinitiv Source Sink Library
RWF	Refinitiv Wire Format
SOA	Service-Oriented Architecture
SSL	Source Sink Library
UML	Unified Modeling Language
WCF	Windows Communication Foundation
WPF	Windows Presentation Foundation

Table 2: Acronyms and Abbreviations

1.9 References

- [1] *RFA RDM Usage Guide .NET Edition*
Describes the Refinitiv Data Models (RDM).
- [2] *RFA Reference Manual .NET Edition*
Sets of HTML pages that describe interfaces for the Session Layer, Message, Data, Logger, Config, Common, RDM and Private Stream packages.
- [3] *AnsiPage API Developers Guide .NET Edition*
Describes the concepts and usage of page-based encoding and decoding.
- [4] *AnsiPage API Reference Manual .NET Edition*
Sets of HTML pages that describe interfaces for page-based encoding and decoding.
- [5] *DACSLOCK API Developers Guide .NET Edition*
Describes the concepts and usage for managing authorization lock information.
- [6] *DACSLOCK API Reference Manual .NET Edition*
Sets of HTML pages that describe interfaces for managing authorization lock information.
- [7] *RFA Configuration Guide .NET Edition*
Explains the configuration parameters for RFA.
- [8] *TS1 API Developers Guide .NET Edition*
Describes the concepts and usage for accessing historical data.
- [9] *TS1 API Reference Manual .NET Edition*
Sets of HTML pages that describe interfaces for accessing historical data.
- [10] *RFA Value Added Components Developers Guide .NET Edition*
Describes the concepts and usage of Value Added Components which is designed to ease and speed up RFA application development.
- [11] *RFA Value Added Components Reference Manual .NET Edition*
Sets of HTML pages that describe Value Added interfaces for accessing historical data.
- [12] *RFA Migration Guide .NET Edition*
Describes migration issues from older to newer versions of RFA .NET.
- [13] *RFA Open Source Performance Tools Guide .NET Edition*
Describes the RFA Open Source Performance Tools.

Chapter 2 Product Description

2.1 What is RFA .NET API?

The RFA.NET edition is a wrapper API suite on RFA C++ edition that can access, use, create and provide market information for clients wishing to develop applications in a Microsoft .NET programming environment. It provides a robust framework for components or object-oriented applications developed in .NET framework programming languages such as C#, VB.NET, C++/CLI, F# and the ASP.NET web framework.

The API has been designed with performance in mind to reduce overhead of wrapping RFA C++ edition. It can be used for applications that require high throughput, low latency, or both. RFA has a flexible thread model that is both thread safe and thread aware. Because RFA.NET is implemented on top of RFA C++, it has a slightly performance overhead compared to the RFA C++ edition².

RFA has also been designed to connect with many different types of servers and data feeds. RFA can normalize data events through the use of 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 for how data is manipulated.

RFA .NET edition currently includes the following packages: Common, Configuration, Logger, Session Layer, OMM Message and Data, RDM, Private Stream, ANSI Page, DACS Lock, and TS1.

- The 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. A Session corresponds to a configuration context, one or more connections, and a thread context. The Session Layer Package has OMM interfaces to support the OMM provided by RDF Direct and RTDS.
- The OMM Message and Data, ANSI Page, DACS Lock, and TS1 Packages are used for encoding and decoding various data formats that can be sent and received through the Session Layer Package.
- The RDM package contains Refinitiv Domain Model specific definitions that can be used by the application.
- The Private Stream package is used with Request Message and Response Message to request and establish private streams between consumer and provider applications.

As a product, RFA.NET also includes many example programs, developer guides, and HTML-based references manuals.

² It depends on the application type.

2.2 Features of RFA

RFA offers a data delivery solution by supporting the data models known as Refinitiv Domain Model (RDM) defined by RDF Direct 1.x and Refinitiv Real-Time. It also supports User-Defined Data Models based on OMM.

RDM represents information in familiar ways and includes Level 1 and Level 2 data. RDM provides support for content not available on Refinitiv Real-Time such as Market-by-Order (Order Book) and Market-by-Price (Market Depth) and a more logical structure for some content that is available on Refinitiv Real-Time (e.g. Market Maker).

Throughout this document, OMM Data refers to RDM data sourced from Refinitiv Real-Time, RDF Direct 1.x, and the RTDS. Service consumer applications and service provider applications are also known as OMM Consumers and OMM Providers.

The following table lists the features of RFA .NET:

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.
Enhanced Symbol List	The Symbol List feature has been enhanced to allow an application to be able not only to request the symbol list, but also to optionally subscribe to all items in that symbol list on user's behalf. For more details, refer to Section 7.1.10.2.
Visible Publisher Identifier (VPI)	Originating publisher information (i.e. VPI) is available via the PublisherPrincipalIdentity parameter . This class provides both publisher ID and publisher address. This PublisherPrincipalIdentity is made available within the RespMsg and PostMsg . For more details, refer to Section 7.1.13.
Performance Enhancement for Outbound Messages at Low Update Rates	The new connection-level configuration parameter flushTimerInterval 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 connection types. For more details, refer to the <i>RFA Configuration Guide .NET Edition</i> .
Performance Enhancement for Inbound Message at Low Update Rates	The existing configuration parameter responseQueueBatchInterval has been enhanced to allow the value to be as low as 1 millisecond (previously 10 milliseconds for Unix and 15 milliseconds for Windows), resulting in lower latency for inbound messages at low update rates. RFA will attempt to flush the internal response queue no later than the configured value. This feature benefits all connection types. This does not apply if threadModel=Single For more details, refer to the <i>RFA Configuration Guide .NET Edition</i> .
More Descriptive Error Messages	Select log messages have been updated to be more descriptive.
Provide Connection Statistic Information	RFA allow users to register interest for receiving connection statistics, events for a connection, or a list of connections. Connections statistics are the number of bytes read and written on the wire provided periodically by RFA.
Value Added Admin Component and Domain Representations	The Value Added Components aim to provide an alternate and simpler entry point to leverage the features of RFA with more ease-of-use and simplicity. They are offered alongside the native RFA APIs in order to maximize the user experience by allowing developers to more easily and rapidly develop RFA applications. For more information, refer to the <i>RFA Value Added Developers Guide .NET Edition</i> and the <i>RFA Value Added Reference Manual .NET Edition</i> .
Multicast Non-Interactive Provider	This release extends the current capability of Non-Interactive Publishers. In the past, Non-Interactive Publishers would make TCP/IP connections and publish into an ADH Cache. Now clients can choose to publish each message simultaneously to multiple ADHs (ADH 2.2 or higher) across a multicast channel. With the addition of other features in ADS/ADH 2.2 as well as DACS 7.7, clients can control who and what can be published. See API, ADH/ADS, and DACS Documentation for more details.

FEATURE	DESCRIPTION
Normal to Private Stream Item-based Recall	RFA now 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. For more information, refer to Section 13.5.
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. For more information, refer to Section 13.4.
Domain Message Validation	RFA now 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. For more information, refer to Section 7.2.1.6.
RDM	RFA allows clients to consume RDM-provided data either directly from Refinitiv Real-Time, RDF Direct 1.x or third-party RFA (6.5 or later) provider application, or indirectly through the RTDS (via an ADS). Older Providers(SSL-Based) can have their data converted to RDM(Market Price) via RTDS (ADH/ADS). 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 Message and receive Response Message. The service providers receive Request Messages and send Response Message.
Hybrid Applications	RFA supports hybrid applications, which are applications that receive Request Message from a consumer and forward the same Request Message to another provider. Similarly, a hybrid application receives Response Message 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 Message 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 Message, RFA must first establish a standard stream. For more information on generic messages, refer to Section 13.3.
Private Streams	Using Private Streams, applications can privately access and exchange information. Private streams are exclusively established between two points: a consumer and a provider. In contrast with standard streams, data on a private stream (e.g., data related to transactions) is not shared with any other consumers. For more information, refer to Section 13.5.
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 Message 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). For more information, refer to Section 14.4.6.
Posting	OMM applications can use the post capability to send post messages through RTDS. A Post message is an OMM message type that can contain any OMM data container type (e.g., Map, ElementList, etc.), opaque data, or another OMM message as its payload. An OMM post enables end-to-end posting 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 Message can be single-part or multi-part messages, and, if requested, Post Message can be sent with an acknowledgement requested from the Posting provider. When using RTDS components (ADS / ADH), message translation between OMM and MarketFeed is supported. For example, posting OMM Level 1 data on RTDS can result in a conversion to MarketFeed data format. For more information, refer to Section 13.4.

FEATURE	DESCRIPTION
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 Section 5.2.1.6.</p>
Batch	<p>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 <i>RFA RDM Usage Guide .NET Edition</i> (e.g. Login, Directory, and Dictionary).</p> <p>Batch functionality also allows clients to close multiple items in a single close request message. The batch reissue feature also allows clients 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 13.1.</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 13.2.</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 13.11.1.</p>
Warm Standby	<p>Warm Standby allows failover to a standby stream in the event that 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 13.12.</p>
Connection Redirection (Load Balancing)	<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> <p>For more information, refer to Section 13.9.</p>
RDM Definitions and RDMFieldDictionary Utility	<p>RFA provides the RDMFieldDictionary utility and RDMFidDef 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 Section 13.15 and the <i>RFA Reference Manual .NET Edition</i>. The Provider_Interactive and Consumer examples illustrate how to use this utility.</p>

Table 3: Features of RFA .NET

2.3 Package Overview

RFA .NET is bundled as a set of logical packages (further discussed in respective chapters):

PACKAGE	DESCRIPTION
Common Package	<p>The Common Package provides general functionality used by other RFA packages. The bulk of this functionality is the Event Distribution Model, which is a mechanism to deliver asynchronous notifications. This mechanism is thread safe and thread aware, imposing little impact on the application's threading model.</p> <p>The Common package also provides low-level classes such as RFA_String and exception handlers.</p> <p>For more information, refer to Chapter 5, "Common Package."</p>
Data Package	<p>The Data Package provides data interfaces that manage and manipulate raw data in an extensible representation. The data structures model the primitive and container types defined by OMM, such as FieldLists, ElementLists, and Maps.</p> <p>The data structures support a variety of capabilities and wire optimizations. These include nested containers, fragmentation of "large" data models, the ability to split data definition from raw data content, and fixed-place real types. The Data Package also includes iterator interfaces to encode and decode the data.</p> <p>The interfaces in the Data Package are concrete representations of the Data interface defined in the Common namespace. An application uses these interfaces to access the OMM payload information.</p> <p>Messages containing these data structures flow between provider and consumer applications.</p> <p>For more information, refer to Chapter 6, "Data Package."</p>
Message Package	<p>The Message Package defines the OMM messages that flow between service provider and consumer applications. A Message is an abstract container comprising of a header and raw data. It provides the flexibility to create and handle new content needed by user-defined domain models.</p> <p>Interest Specifications, Events and Commands contain Messages. Applications process events that RFA has internally converted to an Event from a Message.</p> <p>For more information, refer to Chapter 7, "Message Package."</p>
Configuration Package	<p>The Configuration Package provides access and management of RFA .NET configuration parameters. Applications may populate this configuration in a number of ways, such as programing, from a flat file, or from the Windows registry.</p> <p>Applications may also use the Configuration Package to manage their own configuration parameters, so the configuration can be handled consistently between the application and RFA .NET.</p> <p>For more information, refer to Chapter 8, "Configuration Package."</p>
SessionLayer Package	<p>The Session Layer Package is the central component of RFA .NET. Its main responsibility include:</p> <ul style="list-style-type: none"> • Providing the main RFA .NET interface to the application. • Allocating sessions and connections on the application's behalf. • Controlling data exchange between RFA .NET and the application using the Event Distribution Model • Managing RFA .NET internal processing, such as routing by Quality of Service, service information, item and connection recovery, and data exchange between the session and connection layers of RFA .NET. <p>The application uses the Session Interface to acquire a concrete session, register Interest Spec, and to exchange data.</p> <p>Using the Session Layer, an application can consume (subscribe), provide (publish), and post (contribute) market information. Applications can access data across one or more thread contexts and receive asynchronous notifications in the same or different thread contexts.</p> <p>For more information, refer to Chapter 9, "SessionLayer Package."</p>
Logger Package	<p>The Logger Package provides access and management of log events. The Logger Package records log events to persistent storage, such as flat file or Windows Event Log.</p> <p>Other RFA packages use the Logger Package to submit log events. Additionally, applications may use the Logger Package to submit application-specific log events. Whether submitted by RFA packages or the application itself, applications can register to receive asynchronous notification of these log events. Applications access log events through extensions to the Event Distribution Model.</p> <p>For more information, refer to Chapter 10, "Logger Package."</p>

PACKAGE	DESCRIPTION
Private Stream Package	The Private Stream Package allows RFA users to establish a Private Stream. The IndicationMaskFlag.PrivateStream used with request and response messages to request and establish private streams between consumer and provider applications. For more information, refer to Chapter 7, "Message Package."
Other Packages	RFA also includes the DACSLOCK, AnsiPage, and TS1 packages. This document refers to these APIs where appropriate. See Section 1.9 for references to these APIs.

Table 4: Package Overview

Chapter 3 RFA Concepts

3.1 Refinitiv APIs

3.1.1 Refinitiv Robust Foundation API (RFA)

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 .NET languages, C++, or Java. Furthermore, RFA also 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.

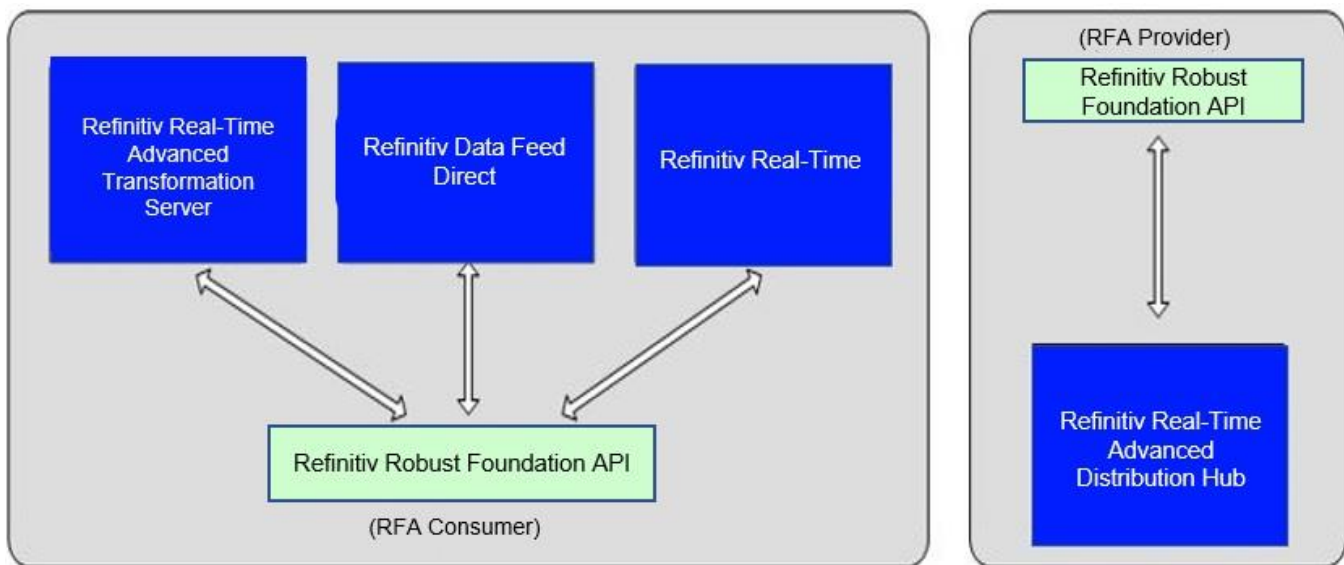


Figure 2: RFA – Core Diagram

3.1.2 Enterprise Transport API (ETA)

ETA is the customer release of Refinitiv's low-level internal API, currently used by the RTDS (and its dependent APIs) for the optimal distribution of OMM/RWF data.

ETA 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.

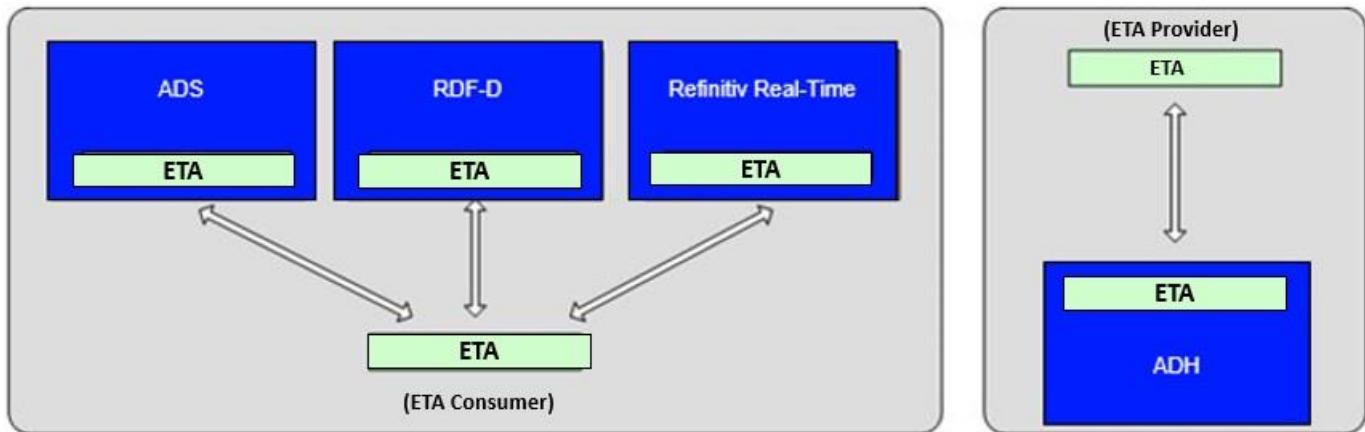


Figure 3: ETA – Core Diagram

3.1.3 A Contrast of RFA and ETA

RFA and ETA 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 RTDS. Customers can choose between a feature loaded session level API (RFA) and a high performance transport level API (ETA). RFA uses ETA as its transport layer and builds its session layer upon it.

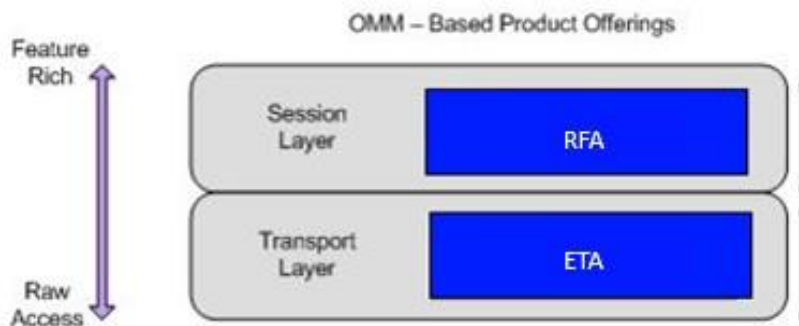


Figure 4: OMM-Based Product Offerings

3.1.4 Performance

The following table lists a high-level comparison of existing API products and their performance on OMM data. Key factors are latency, throughput, memory, and thread safety. Results may vary depending on whether the application uses watchlists or 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
ETA	Safe and aware	Very high	Lowest	Lowest
RFA C++ (OMM)	Safe and aware	High	Low	Medium (Watchlist, optional queues)
RFA .NET	Safe and aware	Medium	Medium	Medium (Watchlist, optional queues, marshalling objects)
RFAJ(OMM)	Safe and aware	Medium	Medium	Medium/High (Watchlist, optional queues)

Table 5: Performance Comparison

3.1.5 Fuctionality

To make an informed decision on which API to use, tradeoffs must be made between performance and capabilities. This section focuses on differences in capability. For details on differences in terms of performance, refer to section 3.1.4, Performance.

3.1.5.1 General Capability Comparison

CAPABILITY TYPE	CAPABILITY	RFA	ETA
Transport	Compression via OMM	X	X
	HTTP via WinInet (RWF)	X	X
	RV Multicast	X	
	TCP/IP: RWF	X	X
	TCP/IP: SSL	X	
	Reliable Multicast: RWF	X	X
	Unidirectional Shared Memory		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	ETA
	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
	DACS Lock	X	X
	OMM	X	X
	RMTES	X	
	TS1 Parser	X	

Table 6: Capabilities by API

3.1.5.2 Layer-Specific Capability Comparison

The following table lists the capabilities specific to individual session-layer (RFA) or transport-layer (ETA) . RFA uses the information provided from ETA and creates a specific implementation of a capability. Although these capabilities are not implemented by ETA , ETA clients can use the information provided by ETA to implement the same functionality (i.e., as provided by RFA) on top of ETA .

CAPABILITY	RFA	ETA
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
QoS Management	X	*
Network Pings: automatic	X	*
Recovery: connection	X	*
Recover: items	X	*
Request routing	X	*
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

Table 7: Layer-Specific Capabilities

* ETA customers can get the same functionality but must implement it themselves. RFA implements this feature.

3.2 Concepts: Consumer/Provider

The following sections provide details of general concepts regarding consumers and providers.

At a very high level, the RTDS facilitates controlled and managed interactions between many different service **providers** and **consumers**. This real-time streaming Service-Oriented Architecture (SOA) is used extensively as the middleware to integrate financial-service applications. While providers implement services and expose a certain set of capabilities (e.g. content, workflow, etc.), consumers use the capabilities offered by the service providers 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 application.

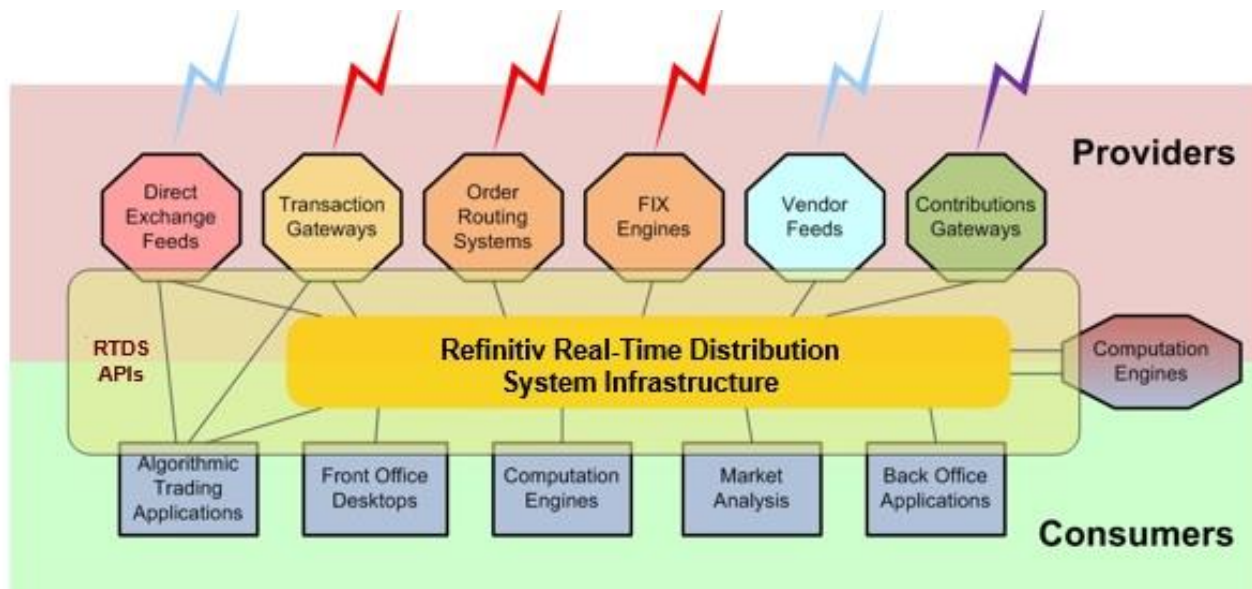


Figure 5: RTDS Infrastructure

To access needed capabilities, consumers always interact with a provider, either directly and/or via RTDS. Consumer applications that want the lowest possible latency can communicate directly, using the RTDS or Refinitiv Real-Time API's, with the needed service providers. You can realize more complex implementations (i.e., integrating multiple providers, managing local content, automated resiliency, scalability, control, and protection) by situating RTDS between provider and consumer applications.

3.2.1 Consumer

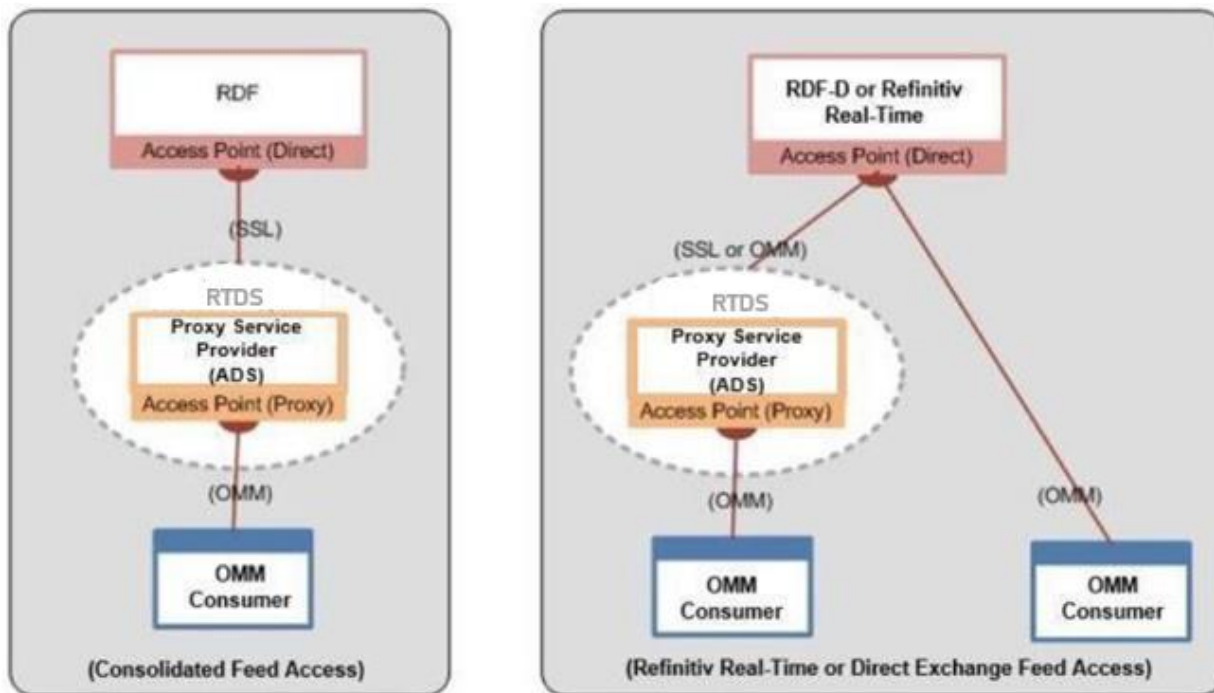


Figure 6: RFA.NET as a Consumer

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 concrete consumer access point. Such an access point is implemented by the service-provider application when it supports direct connections from consumers. In the diagram, this is illustrated by RFA connecting to Refinitiv Real-Time directly via the Access point (Direct).
- A proxy access point. Such an access point is point-to-point based and implemented by the ADS. In the diagram, this is illustrated by RFA connecting to the provider by first passing through the Access Point.

Examples of consumer are as follows:

- An application that subscribes to data via RTDS, RDF Direct, or Refinitiv Real-Time.
- An application that posts data to RTDS or Refinitiv Real-Time. This functionality includes such concepts as Contributions/Inserts or Local Publication into a cache.
- An application that communicates (sends/receives) via Generic Messages with RTDS or Refinitiv Real-Time.

3.2.1.1 Subscription: Request/Response

In order to receive data, consumer applications must successfully log in to an ADS or Refinitiv Real-Time. This establishes a Login Stream, which authenticates a consumer to subscribe and receive data information. This subscription is made to a Service, which 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 or 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.

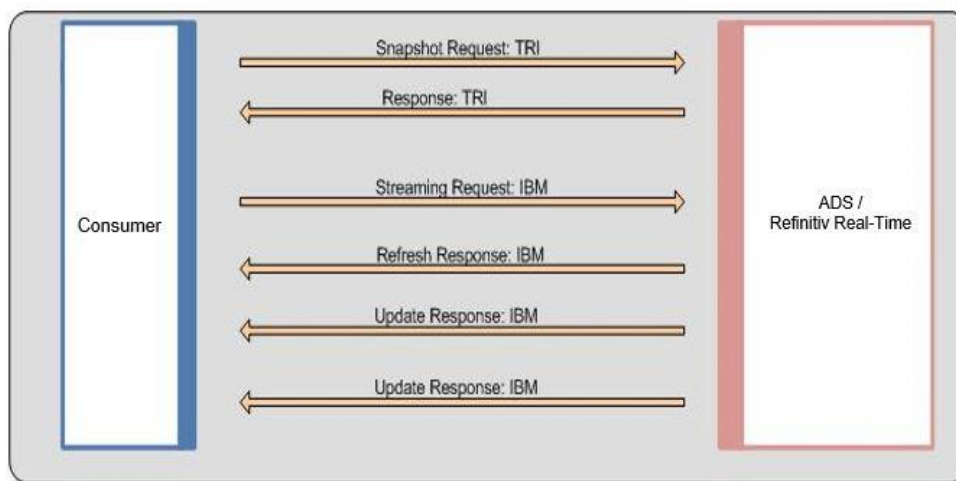


Figure 7: Request/Response

3.2.1.2 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.

3.2.1.3 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.N”, “GE.N”, and “INTC.O.”

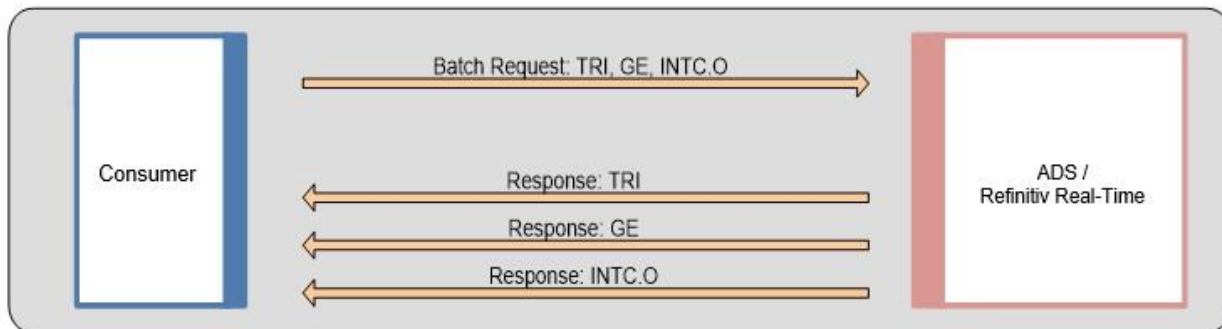


Figure 8: Batch Request

3.2.1.4 View

One method of reducing the amount of data flowing across the network is to use the Filtering capabilities of the RTDS. Filtering out certain fields uses bandwidth more efficiently by reducing network traffic. When filtering is enabled in the RTDS, 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 View. 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. View is specified per request. Multiple items can be requested each with its own view using multiple item requests.

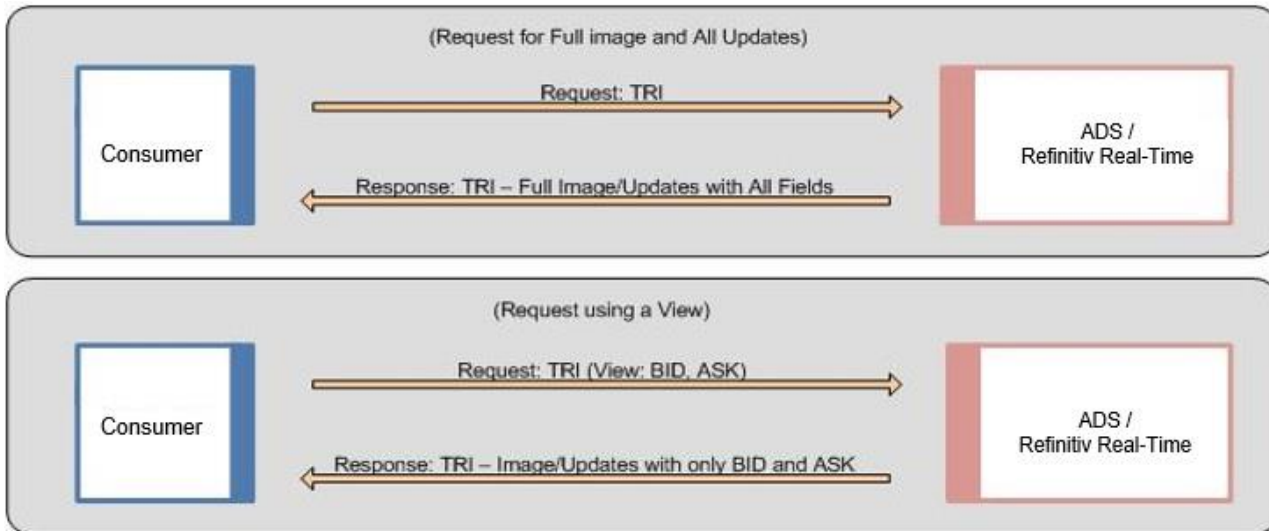


Figure 9: View Request

In conjunction with server-side filtering, view 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.

3.2.1.5 Symbol List

Clients can open many items using a Batch request; however, they must know all the items they want to open. **Symbol List** is like a reverse batch. Thus, the clients can use a Symbol List to open many items when they do not know all the items they might want. This replaces the functionality for clients that previously used the SSL 4.5 API with Criteria-Based Requests (CBR).

For example, the client may have some key name such as “FRED” and might make a Symbol List request for “FRED.” The request will flow through the platform to a Provider that is capable of resolving “FRED.” The Provider sends back a list of names that map to “FRED.” The client can then choose to open the items individually or make a batch request to open many items. The batch or individual item request is resolved by the first cache that contains the data as illustrated in the figure below.

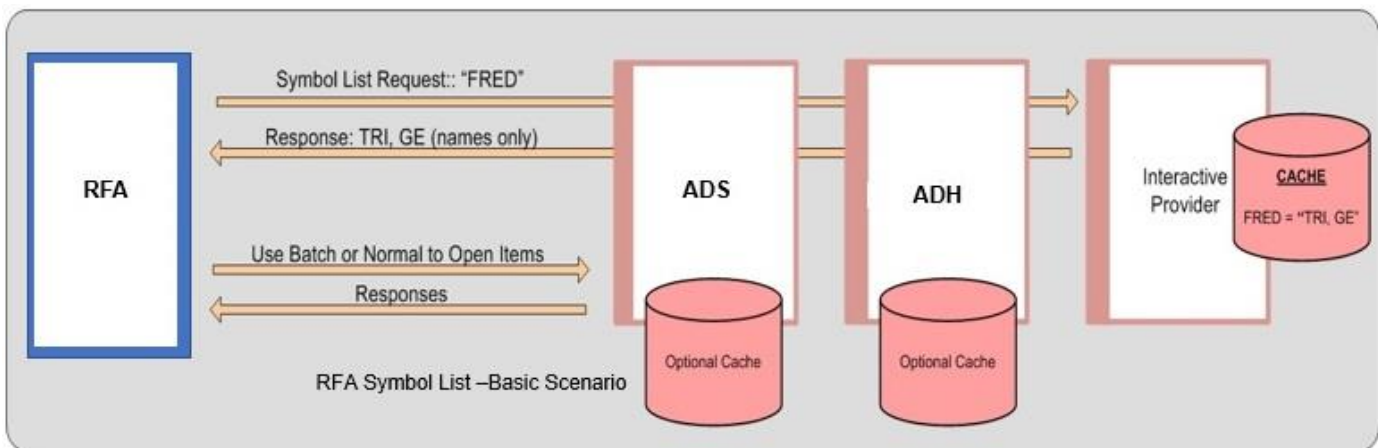


Figure 10: Symbol List: Basic Scenario

The second scenario is for clients that use the ADS Cache and want the ability to dump the cache for their OMM clients. In the diagram below, a client requests a service for the item **_ADS_CACHE_LIST**. The ADS will receive the request and respond with the names of all items in its cache. The client can then choose to open items individually or make a batch request to open multiple items.

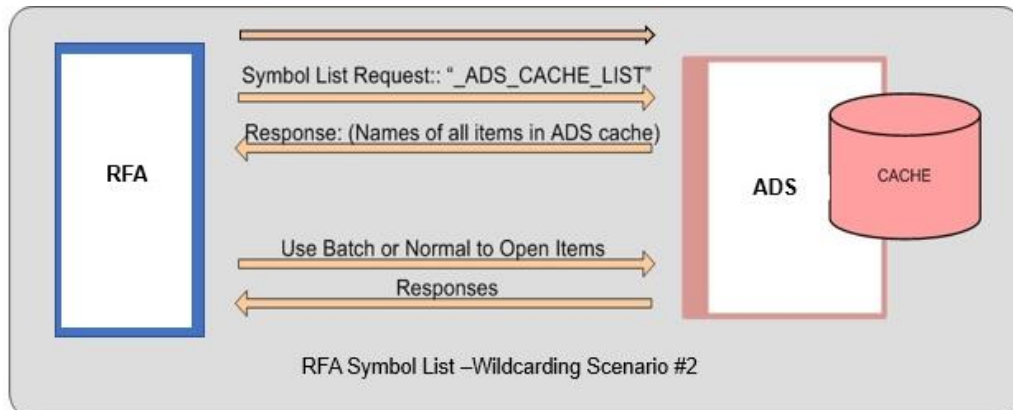


Figure 11: Symbol List: Wildcarding with an ADS Cache

3.2.1.6 Enhanced Symbol List

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

The example below shows how the application gets data along with names on the symbol list domain.

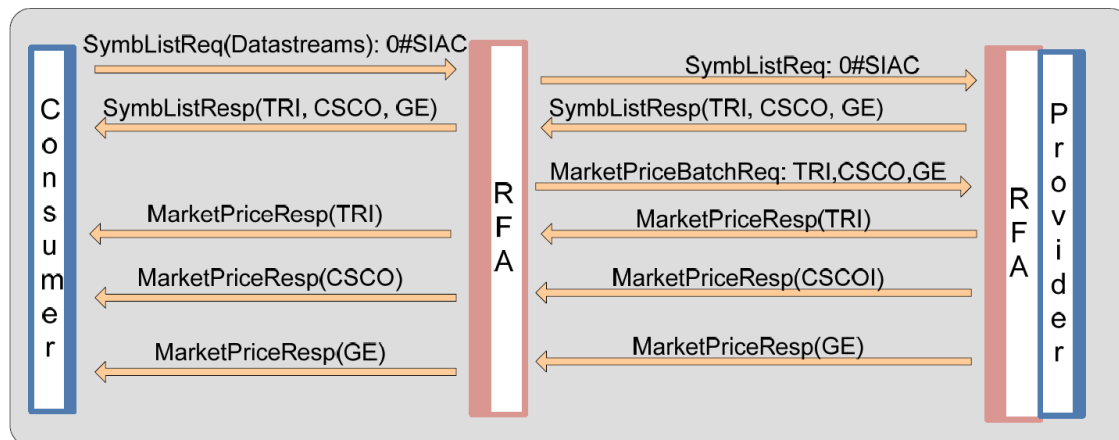


Figure 12: Symbol List Request for Named Data

3.2.1.7 Pause/Resume

The **Pause/Resume** feature optimizes network bandwidth. You can use Pause/Resume to reduce the amount of data flowing across the network for a single item or for many items that might already be openly streaming data to a client. To initiate a Pause/Resume, the client sends a request to pause an item to the ADS. The ADS receive the pause request and stops sending new data to the client for that item. 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 the new data to the client). When the client wants to start receiving messages for the item again, the client sends a resume to the ADS which then responds by sending an aggregated update or a refresh (a current image) to the client. The ADS continues to send all subsequent messages from the moment it resumes sending. By using the Pause/Resume feature a client can remove cases of multiple open/close scenarios which can disrupt the ADS. It can also create faster recovery times. There are two main use-case scenarios for this feature.

3.2.1.7.1 Pause / Resume Use Case: Backend Processing

The first use case assumes a client application performs a lot of backend processing. In this scenario, a client application has multiple items open. So many that the client is just barely able to keep up 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 is going to cut into its ability to effectively process downstream message traffic. The client does not want to back up its messages from the ADS and risk having ADS abruptly cut its connection. But neither does the client want to close its own connection (or close all the items on the ADS) which can cause the ADS to thrash, and would also require that the client re-open all items when its backend processing is done.

In this case, the client 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 refresh, or current image, to the client for all paused items and then continues to send all subsequent messages from the point it resumed.

3.2.1.7.2 Pause / Resume Use Case: Display Applications

The second use case assumes the application displays a lot of data. In this scenario, the user has two windows open. One window has item “TRI” open and is updating (Window 1). The other has “INTC.O” open and is updating (Window 2). On his screen, the user moves Window 1 to cover Window 2 and 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 refresh, or current image, to the client for the item “INTC.O” and then continues to send all subsequent messages from the point it resumed.

3.2.1.8 Posting

Through posting, consumers can easily push content into any cache within the RTDS (i.e., an HTTP POST request). Data contributions/inserts into the ATS or publishing into a cache offer similar capabilities today. When posting, consumer applications reuse their existing sessions to publish content to any cache(s) in the RTDS (i.e. service provider(s) and/or in infrastructure components).

When compared to spreadsheets or other applications, RFA posting offers a more efficient form of publishing because the application does not need to create a separate provider session or manage event streams. The posting capability, unlike unmanaged publishing or inserts, offers optional acknowledgments per posted message. The two types of posting are On-Stream and Off-Stream.

- **On-Stream Post:** Before 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 in the Core Infrastructure components.

By posting, a user can post to ADS’s local cache and also push a post up to the ADH (or beyond) if the client configures RTDS infrastructure to allow for this behavior. Such posting flexibility is a good solution if one’s applications were restricted to the LAN where the ADS resides.

Posting also supports **OMM-based contributions**. Through such posting, clients can contribute data to a device on the head-end or a custom-provider.

The following diagram helps illustrate the usefulness of posting. Green and Red services support internal posting and are fully implemented within the ADH. In both cases the ADH receives posted messages and then distributes these messages to interested consumers. In the right-side segment, the ADS component has enabled caching (for the Red service). In this case posted messages received from connected applications will be cached and distributed to these local applications before being forwarded (re-posted) up into the ADH cache. Posting to provider applications (i.e., the Purple service in this diagram), can also be done if supported by the provider.

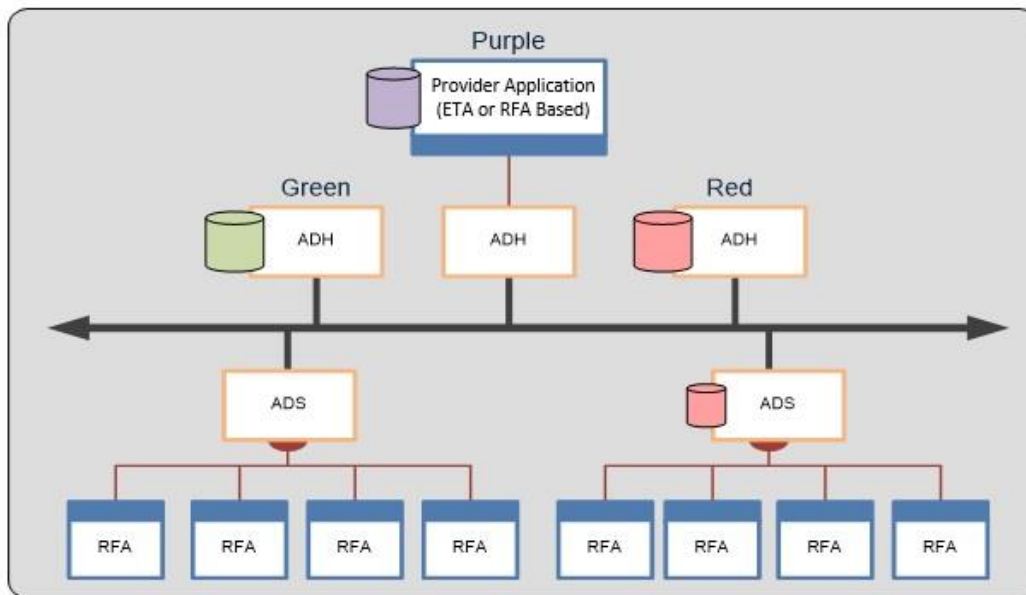


Figure 13: Posting into a Cache

3.2.1.9 Generic Message

Using a **Generic Message**, an application can send or receive a bi-directional message. A generic message can contain any OMM primitive inside it. Whereas the Request/Response type message flows from RTDS to a consumer application, a generic message can flow in any direction, and a response is not required or expected. One advantage to using generic messages is its freedom from the traditional Request/Response data flow.

In a generic message use case, the consumer sends a generic message to an ADS while the ADS also publishes a generic message to the consumer application. Moreover, the consumer can send a generic message to the provider and vice versa. Generic message behavior is not limited to Market Data-based Domains (such as MarketPrice, etc). Any domain can have this type of behavior. If a generic message is sent to an older component, one that does not understand generic messages, the component will receive the message but the message is ignored or discarded.

3.2.1.10 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 reestablished. The retry interval is configurable and defaults to 15 seconds. Once a connection is reestablished, RFA will relogin on behalf of the application using the same credentials used during the initial login. If the application is a consumer and it requested SingleOpen in the login request, RFA will also perform item recovery.

3.2.1.11 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 re-established. If the client has requested SingleOpen, then RFA will re-request all streaming items from the provider as soon as the connection is re-established. If SingleOpen is not request by the client, the client is responsible for re-requesting all items when the connection is re-established.

3.2.1.12 Warm Standby

Warm Standby allows failover to a standby connection in the event that the primary connection fails. Since 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.

3.2.1.13 Private Stream

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 connection through the RTDS, or via a cascaded set of Platforms. The following diagram illustrates these configurations.

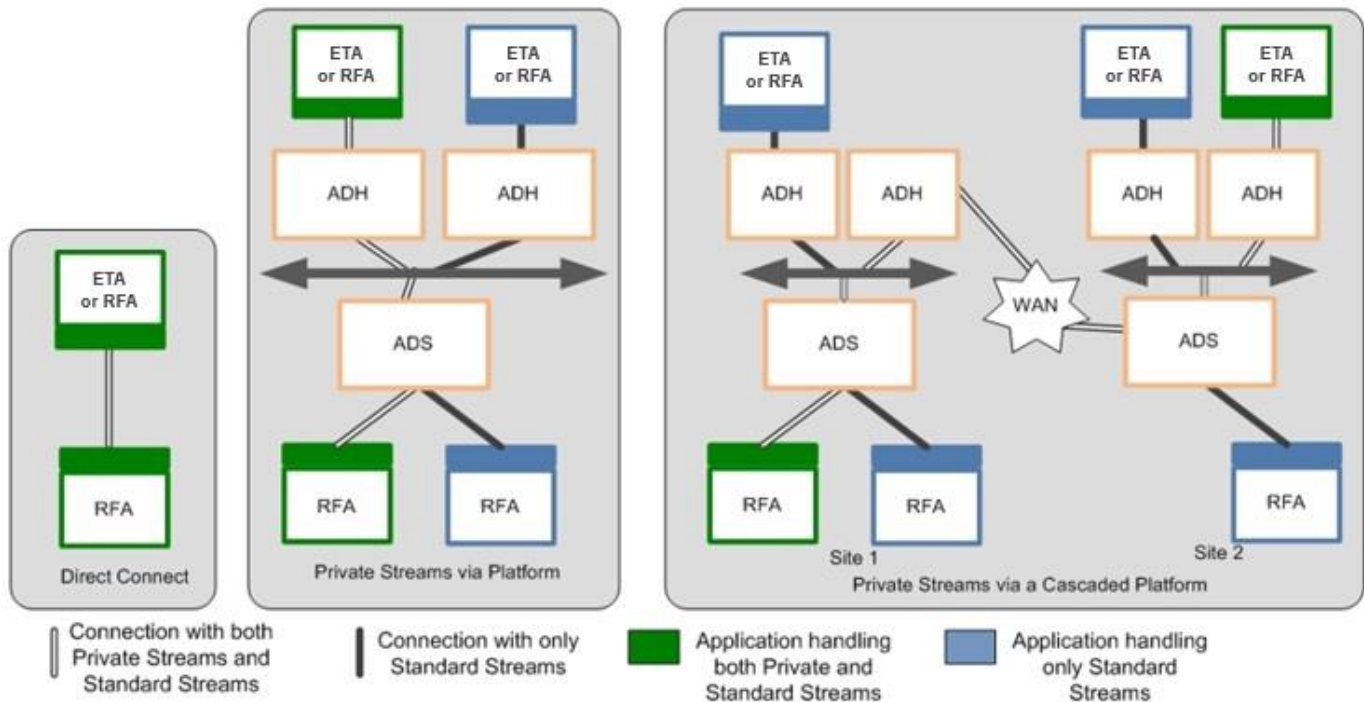


Figure 14: Private Stream Scenarios

A virtual private connection is made up of the existing individual point-to-point 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 neither fanned out by RFA or by any RTDS components nor shared with other consumers or providers.

In the diagrams above, 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. Once 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 on the Private Stream.

Any break in a “virtual connection” causes the provider and consumer to be notified of the connection loss. 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, Snapshot, and Batch requests, Post and Generic Messages, and Views on all Refinitiv Domain Models and Custom Domain Models.

3.2.1.14 Connection Redirection (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.

3.2.2 Provider

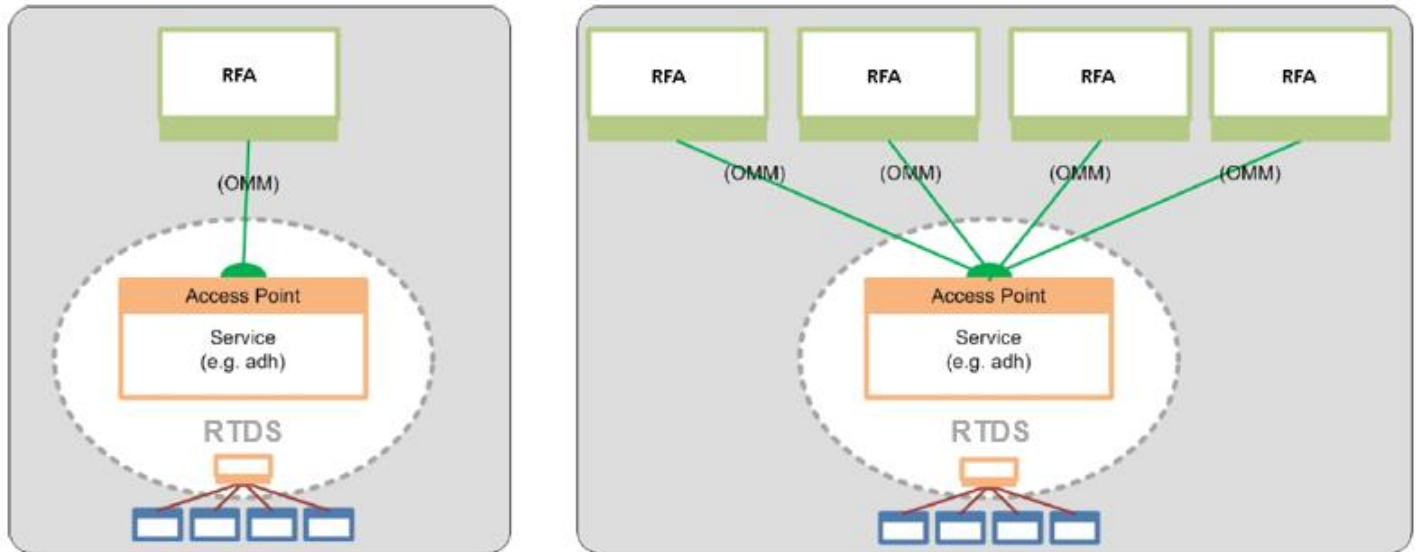


Figure 15: Provider Access Point

Providers make their services available to consumers through RTDS infrastructure components. Every Provider-based application must attach to a provider access point to interoperate with consumers. All provider access points are considered concrete/direct and implemented by a RTDS infrastructure component (like ADH).

Examples of Providers include:

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

3.2.2.1 Interactive Provider

An **Interactive provider** communicates with RTDS by accepting and managing multiple connections with RTDS components. The following diagram illustrates this concept.

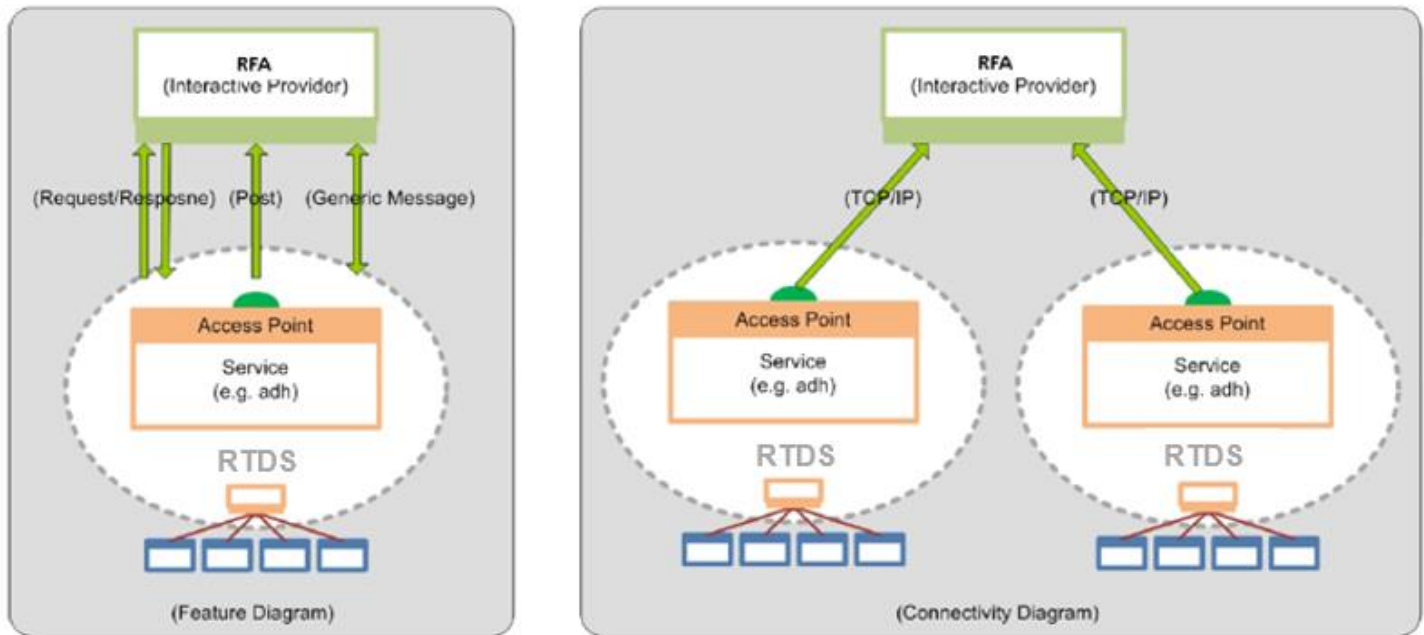


Figure 16: Interactive Provider

An interactive provider receives connection requests from RTDS. The Interactive Provider sends information as to what services, domains, and capabilities it can provide or for which it can receive requests. It then sends information about its data dictionary, describing the format of expected data types. After this is completed, its behavior is interactive. The Interactive Provider concept is similar to what was called a Sink Driven Server or Managed Server Application in legacy Refinitiv systems. Interactive Providers act like servers in a client-server relationship. An Interactive Provider can accept connections from multiple RTDS components and allow the Interactive Provider to manage those connections.

3.2.2.1.1 Request/Response

In a standard Request/Response scenario, an Interactive Provider receives requests from consumers on RTDS (e.g., "Provide data for item TRI.N"). The consumer then expects the Interactive provider to provide a response, status, and possible updates whenever the information changes. If the item cannot be provided by the Interactive provider, the consumer expects the provider to reject the request by providing an appropriate response. Request and response behaviour is not limited to Market Data-based domains. Any domain can have this type of behaviour.

Interactive providers can receive any consumer-style request described in the consumer section of this document, including Batch requests, Views, Symbol Lists, Pause/Resume, etc. Provider applications should respond with a negative acknowledgement or response if the interactive application cannot provide the expected response to a request.

3.2.2.1.2 Posts

In a standard Post scenario, the Interactive Provider can receive a Post Message from the RTDS. The message will state whether an acknowledgment is required. If required, RTDS will expect the Interactive provider to provide a response, in the form of a positive or negative acknowledgement. Post behavior is not limited to Market Data-based Domains. Any Domain can have this type of behavior. When the Interactive Provider connects to the RTDS and publishes the supported domains, it then states whether the post capability is supported. Use cases for posting can be found in [Section 3.2.1.8](#).

3.2.2.1.3 Generic Message

Providers can receive a generic message from and publish a generic message to the RTDS or some OMM consumer. Use cases for generic messages can be found in [Section 3.2.1.9](#)

3.2.2.2 Non-Interactive Provider

A **Non-Interactive Provider** does not wait for a connection request to response as it is the one who makes a connection to RTDS and sends a specific set (non-interactive) of information (services, domains, and capabilities).

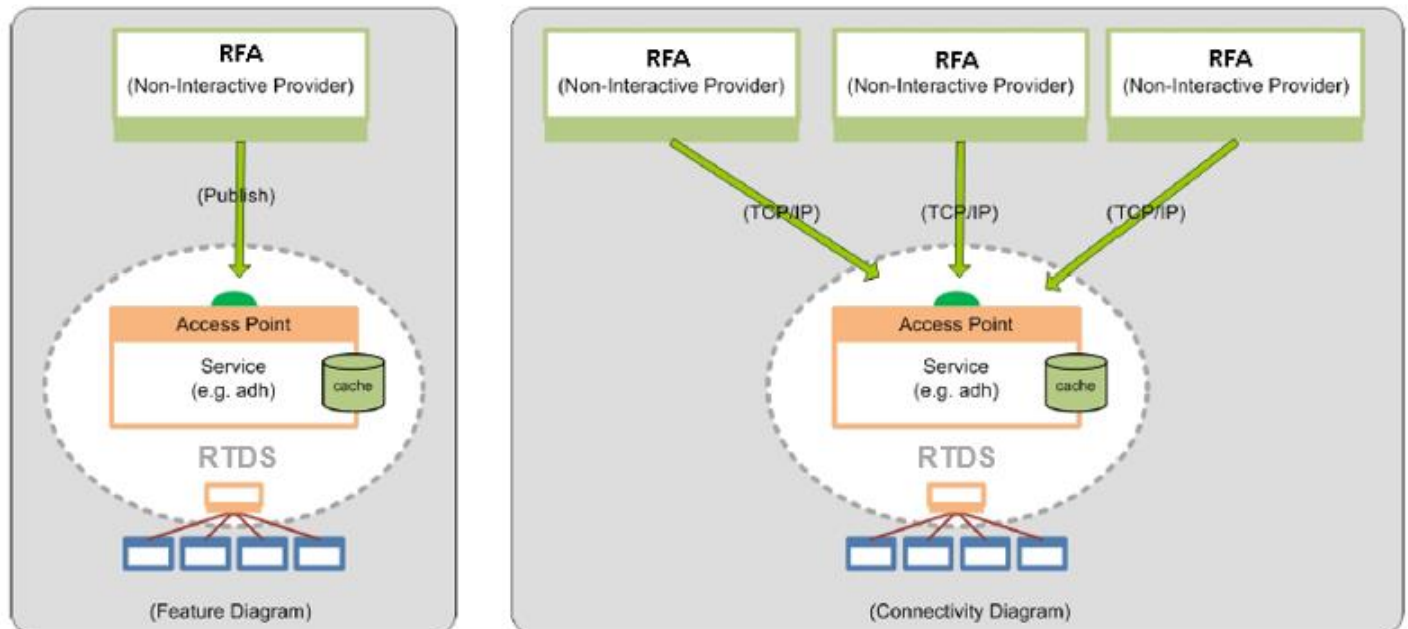


Figure 17: Non-Interactive Provider

After a Non-Interactive Provider connects to RTDS, the Non-Interactive Provider can start sending information for any supported item and any supported domain to the RTDS. The Non-Interactive Provider concept is similar to what was once called a Source Driven or Broadcast Server Application on legacy Refinitiv systems.

Non-Interactive Providers act like clients in a client-server relationship. Multiple Non-Interactive Providers can connect to the same RTDS 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 RTDS.

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.

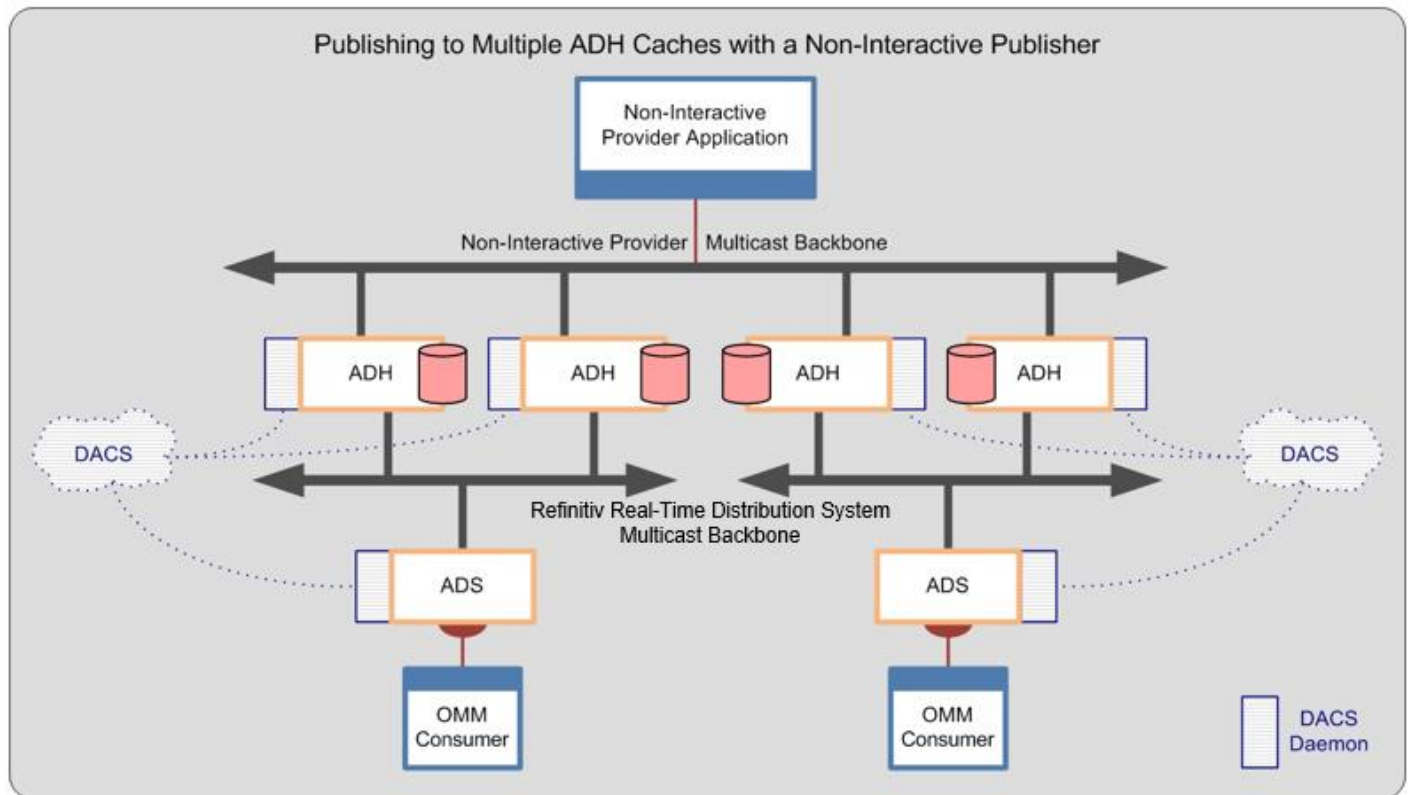


Figure 18: Publishing to Multiple ADH Caches

3.2.3 Hybrids

OMM Hybrid applications fill a niche for applications that can both provide *and* consume data. **Hybrids** can receive a request message from a consumer and forward the same request message to another provider. Conversely, a hybrid application can receive a response message from a provider and forward the same response message to the originating consumer. While the data is passing through the hybrid, the contents of the messages can be decoded, changed, and reencoded.

The following diagram shows a hybrid application layout.

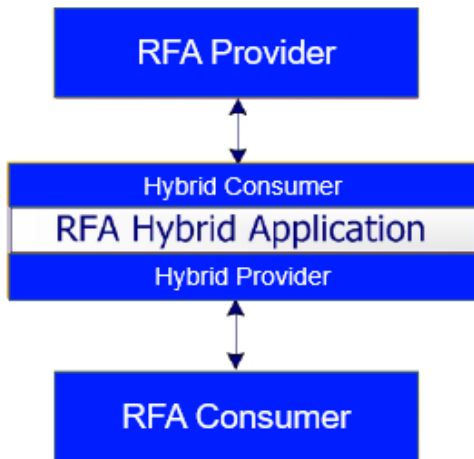


Figure 19: Hybrid Application

The hybrid application is comprised of two component parts: a consumer and a provider. The consumer part acts as a directly-connected consumer to an RFA provider in the above figure. The provider part is directly connected to another RFA consumer in the above figure. The Hybrid Consumer and Hybrid Provider do not need to send messages to each other because both exist in a single application.

3.3 Model Overview

3.3.1 Open Message Model (OMM)

Open Message Model (OMM) 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 are detailed in Chapter 6 “Data Package.”

The layout and interpretation of any specific OMM model (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. ETA and RFA provide support for OMM constructs and manage the RWF binary-encoded representation of the OMM.

3.3.2 Domain Message Model (DMM)

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

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

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

3.3.3 Refinitiv Domain Model (RDM)

Refinitiv Domain Model (RDM) 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 provide or consume various types of market data (e.g., Market Price, Market by Order, Market by Price).

3.3.4 User Defined Domain Model

User Defined Domain Model is a DMM defined by a third party. These might be defined to solve a need specific to a user or system in a particular deployment that is not resolved through the use of an 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.

3.3.5 Refinitiv Wire Format (RWF)

Refinitiv Wire Format (RWF) 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 a more complex, nested hierarchical content.

3.4 OMM Concepts

3.4.1 OMM Messages

This section provides a description of some key RDM message types. For more completed lists and descriptions, see Chapter 7 “Message Package.” Message concepts apply to both consumers and providers. For example, a request message is called a request message from both the provider and consumer application viewpoints.

3.4.1.1 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 either streaming or non-streaming (which are also known as snapshot requests). Requests can also be re-issued if the application wishes to change a characteristic of the original request. However, the application is allowed to change only the following characteristics:

- Priority class and count: allows consumers to tune the pre-emption algorithm used by the Providers.
- Interaction type: allows consumers to request a new refresh, and to pause or resume a given item.
- Data mask (contained on the attributes of the request message)
- Indication mask: allows consumers to modify an item's view.

3.4.1.2 Response Message

A response message is sent by a provider to a 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 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.N”, the provider will send back a refresh message containing its current data regarding “IBM.N.” After sending the refresh, the provider will send update messages to the consumer each time data for “IBM.N” (such as current price) changes. For more details on response messages, refer to section 6.2.2, Data Decoding.

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 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.

3.4.1.3 Generic Message

By 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 type inside it. One advantage of using Generic Messages is its freedom from the traditional Request/Response data flow. Generic messages can either be sent from consumers to providers or from providers to consumers.

Additionally, attributes of a Generic Message need not match the attributes of the stream over which the generic message flows. Thus, attributes information can be used independently within the stream.

3.5 Refinitiv Domain Models (RDM) 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 RDM Usage Guide .NET Edition*.

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.

The table below provides a high-level overview of the currently available Refinitiv Domain Models.

DOMAIN	PURPOSE
Login	Authenticates users and advertises/requests features that are not domain-specific. Use and support of this domain is required for all OMM applications. This is considered an administrative domain. Many Refinitiv components require a content, which is also expected to conform to the domain model definition.
Service Directory (also Source Directory)	Advertises information about available services and their state, QoS, and capabilities. This domain also conveys any group status and group merge information. Interactive and Non-Interactive OMM Provider applications require support for this domain. This is considered an administrative domain. Many Refinitiv components require a content, which is also expected to conform to the domain model definition.
Dictionary	Used to provide dictionaries that may be necessary to decode data. Dictionary domain use is optional, it is recommended for Provider applications to support this. This is considered an administrative domain. Many Refinitiv components require a content, which is also expected to conform to the domain model definition.
MarketPrice	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.
MarketByOrder	Provides access to Level II full order books. Contains a list of orders, keyed by order Id along with the information related to that order, such as price, whether it is a bid/ask order, size, quote time, and market maker identifier.
MarketByPrice	Provides access to Level II market depth information. Contains a list of price points (using the price and it's bid/ask side as its key) along with the information related to that price point.
MarketMaker	Provides access to market maker quotes and trade information. Contains a list of market makers (using that 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.
SymbolList	Provides access to a set of symbol names, 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.
YieldCurve	The YieldCurve domain shows the relation between the interest rate and the term, or time to maturity, associated with the debt of a borrower. The shape of the yield curve can help to give an idea of future economic activity and interest rates.

Table 8: Overview of Refinitiv Domain Models (RDM)

3.5.1 Service, Concrete Service, and Service Group

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 that provides 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 that provide 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.

3.6 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 3.6.1, OMM Messages. For more detailed information, refer to Chapter 6, “Data Package.”

RFA data can be represented in OMM form or in encoded RWF form. OMM is used for programmatic manipulation, where as 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. Both RFA and ETA Message and Data packages provide support for encoding and decoding data.

3.6.1 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.”

3.6.2 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 is 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 of hours, minutes, seconds, milliseconds, microseconds, and nanoseconds. DateTime is a combination of Date and Time. Uses Reuters.RFA.Data.Date , Reuters.RFA.Data.Time and Reuters.RFA.Data.DateTime .
String and Buffer	Buffers can contain any length-specified data, including strings.
QoS	Quality of Service provides a classification of the tier of service, divided into orthogonal sets of distinct properties. The two properties that make up QoS include: <ul style="list-style-type: none"> • Timeliness: the age of data • Rate: the period of change in data (i.e., the rate at which data changes)
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 9: RFA Primitive Types

3.6.3 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 being described first.

DATA TYPE	DESCRIPTION	CONTAINED ENTRY	DISTINGUISHING FEATURES
FilterList	<p>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 attribInfos filter member. Each entry must contain a container type.</p> <p>A container of loosely coupled entries.</p> <p>A FilterList can contain zero to N entries, with zero indicating an empty FilterList, though this type is typically limited by the number of available of filterId values.</p>	FilterEntry	Selectable entries
Vector	<p>A container of position-oriented, index-value, paired entries. Each entry's index is represented by an unsigned integer with a value between 0 and 230. A single fragment of a Vector may contain 64K Entries.</p> <p>A Vector can contain zero to N entries, where zero entries indicates an empty Vector.</p>	VectorEntry	<ul style="list-style-type: none"> Position-specified entries Optionally sortable by a Provider No risk of data loss without detection (Sorted) Applications can set, delete, update, clear or insert entries
Element-List	<p>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.</p>	ElementEntry	<p>A container of flexible, self-describing named entries.</p> <p>An Element List can contain zero to N entries, with zero indicating an empty Element List.</p>
FieldList	<p>FieldList is a non-uniform container of field identifier-value paired entries known as a fieldEntry. A Field Identifier, also known as a fieldId (or FID), 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 FieldList can be a primitive or container type.</p> <p>A FieldList can contain zero to N entries, with zero indicating an empty FieldList.</p>	FieldEntry	<ul style="list-style-type: none"> Efficient caching Conventional price discovery data structure
Map	<p>A container of key-value paired Entries. A Map may contain between 0 and 230 total Entries. zero to N entries, where zero entries indicate an empty Map.</p> <p>A single fragment of a Map may contain up to 64K Entries.</p>	MapEntry	<ul style="list-style-type: none"> Sortable by Consumer (not Provider). Flexible key type Unique identifier available per entry Critical to avoid incorrect action after data loss Applications can add, update or delete entries
Series	<p>A container of uniform Entries, often used to represent table-based information, where no explicit indexing is present or required.</p> <p>A Series can contain zero to N entries, where zero entries indicates an empty Series.</p>	SeriesEntry	<p>No entry identifier.</p> <p>Entries have an implicit order within the container.</p> <p>Entries have the same data format.</p>

DATA TYPE	DESCRIPTION	CONTAINED ENTRY	DISTINGUISHING FEATURES
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. An Array can contain zero to N primitive type entries , with zero indicating an empty Array.	ArrayEntry	Entries optionally support fixed-width contained entries, to minimize bandwidth consumption
Data-Buffer	A leaf that contains simple types including Int, UInt, Date, Time, DateTime and Real. This is the only Leaf supported	N/A	
Msg	The Msg contains header information. Additionally it can contain attributes and/or payload data	AttribInfo Data (as payload) Manifest Header	Flow between the provider and consumer applications.

Table 10: RFA Container Types

3.6.4 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 being included with each entry. Currency type is an example of information that may be communicated via summary data because it is likely to be the same for all entries in the container. Summary data is optional and applications can determine when to deploy it. Specific domain model definitions typically indicate whether summary data should be present, along with any content information. When included, the container type of summary data should match the container type of payload.

3.6.5 Defined Data

A local data definition allows additional optimizations to be performed on the contents of a FieldList or ElementList, resulting in fewer bytes being 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.

3.6.6 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 typespecific 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 was used, then only one iterator is needed to decode the entire FieldList.

Chapter 4 System View

You can deploy the RTDS in a variety of ways, including LAN, WAN, and Internet distribution. It can distribute many different types of data. RFA is designed for use in many different deployment environments. The following sections show where RFA provider and consumer applications can be deployed in the RTDS (and even in environments that do not use the RTDS).

4.1 Background

4.1.1 RTDS Connection and Protocol Types

PROTOCOL TYPE	DESCRIPTION
RSSL	Refinitiv SSL – TCP/IP based protocol used by the RTDS. RFA uses RSSL, RSSL_PROV and RSSL_NIPROV connection types.

Table 11: Protocol Types

4.1.2 RTDS Components

COMPONENT	DESCRIPTION
Data Feed Direct	Refinitiv Data Feed Direct (RDF Direct) is a fully managed Refinitiv exchange feed providing an ultra-low-latency solution for consuming data from specific exchanges. The RDF Direct normalizes all exchange data using the OMM.
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 the OMM.
ADH	The ADH is a networked, data distribution server that runs in the RTDS. It consumes data from a variety of content providers and reliably fans this data out to multiple ADSs over a backbone network (using either multicast or broadcast). RFA Non-Interactive or Interactive Provider applications can publish content directly into an ADH, thus distributing data more widely across the network.
ADS	The ADS provides a consolidated point-to-point distribution solution for Refinitiv, value-added, and third party data for trading room systems. It distributes information using the same OMM and RWF protocols exposed by the RFA.
ATS	ATS is a multifunctional server that brings real-time data together from multiple sources, transforms, calculates and publishes resulting data onto the platform for further distribution and consumption by other applications and users.
DACS	DACS is the entitlement system for the RTDS. To perform a permission check for an item, DACS must have requirements information for the item and a profile for the user (a.k.a. content based permissioning).

Table 12: RTDS Components

4.1.3 RTDS Data Formats

PROTOCOL TYPE	DESCRIPTION
Refinitiv Wire Format (RWF)	Bandwidth optimized, binary wire format used for transmitting OMM messages and data. The OMM Data and Message Packages support decoding and encoding RWF. RWF actually supports a message format and a family of flexible data formats. The message format has a mechanism to identify which RWF data format or other non-RWF data format is used in the message payload.
ANSI Page	ANSI X.34 encoded, ready-to-display page data. Can be encoded and decoded by the ANSI Page Package. It can be used as a payload in RWF. This data format is used by many third-party data feeds.

Table 13: RTDS Data Formats

4.1.4 Connections and Supported Data Dictionaries

This section describes the relationship between the Connection Types and their supported Data Dictionaries. Currently RFA applications support Data Dictionaries type only RDM Field Dictionary.

The Data Dictionary can be either downloaded from the infrastructure (Download) or loaded from the local directory (File). The following tables detail how Data Dictionary loading is supported by each connection type.

INFRASTRUCTURE COMPONENT	CONNECTION TYPE	DATA DICTIONARY TYPE	DATA DICTIONARY SOURCE
ADH	RSSL	RDM Field Dictionary	RDMFieldDictionary and enumtype.def files
ADS	RSSL	RDM Field Dictionary	RDMFieldDictionary and enumtype.def files / RDM Dictionary download
RSSL Direct	RSSL	RDM Field Dictionary	RDMFieldDictionary and enumtype.def files / RDM Dictionary download

Table 14: Infrastructure Component and Type of Data Dictionary Supported

4.2 OMM Infrastructures

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 Message 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 ETA or RFA.

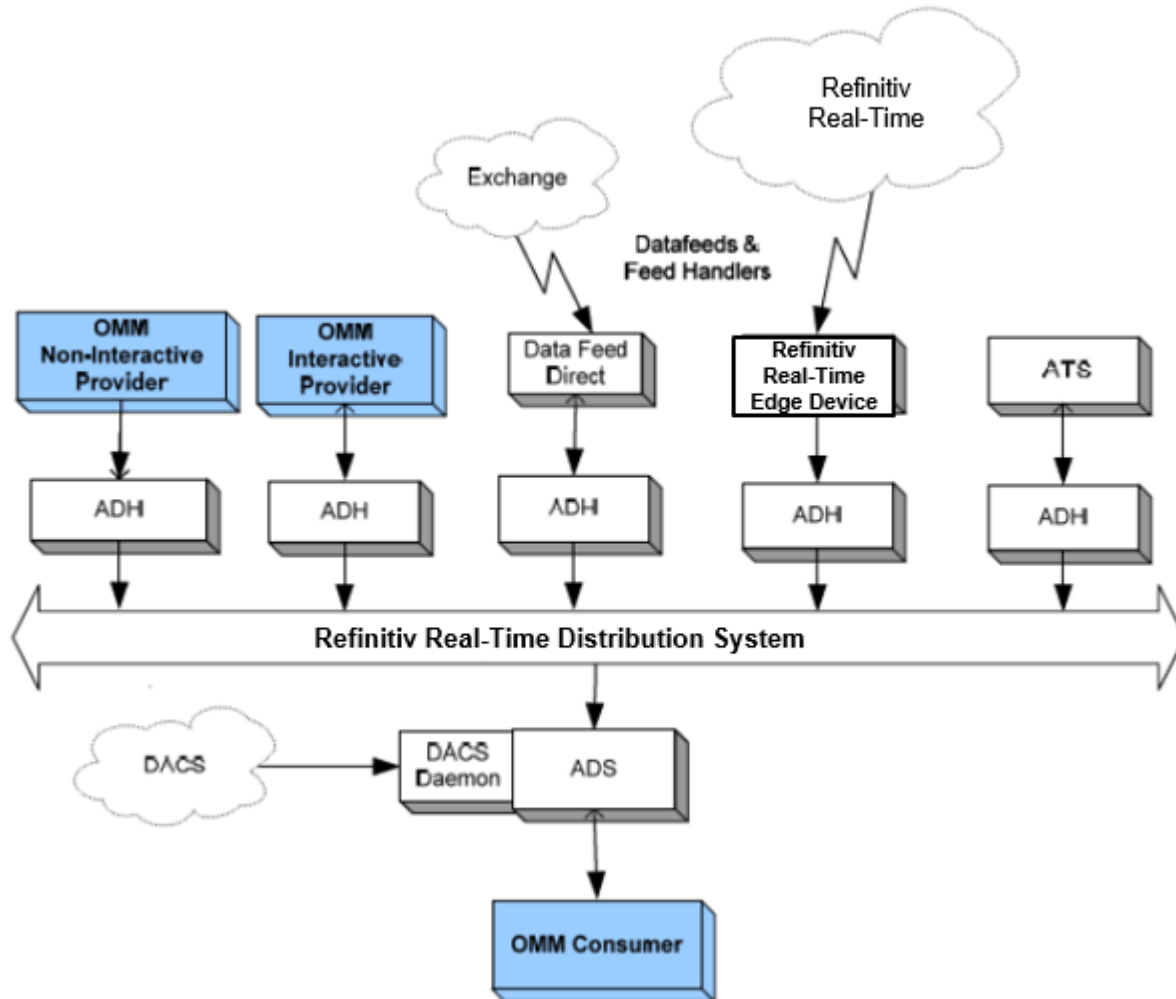


Figure 20: Typical Components in an RTDS Network

4.2.1 Refinitiv Real-Time Advanced Distribution Server (ADS)

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

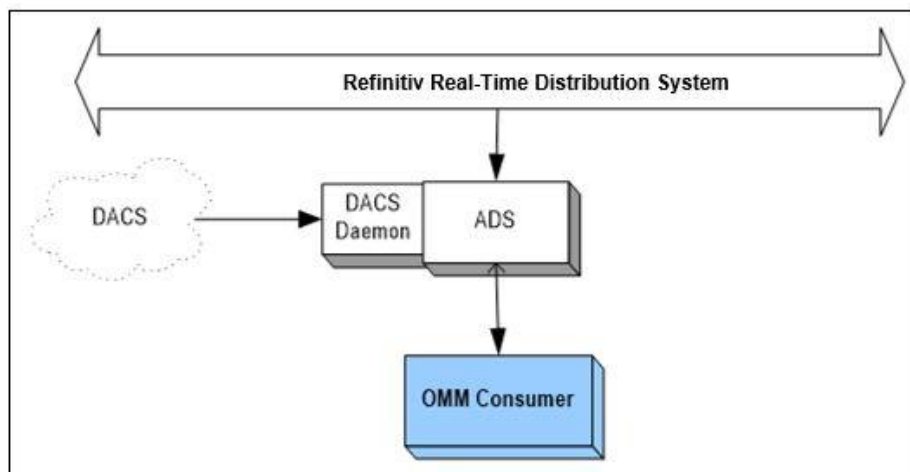


Figure 21: Refinitiv Real-Time Advanced Distribution Server

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.

4.2.2 Refinitiv Real-Time Advanced Distribution Hub (ADH)

The ADH is a networked, data distribution server that runs in the RTDS. It consumes data from content providers and reliably fans this data out to multiple ADSs 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.

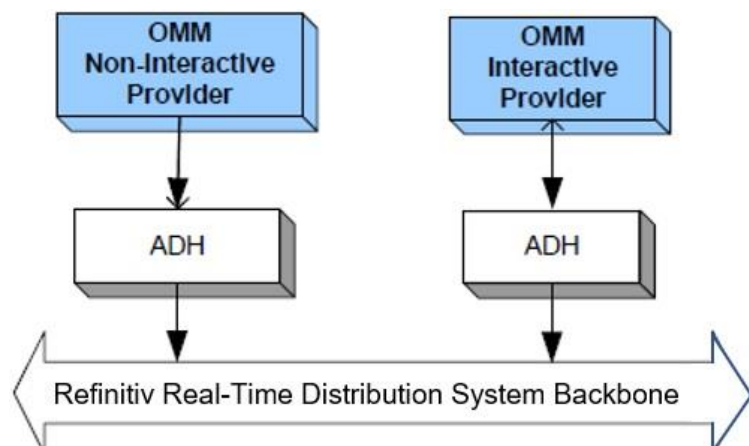


Figure 22: Refinitiv Real-Time Advanced Distribution Hub

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.

4.2.3 Refinitiv Real-Time Advanced Transformation Server (ATS)

The following setup shows the RTDS point-to-point setup. DACS is omitted from the diagram for clarity. The primary source of the data in these applications is an OMM Contributor.

The connection types used for OMM Contributors are the same as for OMM Consumer. However most of the data flows in the reverse direction, from the client distribution network through the ADS to the RTDS backbone. On the RTDS backbone, contribution systems, ATS, and even OMM Provider can receive the contributed data. They are responsible for sending an ACK or NAK acknowledgement back through the RTDS to the OMM Contributors.

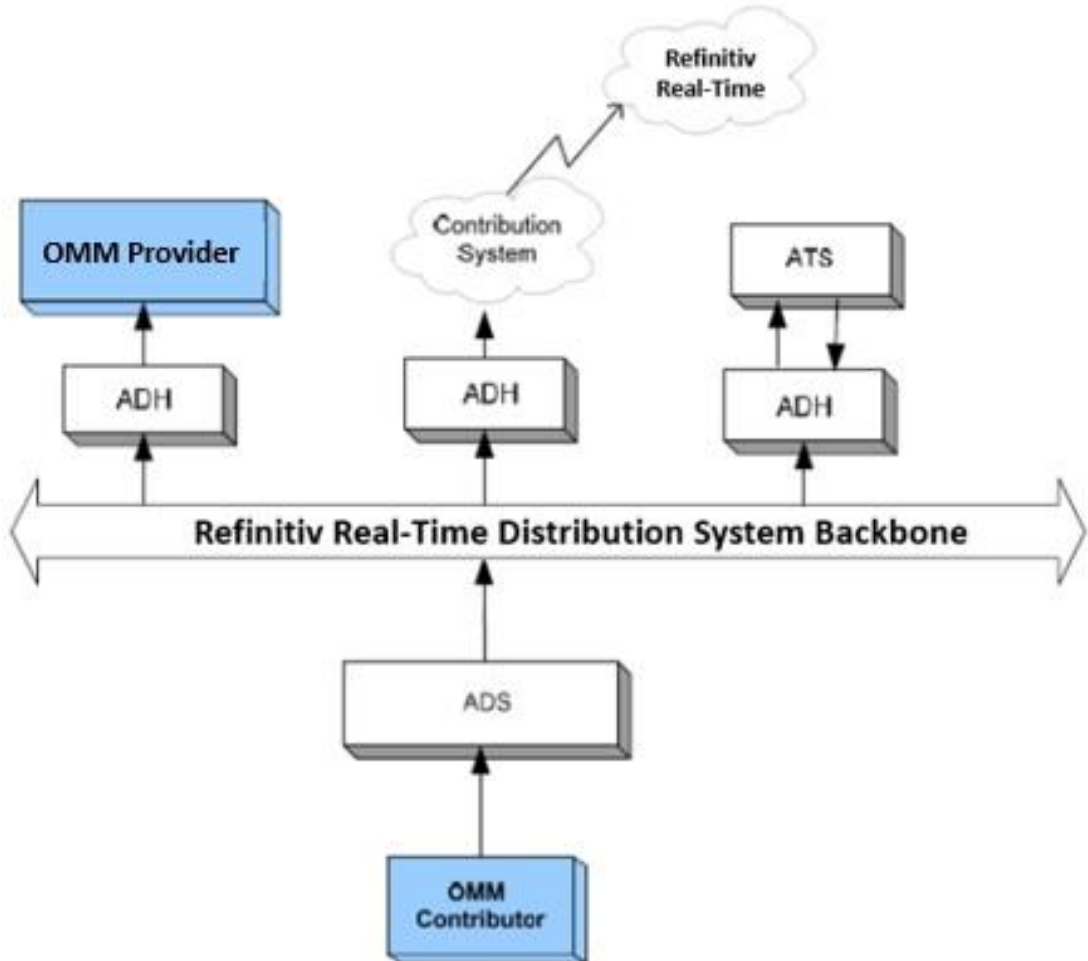


Figure 23: Refinitiv Real-Time Advanced Transformation Server

4.2.4 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 the OMM.

The following diagram shows how Refinitiv Real-Time setup with ADS and consumer application. The Refinitiv Real-Time Edge Device, based on ADS technology, is the access point for consuming this data. To access this content, the consumer applications can connect directly to the Edge Device or via a cascaded RTDS.

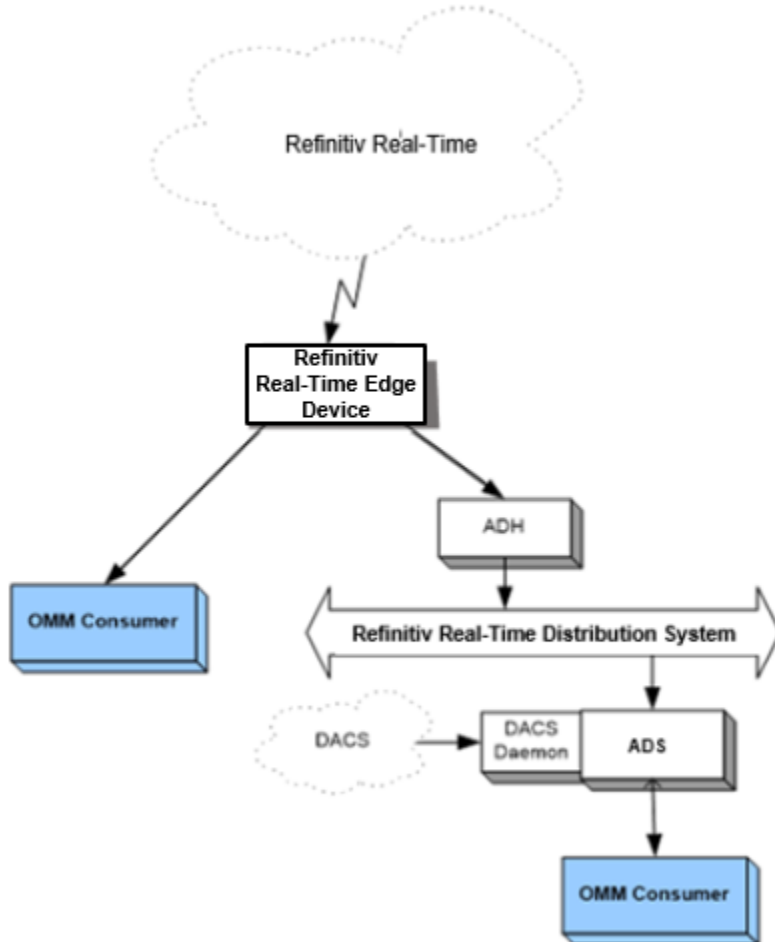


Figure 24: Refinitiv Real-Time

4.2.5 Refinitiv Data Feed Direct (RDF Direct)

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

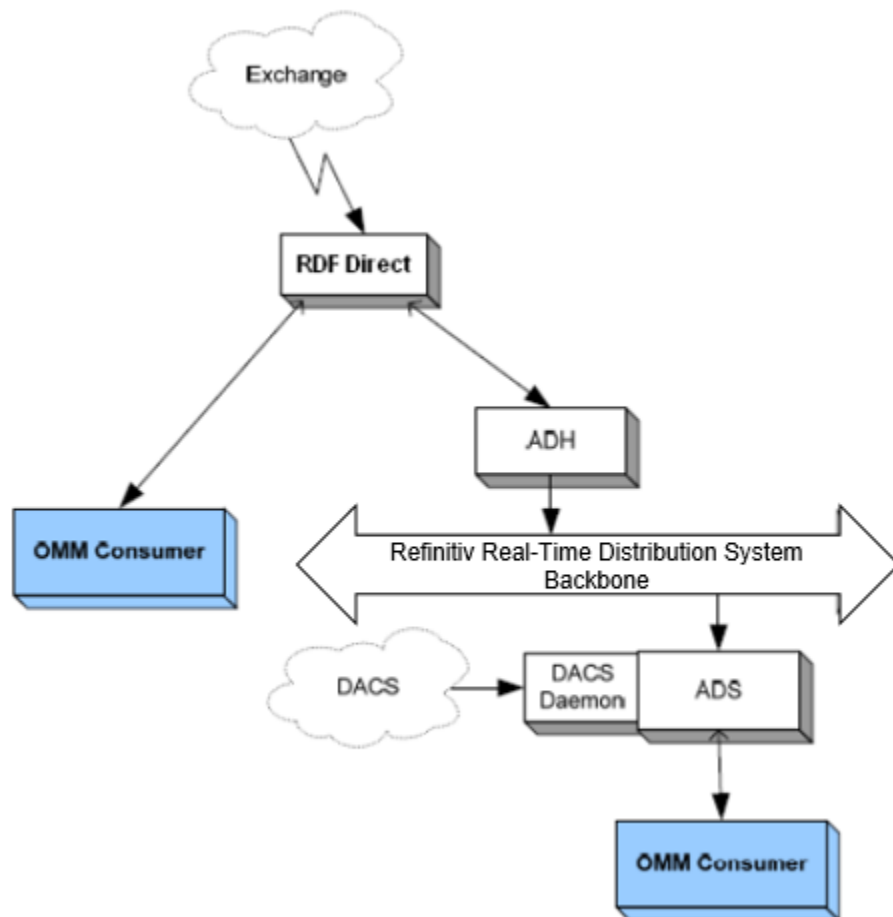


Figure 25: Refinitiv Data Feed Direct (RDF Direct)

To access this content, consumer applications can connect directly to the Data Feed Direct or via a cascaded RTDS.

4.2.6 Direct-Connect

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

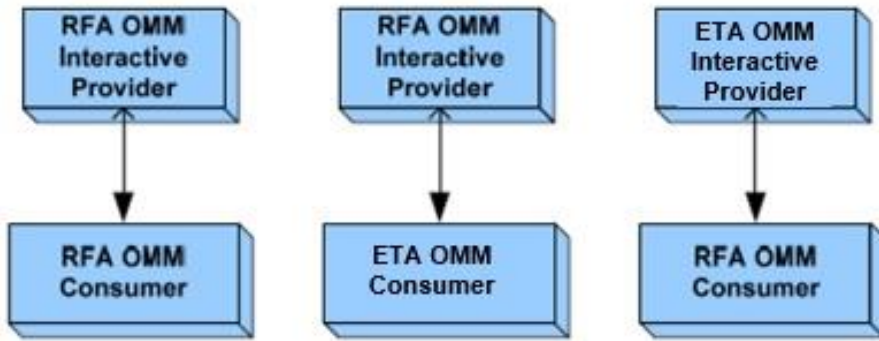


Figure 26: Direct-Connect

Direct connections are useful for extremely high performance applications that need the lowest achievable latency and do not need the resiliency provided by the RTDS. Direct connections can also be extremely useful during application development. Applications can be tested with a direct connection to a custom provider and then deployed with a full RTDS.

Non-Interactive Providers cannot be used with direct connections.

4.2.7 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 RSSL Transport connections tunnelled via HTTP (such functionality is available across all supported platforms). Establishing an HTTPS connection to a provider requires the use of an SSL 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 Microsoft WinInet. Users can configure certificates and proxies via Internet Explorer. Because of its dependency on the WinInet library, consumer HTTP and HTTPS tunneling are available only on supported Windows platforms. For the specific RFA tunneling configuration parameters, tunnelingObjectName, tunnelingReconnectionTime, and tunnelingType, see the *RFA Configuration Guide .NET Edition*.

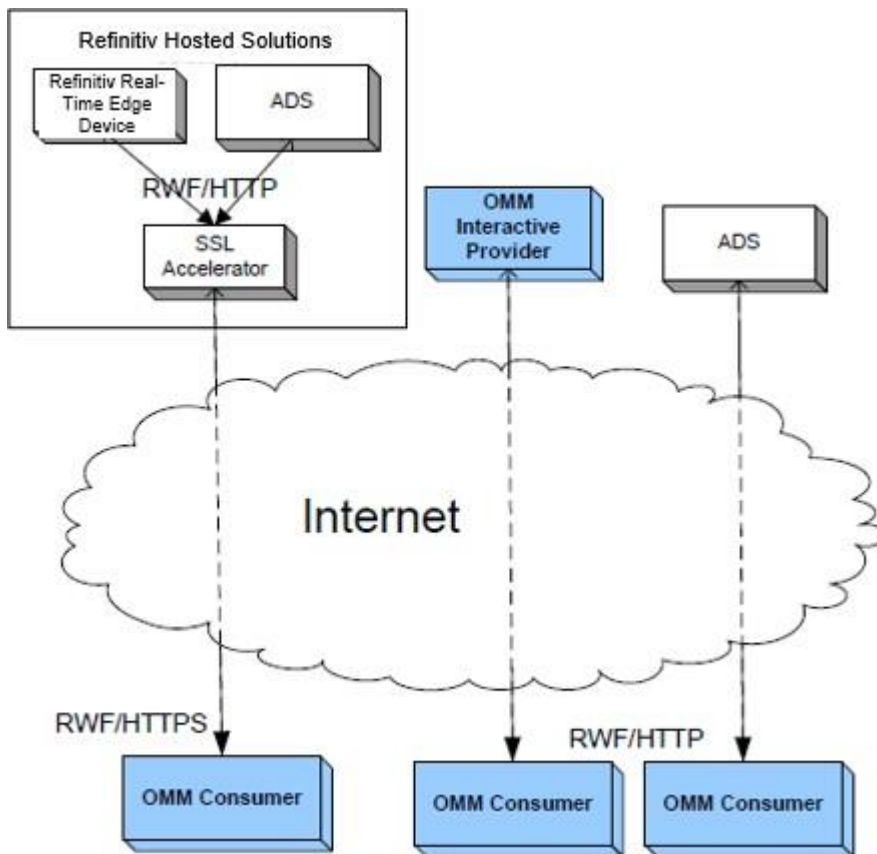


Figure 27: Internet Connectivity

Chapter 5 Common Package

5.1 Common Package Concepts

The **Common Package** provides general 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:

- An Event Distribution mechanism
- A Context used to coordinate the interaction between all other RFA packages
- Supports OMM through the abstract **Data** and **Msg** interfaces
- Class interfaces such as **Buffer** and **RFA_String**.
- 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
- Interfaces to support Quality of Service
- Abstract interfaces for other RFA packages to build on

One of the advantages of having this functionality implemented by the Common Package is that it gives a consistent look and feel to other RFA packages that depend on the Common Package.

5.1.1 Event Distribution Model

The Common Package provides the Event Distribution Model, which is a thread-safe³, thread-aware mechanism to deliver asynchronous notification. It is designed for use in both single- and multi-threaded applications.

The Event Distribution Model introduces the fundamental concept of interest. For example, an application that requests a particular market data Item is said to have interest in that market data Item.

The Event Distribution Model provides interfaces to manage interest. The following table defines concepts related to the Event Distribution Model. This section describes interfaces related to these concepts.

CONCEPT	DESCRIPTION
Client	Application callback that receives dispatched Events. A Client can execute in either the application's or the Session Layer's thread context.
Closure	A handle to System.Object supplied by the application. An application may supply Closure upon sending an Interest Specification. RFA returns the Closure in Events that correspond to the item associated with the Closure.
Dispatch	An application action that obtains the next available Event from an Event Queue or Event Queue Group.
Event	A notification that there is data to process. Applications are notified when there is data to process, and the notification is called an event. The data is contained in the event.
Event Queue	A container of Events awaiting delivery to an application.

³ Thread safety in RFA interfaces is interface-specific. Some interfaces are thread-safe at the class level, which means that static methods can be called from multiple threads at the same time, and that if there are any class-wide resources (i.e., static data members), access to these resources from class instances is properly synchronized. Other interfaces are thread-safe at the object level, which 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. The level of thread safety for each interfaces is documented in the *RFA Reference Manual*.

CONCEPT	DESCRIPTION
Event Queue Group	A container of Event Queues that emulates a single Event Queue.
Event Source	An RFA component that sends one or more Events. For example, an OMMConsumer and OMMProvider are both Event Sources.
Event Source Factory	An entity that creates Event Sources
Event Stream	The sequence of Events associated with a specific request.
Handle	A unique identifier representing an open Event Stream that is valid for the life of that stream.
Interest Specification	A request sent from an application with the intent of opening interest in a particular entity.
Notification Client	Application callback 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.

Table 15: Event Distribution Model Concepts

5.1.1.1 Interest Specification

For market information, the most typical interaction pattern for consuming applications can be described as “single request, multiple asynchronous responses.” For example, an application can make a request which typically results in an initial response followed by a continuous stream of subsequent responses.

When an application makes a request that will result in one or more responses, the application effectively specifies its “interest.” The specification of interest is called an Interest Specification.

An application specifies interest by submitting its Interest Specification to an Event Source, which is an entity that typically reacts to an Interest Specification by sending one or more asynchronous responses. The Event Distribution mechanism assigns a distinct Handle each time an application makes a request that specifies an application's interest of some type. Each event received as a result of a previously made request contains, among other things, the corresponding handle.

5.1.1.2 Events and Event Streams

An RFA package delivers asynchronous notifications to the application as Events. An Event is nothing more than 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. Each Event Stream corresponds to a unique Handle.

5.1.1.3 Event Queues and Event Queue Groups

Events are delivered to an application via Event Queues. An application processes queued-up events by dispatching them from an Event Queue or an Event Queue Group. The act of dispatching an event results in passing the event object to an application-defined Event Handler. The application can maintain full control over its threads by dispatching Events in its own context. 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.)

Event queues are not always required. Consumer and Provider applications can use a null event queue, where RFA directly calls a consumer or provider callback function respectively. This is known as the Callback Model. This configuration affects overall performance. For more details see section 14.3, Threading Model.

An application can also use an Event Queue Group to dispatch multiple Event Queues from a single thread context. Adding the relevant Event Queues to an Event Queue Group is more efficient than having a single thread poll multiple queues. It also adds the benefit of allowing the application to block (infinite or with a timeout) on the entire Event Queue Group while waiting for an event to become available from any queue within that group. An application typically uses an Event Queue Group to prioritize dispatching among different Event Queues or when dispatching from multiple Event Queues in a single thread context.

5.1.1.4 Opening and Closing Event Streams

Typically an Event Stream associated with an interest specification remains open, or active, until it is closed by the application or closed by an upstream component. There are also some special cases when RFA may decide to close an interest specification. In either case the corresponding Event Stream gets closed.

All currently open Event Streams have application-wide unique handles. Once an Event Stream has been closed the handle will be reused by the API for use with future event streams. The implementation of the Event Distribution mechanism guarantees that it will not reuse the handle for some “reasonably high” number of new interest specifications.

5.1.1.5 Completion Events

The process of closing an interest specification works slightly differently depending on whether the application uses one thread or multiple threads to handle interest specifications and to dispatch events. If an application uses one thread for both closing interest specifications and dispatching events from the associated Event Queue, then the implementation of the Event Distribution mechanism guarantees that once it has returned from the “close interest specification” call, the application will not receive any events related to the closed interest specification. The application can now clean up all the resources associated with this interest specification—e.g., the application can delete the corresponding instance of the Event Handler (assuming that this instance is not being used for any other specifications of interest).

However, if an application closes an interest specification from one of its own threads and simultaneously keeps dispatching events from the associated Event Queue in a different thread, there is the possibility of a race condition where the application will receive an event related to the interest specification being closed after having returned from the “close interest specification” call.

Because of this possibility, the application must not delete the corresponding instance of the Event Handler upon returning from the “close interest specification” call. Instead, it should wait for an indication from the Session Layer that there will be no further events associated with this particular Event Stream.

The Session Layer uses the “Event Stream Closed” flag to indicate that this is the last event for a given Event Stream (and the corresponding Handle). Any Event can have this flag set: it can be an interest specification-specific Event or a special Completion Event. The latter exists solely for the purpose of notifying an application that this is the last Event for a given Event Stream (i.e., a given Handle). RFA guarantees that the application will not receive any events related to the closed interest specification after it has received an Event with this flag set (whether it’s a Completion Event or any other Event). If the stream is closed by an upstream Provider, RFA will send a completion event as the last event dispatched on that handle.

Usage of the “Event Stream Closed” flag and Completion Events is optional; they can be used in both single-threaded and multi-threaded applications, but they are more useful in the latter.

5.1.1.6 Closures

A Closure is a handle to `System.Object` supplied by the application when making an interest specification. Each Event corresponding to this interest specification contains the same Closure.

An application can use a Closure if it needs to pass some application-specific context information from the part of the application where an interest specification is made to the application part responsible for response processing (i.e., to an Event Handler). For example, a Batch request could be associated with a closure by passing in a unique closure pointer in `RegisterClient()`. All responses associated with the batch request would include a reference to the Closure, allowing the application to make an association to the original batch request.

The Closure is not interpreted by the Event Distribution mechanism in any way; its use is optional and application-specific. For an example of a Closure see Section 1.2.2.

5.1.1.7 Notification Clients

In some cases application developers may wish to integrate the event dispatching logic into an existing event dispatching mechanism; such as Threading Notification. To simplify this task the Event Distribution mechanism has a concept of a Notification Client. An application can register a Notification Client with an Event Queue or an Event Queue Group. Every time a new message is posted to the Queue (or one of the Queues within the Group), an application-defined Notification Client will be called.

It's important to note that the Notification Client is called in the context of an RFA thread that just queued up the event to an Event Queue. In this case the Common Package calls application code in the context of an RFA thread. The ability for an application to choose between polling (i.e., dispatching), polling with a time-out, or asynchronous notification (via a Notification Client) is known as Controlled Dispatching.

5.1.1.8 Example of the Event Distribution Model

Even though the Session Layer Package is described later, a concrete example might make it easier to understand these Event Distribution concepts:

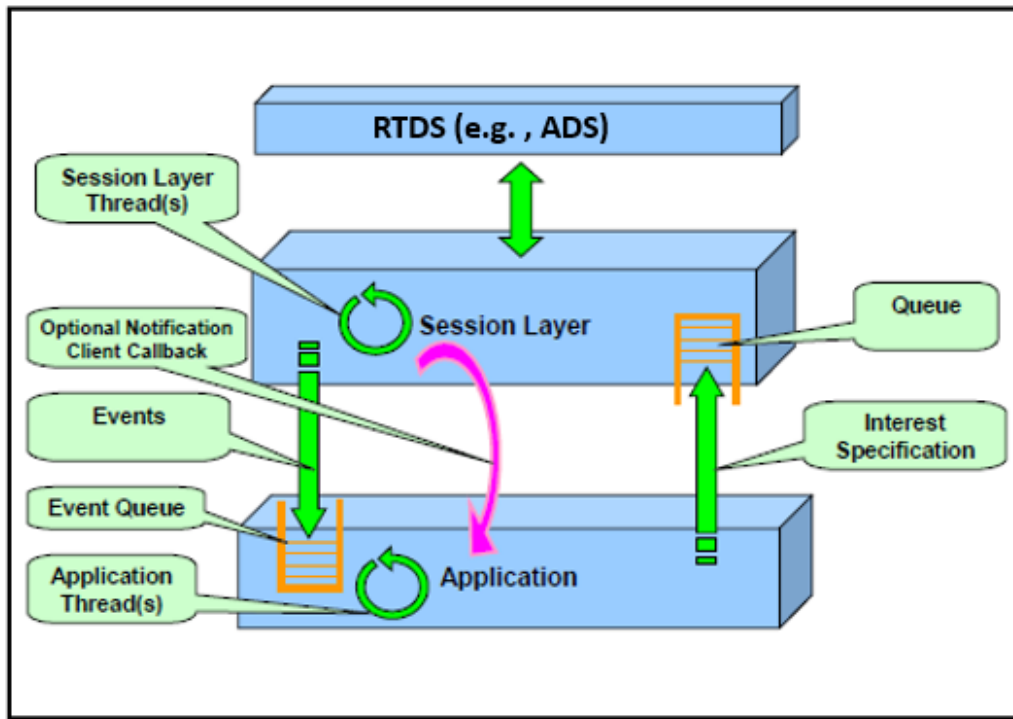


Figure 28: Session Layer's Usage of the Event Distribution Mechanism

One of the functionalities of the Session Layer Package is to obtain market information. When an application decides that it has an interest in a specific item or a group of items it makes an interest specification. The market information that is requested (e.g., the item name) is contained in an Interest Specification.

When an Interest Specification is submitted to the Session Layer, the application receives a unique Handle. In addition to using the handle to reference received events, the Handle is used by the application when modifying (reissuing) or closing (unregistering) its interest.

As a result of sending an Interest Specification, the Session Layer returns one or more Events such as an item image or item updates. These Events are pushed onto the Event Queue that was specified by the application when making the subscription request. Applications process these events by dispatching them from the Event Queue.

5.1.2 General Common Package Concepts

The Common Package introduces other concepts such as buffer and status. Several of these concepts directly map to interfaces provided by the Common Package. The following table defines general Common Package concepts and describes the related interfaces.

CONCEPT	DESCRIPTION
Buffer	A class that provides simple encapsulation of raw data as an object that contains both the data and data length. The Buffer supports typical operations such as copy and compare.
Command	The Cmd interface is an abstraction for all commands. For example, applications use Commands to submit data. Concrete descendants of a Cmd do not establish interest when sent to an Event Source.
Context	Used by an application to integrate multiple RFA packages and coordinate the initialization and shutdown of these packages. The context also provides versioning information for all RFA packages and underlying libraries.
Data	The abstract interface for all data contained in the Data Package. An application uses concrete realizations of this to access OMM data.
Exceptions	A set of RFA specific exception classes used to convey error conditions such as invalid configuration or invalid parameters.
Message	An abstract container class of header and raw data. Messages flow between service provider and service consumer applications. The Message Package defines five types of messages known as ReqMsg , RespMsg , GenericMsg , PostMsg , and AckMsg . All messages include a Message Model type which identifies specific type of market information (e.g., MarketPrice) and other information (e.g., Login). From an interface viewpoint, RDM is a namespace that contains these specific Message Model types. All concrete descendants of Interest Specifications, Events, and Commands contain Messages.
Quality of Service	A method of classifying the rate and timeliness of information provided by a service. The Session Layer Package uses Quality of Service (QoS).
RFA String RFA Wide String	A class that provides simple encapsulation of non-binary data that accepts length-specified or nullterminated data. It is a simple container that contains methods useful for string manipulation.
Response Status	Conveys health information associated with both the item stream and the data being exchanged.
Service (Concrete Service)	A named grouping of data items supplied by a single data vendor.
Service Group	A combination of multiple services into what appears to the user to be a single service. It has its own name and allows for item routing and recovery. Service Groups may be supplied from one or multiple providers.
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.
Status	Provides a simple Status interface. Various RFA packages define specific status interfaces that derive from this Status interface.

Table 16: Common Package Concepts

5.2 Common Package Usage

5.2.1 Event Distribution Model

As mentioned in Section 5.1.1, an Interest Specification is a request sent from an application with the goal of opening interest in a particular entity. RFA will send events to the application after it has obtained interest.

An Event Stream is the sequence of Events that are sent to the application. An application always opens an Event Stream but either the application or RFA may close the Event Stream. An example of the former is an application that closes an updating item stream. An example of the latter is the close, via a [RespStatus](#), of an item initiated from a back-end system.

Figure 29 shows the typical activities for an application using the Event Distribution Model. Each oval icon depicts a particular activity. The vertical line separates activities performed by the application from those performed by RFA.

The arrows depict transitions from one activity to another. Transitions that occur in parallel imply the application may perform the set of activities in any order. However, the application must perform all activities before performing a converging activity—for example, the path beginning from the start state (denoted by the filled dot) and converging at the Open Event Stream icon.

The diagram roughly divides the activities into three horizontal groups. The upper four activities relate to initialization. The lower four activities relate to shutdown. The remaining activities relate to Event Stream processing. This section focuses on Event Stream processing activities.

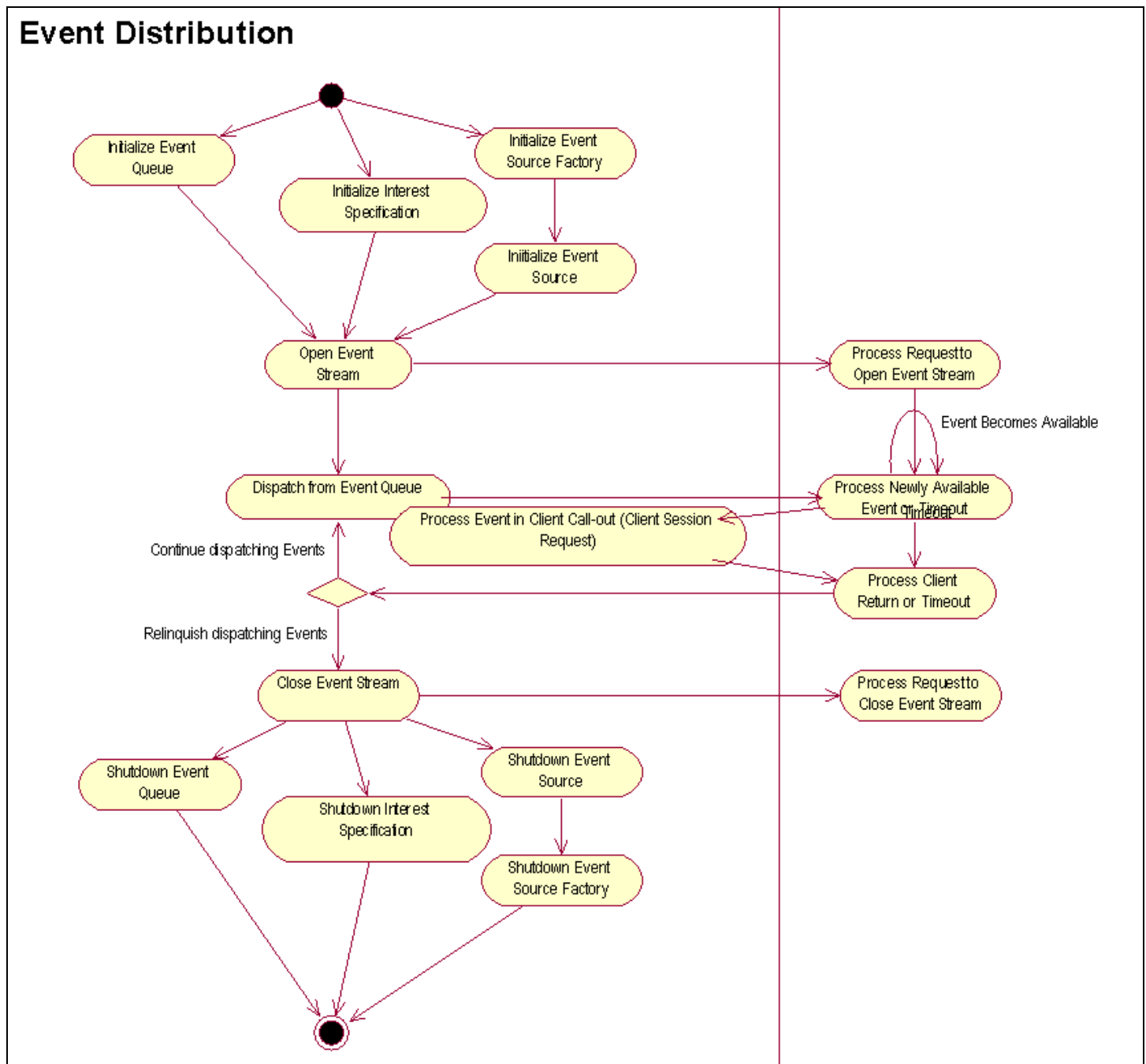


Figure 29: General Event Distribution

5.2.1.1 Using Event and Event Queue

The sections below discuss using Event and Event Queues. An Event is a notification that there is data to process, while an Event Queue is a container of Event awaiting delivery to an application.

5.2.1.1.1 Initialize Event Queue

Typically, one of an application's first activities is to perform the Initialize Event Queue activity. To initialize the Event Queue, the application calls the static `Create()` method and specifies a name, which provides identification for purposes such as logging.⁴

The `Create()` method returns a handle to a unique instance of an Event Queue. The application retains this handle to perform subsequent activities.

```
RFA_String eventQueueName = new RFA_String( "myEventQueue");
EventQueue eventQueue = EventQueue.Create( eventQueueName );
```

Example 1: Initialize Event Queue

The Event Queue can also provide statistics, which allows the application to retrieve the current size of the queue and set the low threshold, high threshold, and maximum depth levels for the queue.

5.2.1.1.2 Dispatch from Event Queue

Assuming the application has completed the previous activities, it typically next performs the Dispatch from Event Queue activity. An application typically receives Events by calling the `Dispatch()` method on the `EventQueue` interface (i.e., an Event Queue handle). Every time an application calls `Dispatch()`, the next Event, if available, is dispatched.

```
long dispatchReturn = eventQueue.Dispatch(5000);

switch (dispatchReturn)
{
    case Dispatchable.DispatchReturnEnum.NothingDispatched:
        Console.WriteLine("Nothing dispatched. Timeout occurred.");
        break;

    case Dispatchable.DispatchReturnEnum.NothingDispatchedInactive:
        Console.WriteLine("Nothing dispatched - Inactive. The EventQueue has been deactivated.");
        break;

    case Dispatchable.DispatchReturnEnum.NothingDispatchedPartOfGroup:
        Console.WriteLine("Nothing dispatched - PartOfGroup. The EventQueue is part of an EventQueueGroup
and should not be used directly.");
        break;

    case Dispatchable.DispatchReturnEnum.NothingDispatchedNoActiveEventStreams:
        Console.WriteLine("Nothing dispatched - NoActiveEventStreams. The EventQueue does not have any
open Event Streams.");
        break;
```

⁴ The application may also not specify a name, in which case an internal name is generated.

```

default:
    Console.WriteLine("Event dispatched. Approximate pending Events:" + dispatchReturn);
    break;
}

```

Example 2: Dispatch from Event Queue with Timeout

The method accepts a single optional parameter, which is a timeout value that informs RFA to block no longer than the specified time period. If an Event does not become available within this time period, the method returns, indicating nothing was dispatched and a timeout occurred.

An application may also call the `Dispatch()` and specify an `InfiniteWait` timeout value. In this case, RFA will block until an Event becomes available. Alternatively, an application may specify a timeout value of zero. In this case, dispatching will not block. Rather, RFA will check if an Event is available. If none is available it will immediately return. Dispatching with a timeout value of zero implies polling.

A non-negative return value from `Dispatch()` means that an Event was dispatched, and indicates an estimate of the number of remaining Events in the Event Queue or the Event Queue Group. Applications should not rely on the exact value as it is only an estimate. Negative return values have special meaning—e.g., a negative value can indicate that there are no active Event Streams associated with a given Queue or Group. An application is allowed to delete the Event Queue or the Event Queue Group only after receiving the negative return value that indicates there are no active Event Streams. (If the Queue or the Group is deleted without receiving this indication the result is unpredictable.) For a list of return values from `Dispatch()`, see the *RFA Reference Manual .NET Edition*.

If an Event is available following a call to `Dispatch()`, RFA invokes the Client callback method specified when opening the Event Stream. The Client callback method executes in the application's thread context provided by the call to `Dispatch()`.

Once the Client callback method returns to RFA, RFA relinquishes control back to the application and frees the currently blocked call to `Dispatch()`.

5.2.1.1.3 Process Event in Client

As indicated above, if an Event becomes available following a call to `Dispatch()`, RFA invokes the Client callback method specified when opening the Event Stream.

Event objects are only valid within the function context of the Event Handler function call (i.e., `ProcessEvent()`). Once the application returns from `ProcessEvent()` the Event object is no longer valid. If the application wishes to asynchronously process the Event it will need to make its own copy for later processing by using the `Clone()` method.

```

public void ProcessEvent(Event rfaEvent)
{
    Console.WriteLine("EventQueue name is:" + rfaEvent.EventQueue.Name.ToString());

    Console.WriteLine("Age of the Event in milliseconds is:" + rfaEvent.EventAge);

    if (rfaEvent.IsEventStreamClosed)
        Console.WriteLine("Event stream is closed");
}

```

Example 3: Process Event in Client

NOTE: Cloning an event performs a heap allocation, which can have a performance impact.

5.2.1.1.4 Shutdown Event Queue

After relinquishing interest, the application shuts down the event queue by calling the `Destroy()` method. An application can also call `Deactivate()` on the Event Queue prior to calling to `Destroy()`, which prevents any more Events from being put onto the Event Queue. If the application desires, it could dispatch any remaining Events before destroying the Event Queue. However, it must not `Destroy()` an Event Queue while it has active event streams.

```
eventQueue.Deactivate();  
  
eventQueue.Destroy();
```

Example 4: Shutdown Event Queue

NOTE: Unlike the Event provided in the Client callback method, the application is responsible for cleaning up a cloned Event with the Event's `Destroy()` method.

5.2.1.2 Using Event Queue Group

An Event Queue Group is a container of Event Queues that emulates a single Event Queue. An application typically uses an Event Queue Group to prioritize dispatching among different Event Queues.

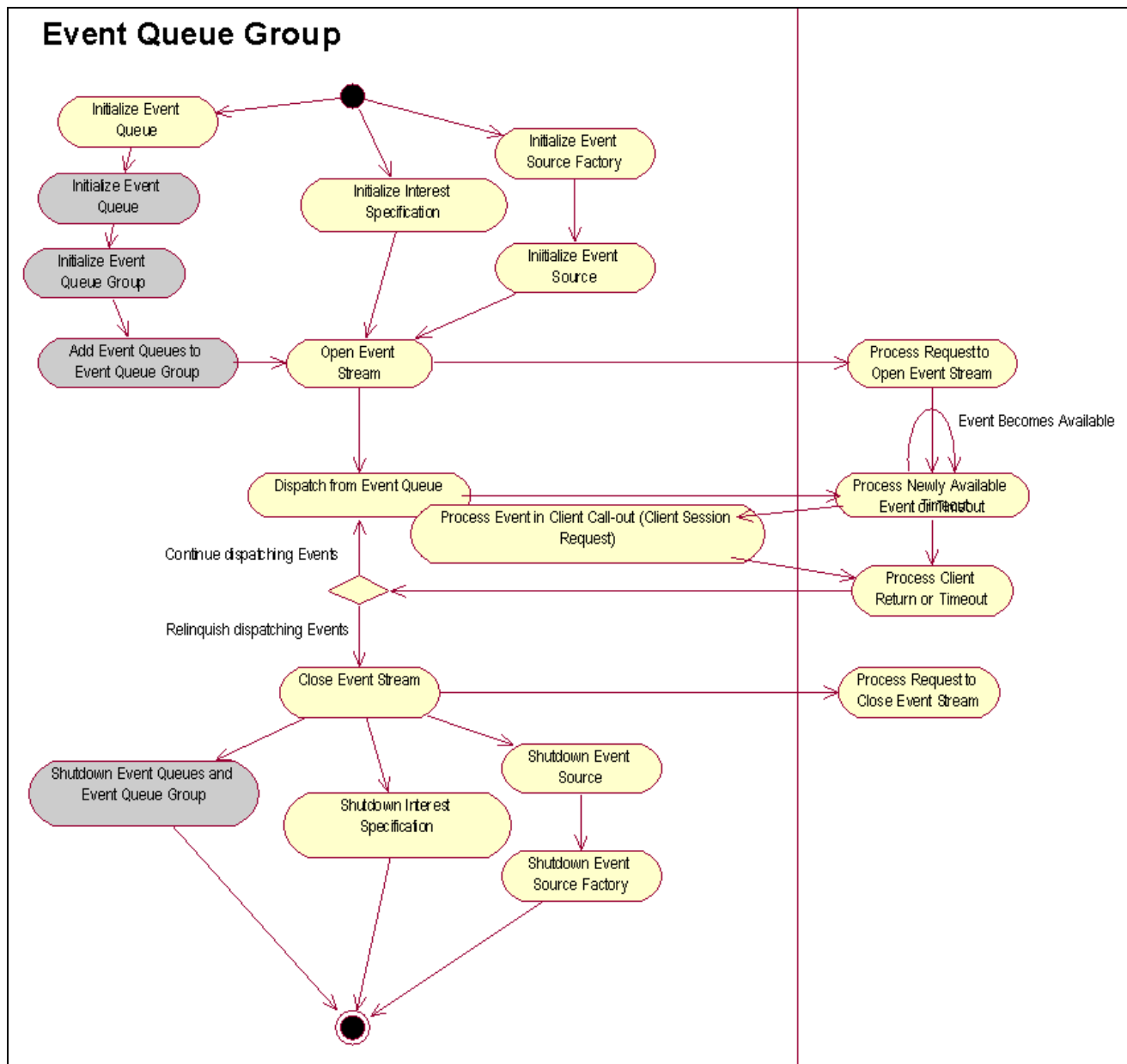


Figure 30: Event Distribution: Event Queue Group

An application initializes an Event Queue Group by adding individual Event Queues to the Event Queue Group. When shutting down an Event Queue Group, an application need only destroy the Event Queue Group. RFA destroys any Event Queues that the application added to the Event Queue Group. The preceding diagram modifies Figure 30 by showing additional Event Queue Group activities as shaded icons.

The following example shows the sequence to initialize an Event Queue Group.

```
// --- Initialize Event Queue Group ---
// Create the first EventQueue
EventQueue eventQueue1 = EventQueue.Create( "DemoEventQueue1" );

// Create the second EventQueue
EventQueue eventQueue2 = EventQueue.Create( "DemoEventQueue2" );

// Create the EventQueueGroup
EventQueueGroup eventQueueGroup = EventQueueGroup.Create( "DemoEventQueueGroup" );

// Add the first EventQueue to the EventQueueGroup
eventQueueGroup.AddEventQueue( eventQueue1, EventQueueGroup.QueuePriorityEnum.High );

// Add the second EventQueue to the EventQueueGroup
eventQueueGroup.AddEventQueue( eventQueue2, EventQueueGroup.QueuePriorityEnum.Low );

// Set the algorithm to dispatch from higher priority EventQueues before dispatching from
// lower priority EventQueues.
eventQueueGroup.Algorithm = EventQueueGroup.AlgorithmTypeEnum.HIGH_TO_LOW ;
```

Example 5: Initialize Event Queue Group

In the above example, the application creates two Event Queues and one Event Queue Group. Next, it adds the Event Queues to the Event Queue Group by calling `AddEventQueue()`. The `AddEventQueue()` method accepts two parameters: the first is the Event Queue and the second is the priority given to the queue.

Finally, the application sets the `Algorithm` property, which accepts one parameter. This parameter defines the algorithm for dispatching the Events from the Event Queues. In this example, the application sets the priority to `EventQueueGroup.AlgorithmTypeEnum.HIGH_TO_LOW`. This implies that RFA dispatches all Events from higher priority Event Queues before dispatching any Events from lower priority Event Queues. The application could have also set the priority to `EventQueueGroup.AlgorithmTypeEnum.ABACAB`, which means that 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.

The following example depicts the sequence to shut down an Event Queue Group. As indicated above, an application need only destroy the Event Queue Group. RFA will destroy Event Queues added to the Event Queue Group. Optionally, the application can call `RemoveEventQueue()` and re-use or destroy the Event Queue independently.

```
// --- Shutdown Event Queue Group ---
eventQueueGroup.Destroy();
```

Example 6: Shutdown Event Queue Group

The application could have optionally called `Deactivate()` on the Event Queue prior to calling to `Destroy()`. The `Deactivate()` method prevents any more Events from being put onto the Event Queue. If the application desires, it could dispatch any remaining Events before destroying the Event Queue. It must not `Destroy()` an Event Queue while it has active event streams.

5.2.1.3 Using Notification Client

A Notification Client is a callback 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. In this case, 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 typically uses a Notification Client for event loop integration with threading notifications. The Application uses the

`EventQueue.RegisterNotificationClient()` interface to inform RFA which objects are used as notification mechanisms for each event queue. The notification client registration should happen before registering for any events.

The thread context used to trigger a Notification Client is RFA's internal Session thread. Each RFA Session contains a single Session thread used to govern request and response traffic between the application and threads that interact with the underlying internal APIs.

The Notification callback code is called by RFA for every event, so its use has potential performance considerations in a high-throughput situation.

Warning! Because the Notification Client is called from within the context of the Event Distribution mechanism, 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 Name property. No other methods on any of the RFA interfaces should be called, directly or indirectly. The application should also not perform any CPU-intensive processing or make any calls that can block for any significant period of time because it may negatively affect internal processing in RFA.

The following diagram alters Figure 29 by depicting additional Notification Client activities as shaded icons.

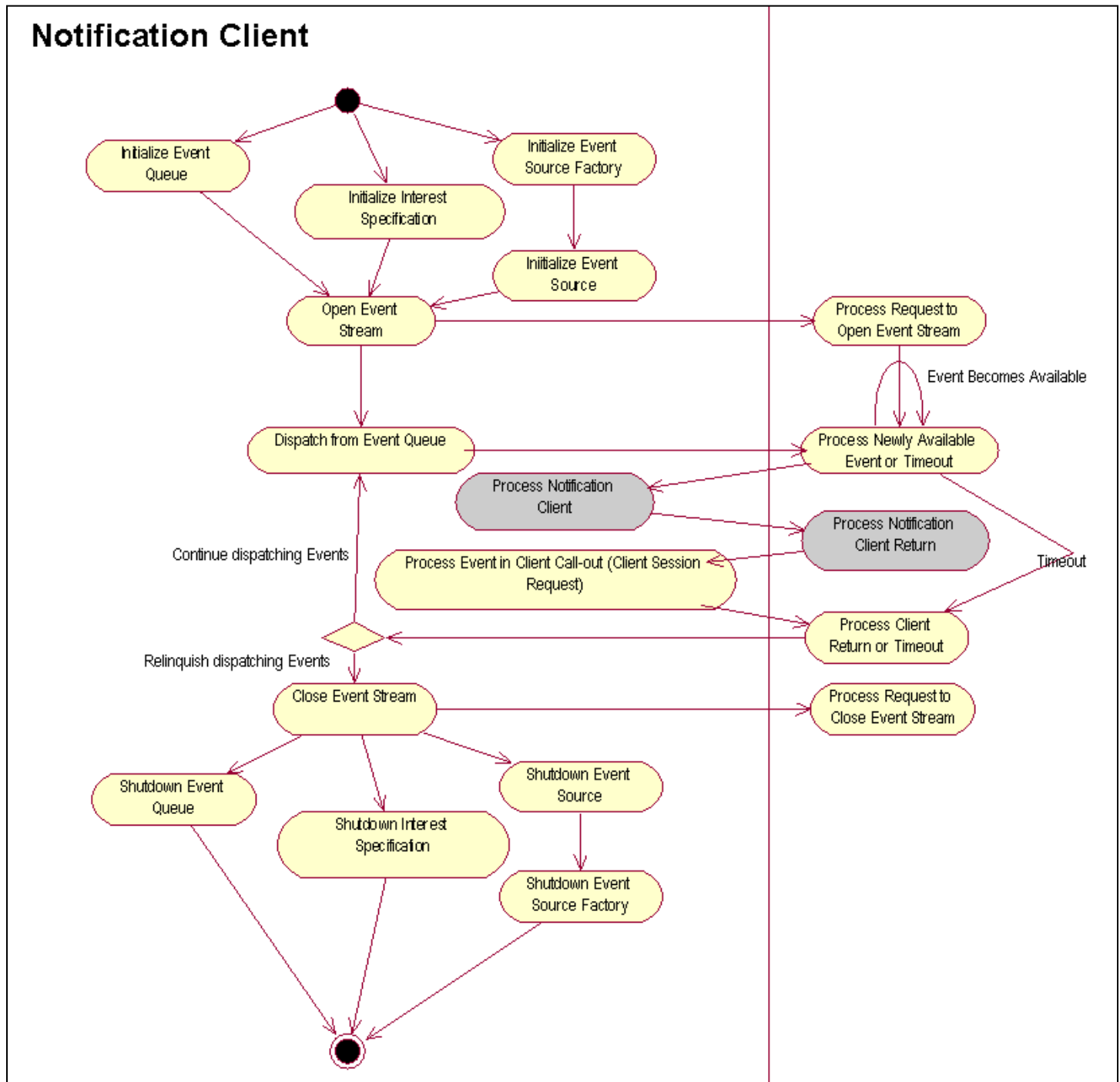


Figure 31: Notification Client

An example of a Notification Client callback method is as follows:

```
// --- Receive Notification - MySimpleNotificationClient ---

public class MySimpleNotificationClient : DispatchableNotificationClient
{
    public void Notify( Dispatchable dispSource, Object closure )
    {
        Console.WriteLine( "Notify() has been invoked with " + dispSource.Name.ToString() );
    }
}
```

Example 7: Process Notification Client

In this example, the application defines the class `MySimpleNotificationClient` by deriving from the `DispatchableNotificationClient` interface. The application implements the `Notify()` callback method, which will be invoked by RFA.

5.2.1.4 Using Event Distribution Model in a Single-thread Context

Using the Event Distribution Model in a single thread context implies that 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 steps other than those described in previous sections. There are two cases in which an event stream can be closed:

- 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.
- RFA dispatches no more Events following indication that the Event Stream has been closed via the `IsEventStreamClosed` property as described in Section 5.2.1.5.

5.2.1.5 Using Event Distribution Model in a Multiple-thread Context

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 one additional Event.

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 as described in Section 5.1.1.5. 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 close the Event Stream. To use Completion Events, an application should specify this option to be true when initializing an Event Source.

This is useful for knowing, among other things, when an application may safely clean up an event source. If one thread deletes an event source immediately after the return from closing the Event Stream, RFA may attempt to call `ProcessEvent()` on the deleted object from another thread, which will crash the application.

Other than specifying the option to use Completion Events, an application does not typically need to perform any special processing for a Completion Event. A Completion Event merely always returns true when the application calls the `IsEventStreamClosed` property. Once the application receives true from `IsEventStreamClosed` it is guaranteed to receive no more Events for this Event Stream.

Some applications may desire to differentiate a Completion Event 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 enumeration is `CommonEventTypeEnum.Completed`, available in the Common Package.

It is important to note that if a thread is blocked on `Dispatch()` when an Event Stream closes, the thread will always be released. When using Completion Events, the Completion Event causes the release of thread. When not using Completion Events, RFA releases the blocked thread regardless. Additionally, RFA also releases a thread blocked on `Dispatch()` when an alternate thread calls `Deactivate()`.

The RFA Package includes the ThreadedExample, which make uses of Completion Events in multiple thread contexts.

5.2.1.6 Event Queue with Statistics

An Event Queue with Statistics allows applications to monitor the number of events in the Event Queue and retrieve the length of time that events are queued. An Event Queue with statistics will notify the application when the number of events in the event queue has exceeded a configurable maximum threshold.

If an application dispatches more slowly than the rate of inbound messages, the Event Queue will grow boundlessly and will eventually run out of memory. Using this feature can be useful for knowing when that limit is approaching and gracefully handling the situation.

Event Queues with statistics do not apply to Event Queue Groups.

Using an Event Queue with statistics negatively impacts performance due to the overhead of messages being timestamped and additional checking for thresholds.

5.2.1.6.1 Maximum Depth Configuration

An application can be notified that an event queue has reached the maximum depth.

Applications can set the `EventQueueMaxDepth` property to specify the maximum number of events allowed in the event queue. The maximum depth value must be greater than 0. (A setting of 0 disables the functionality, and the queue behaves like a normal event queue.)

To keep the application from exhausting memory, all events, including those for generic messages, are discarded after the event queue's maximum depth is reached. Any events already enqueued before the event queue maximum depth is reached remain in the queue. To process remaining events, the application can call `dispatch` as usual.

It is the application's responsibility to perform any recovery after reaching the event queue maximum depth. During the recovery process, the application must call `ResetEventQueue()` to tell the event queue to restart queuing events. Otherwise the event queue will not restart queuing events again.

An applications must register to receive maximum depth notifications. The maximum depth notification is generated when messages are added to the event queue.

Warning! If an application needs to retain all events, the application should not enable the event queue maximum depth functionality.

5.2.1.6.2 High/Low Event Queue Threshold/Maximum Depth Notification

The application can set the low and high threshold values of the event queue with `SetEventQueueThreshold()`. The parameters passed apply to when the number of events in the queue is increasing or decreasing, respectively. The high threshold must be less than the maximum depth and the low threshold must be less than the high threshold. They must both be greater than 0 to enable the functionality.

When a threshold is reached, only one notification is provided regardless of how many times the threshold is passed. This holds true until the other threshold is reached at some future point. This is to prevent an application from being flooded with notifications around a threshold point.

For example, if the high depth threshold is passed, one notification is issued to inform the user of the high threshold breach. If data rates decrease and the low threshold is later crossed, a single notification will be sent to now inform the user of the low threshold breach.

Applications must register to receive event queue threshold and maximum depth notifications. The high threshold notification is generated when messages are added to the event queue; the low threshold notification is generated during a dispatch method.

NOTE: Using the existing `RegisterNotificationClient()` method is equivalent to enabling high threshold notification with a high threshold value of 1. It might not make sense to use both to call the dispatch method if the application is using both as notifications.

The event queue with statistics provides a mechanism to avoid repeatedly triggering threshold notifications when the number of events in the queue oscillates around the high or low threshold value. The following figure illustrates this behavior.

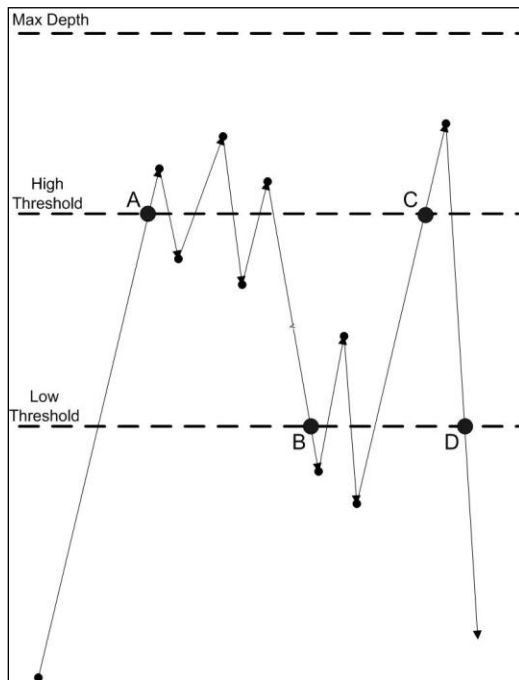


Figure 32: High/Low Threshold Behavior

A threshold notification is generated when the number of queued events reaches the high threshold (point A). This notification is not generated again until the number of queued events first falls to the low threshold (point B). This means that subsequent crossings of the high threshold value do not cause additional notifications unless the number of events in the queue first reaches the low threshold value and later bounces back up to the high threshold value.

If the low threshold notification is enabled, a low threshold notification is generated at point B. A low threshold notification is not generated again until a new high threshold is reached (point C). Subsequent crossings of the low threshold value do not result in additional notifications unless the number of events reaches the high threshold value and later drops back down to the low threshold value (point D).

5.2.1.7 Event Distribution Usage Guideline

5.2.1.7.1 Event Queues and Event Queue Group

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 `Dispatch()` on an individual Event Queue that is a member of a group. The application should also not 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 `Destroy()` 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.

5.2.1.7.2 Using Event Queues and Event Queue Groups from Multiple Thread

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 multicore processors or multiprocessor machines.

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

5.2.1.7.3 Event Handler Restriction

The application does not control the life cycle of an Event object received via `ProcessEvent()`. The application should never call `Destroy()` on the Event object or any other object that can be obtained using the `Event` interface property such as `EventQueue`, `EventSource`, etc. However, once an application clones an Event object, it then controls the lifecycle of that cloned Event object and needs to destroy it after it has finished using it. The following methods can not be called from the `ProcessEvent()` call:

- Any of the methods or properties of an `EventQueue` or an `EventQueueGroup`, except `Name` property.
- The `Destroy()` method of an Event Source associated with the current Event.

Applications may make new requests from within `ProcessEvent()`.

Before deleting an instance of an Event Handler, the application must ensure that the Client instance will not be used in a `Dispatch()` call. The rules for ensuring this are different for single- and multi-threaded applications.

- The single-threaded application must cancel all currently active requests that were made with the instance of the Client specified. Then it can delete the Client instance.
- The multi-threaded application must use Completion Events. The application should cancel all currently active requests that were made with the instance of the Client specified and wait for a Completion Event for each of those requests. Then it can delete the Client instance.

Alternately (for both cases), the application can deactivate all Event Queues or Event Queue Groups with which the Client instance is associated, then delete the Client instance.

5.2.2 Context

The Context class integrates multiple RFA packages into a single application and coordinates the interactions between them. This is necessary due to the interdependencies between the different RFA packages. For example, the Logger Package is dependent on the Configuration Package and the Common Package.

An application must initialize the `Context` interface before using any other RFA Interfaces. The application must un-initialize the `Context` when it will no longer use any RFA Interfaces. If an application has several independent application components it may initialize the `Context` as many times as needed, provided it un-initializes the `Context` once for each time that the `Context` has been initialized.

The `Initialize()` method always returns `true`. The `Uninitialize()` method will return `false` if the number of the `Initialize()` and `Uninitialize()` calls did not match, or the internal shutdown process has failed. The `Uninitialize()` will return `true` if the `Context` is un-initialized successfully.

To initialize the Context, the application simply calls the static `Initialize()` method as follows:

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

Example 8: Initialize Context

To uninitialize the `Context` the application simply calls the static `Uninitialize()` method as follows:

```
// --- Initialize RFA Context ---
Context.Uninitialize();
```

Example 9: Uninitialize Context

`Context.Uninitialize()` is blocking method.

Warning! Do not call these methods from within a callback method. To avoid problems, the application must call these methods within one of its own threads.

5.2.2.1 RFA Version Info

`Context` also provides a `RFAVersionInfo` property to retrieve the build product version of RFA .NET edition and RFA C++ edition. The following example assumes the application has initialized `Context` and depicts using `RFAVersionInfo` property:

```
// --- Get RFA Product version ---
RFA_String productVersion = new RFA_String();
productVersion = Context.RFAVersionInfo.ProductVersion;
Console.WriteLine( "RFA Product Version:" + productVersion.ToString() );
```

Example 10: Get RFA Product Version

NOTE: The version is only available if an application has acquired a Session (i.e., the Session Layer library is loaded).

The `Context` also provides versioning information about the packages and other dependent libraries. An application can use the `GetPackageVersion()` method to retrieve versioning information about an individual package or dependent library. An application can use the `GetPackageNames()` method to get the names of the all currently loaded packages and dependent libraries. The following example assumes the application has initialized the `Context` and depicts using the `GetPackageVersion()` method and `PackageNames` property:

```
// --- Get Package Versions ---
// Obtain the vector of package names
List<RFA_String> packageNames = Context.PackageNames;

// Iterate over the package names, extracting the version numbers
for (int i = 0; i < packageNames.Count; i++)
{
    RFA_String packageName = packageNames[i];
    RFA_String packageVersion = Context.GetPackageVersion(packageName);
    Console.WriteLine("Package:" + packageName.ToString() + " Version:" + packageVersion.ToString());
}
```

Example 11: Get Package Versions

In this example, the application obtains a list of package names via `PackageNames` property. Next, the application iterates over the returned list of `RFA_String`, extracting the version of each package via `GetPackageVersion()`.

The `Context` also provides a `Name` property, which always returns the string "RFA." Alternate RFA packages use this name for configuration and logging (see Chapter 8 and Chapter 10 for details on this string).

NOTE: RFA supports just-in-time loading of libraries to minimize its footprint in some environments. Package names and versions are only available for the libraries that have been loaded.

5.2.3 Buffer

Because one of the goals of the RFA is to provide low-level access to data, it deals with unstructured raw data in many cases. The `Buffer` class provides simple encapsulation of a raw data as an object that contains both the data and the length of that data. Since the `Buffer` is length-specified it does not need to be null terminated. It can contain any binary data.

The `Buffer` supports typical operations such as copy and compare. A simple example of `Buffer` usage is as follows:

```
// --- Create and print buffer ---

byte[] charBuff = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Buffer demoBuffer = new Buffer(charBuff, charBuff.Length);
Console.WriteLine("demoBuffer:" + demoBuffer.ToString());
```

Example 12: Buffer Usage

In this example, the application simply creates a `Buffer` specifying a character buffer (`charBuff`) and length of the character buffer (`charBuff.Length`). Next, the application outputs the contents of the buffer.

5.2.4 RFA_String and RFA_WString

The [RFA_String](#) class is a simple encapsulation of non-binary data that accepts length-specified or null terminated data. It is a simple container that emulates many methods of [System.String](#). For example, it provides methods to set, retrieve, compare, find, modify, and clear strings, among others. For more information on [RFA_String](#) interfaces, refer to the *RFA Reference Manual .NET Edition*.

The [RFA_WString](#) class has the same functionality as [RFA_String](#), except that it supports Unicode characters. Conversion from Unicode characters must be implemented in [RFA_WString](#) and [RFA_String](#).

NOTE: [RFA_String](#) does support Unicode characters which are able to be converted to UTF-8 encoding.

The following example briefly displays the usage of the [RFA_String](#) class.

```
// --- Create and print an RFA_String ---

byte[] charBuff = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
RFA_String demoString = new RFA_String();
demoString.Set( charBuff, (uint) charBuff.Length );
Console.WriteLine("demoString: " + demoString.ToString());
```

Example 13: RFA_String Usage

In the example above, the application creates an [RFA_String](#). Using [Set\(\)](#), the content are initialized by specifying the string, length of the string. Contents in the [RFA_String](#) are accessed using [ToString\(\)](#) and output to the screen.

An application can use the [ToString\(\)](#) method to get the content in [RFA_String](#). The return type of this method is [System.String](#).

[RFA_String](#) implements [System.IEquatable](#) to provide a [Equals\(\)](#) method for indicating whether the [RFA_String](#) object is equal to another [RFA_String](#) object.

[RFA_String](#) implements [System.IComparable](#) to provide a [CompareTo\(\)](#) method for comparing the current [RFA_String](#) instance with another [RFA_String](#) object and returns an integer that indicates whether the current instance precedes, follows, or occurs in the same position in the sort order as the other [RFA_String](#) object.

5.2.4.1 RFA_String Performance Consideration

- When specifying the length of the string in the [RFA_String](#) constructor or the [Set\(\)](#) method, a non-zero length specifies the known length of the string. If the string is null terminated, then the length includes the null byte. A length of zero will cause the [RFA_String](#) class to do a [strlen\(\)](#) to get the length of the string. If it is not specified, the length defaults to zero.
- The string specified in the constructor or the [Set\(\)](#) method does not need to be null terminated.
- Each invocation of [Append\(\)](#), [Equality\(\)](#), or [Addition\(\)](#) will probably result in a heap allocation.
- [RFA_String](#) provides both [Append\(\)](#) method and [Operator+](#) interface for append [System.String](#) or [RFA_String](#) to itself. RFA recommend to use [Append\(\)](#) method to improve performance of the application because the handle will not be changed.
- RFA provides constructor for [RFA_String](#) to create the empty string, and create string with the given contents. In case that the application has content for set to [RFA_String](#) already, RFA recommend to create [RFA_String](#) with the given content.

5.2.5 RMTEConverter

The [RMTEConverter](#) interface is useful for converting the encoded RMTE string payload received as part of the OMM data to a Unicode string. It helps display news in international languages with UCS2 format or transfer data through the network in ISO 2022 or UTF-8 format. The [RMTEConverter](#) interface provides the following major functions:

- [SetBuffer\(\)](#): binds a buffer to a converter (with a heap allocation). If the offset passed into the function is -1, the buffer passed in contains full field data and RFA will internally cache the buffer in memory. If the offset is equal to or greater than 0, the buffer passed in contains partial field data and RFA will internally apply the partial field data on the earlier cached buffer based on offset. The cached buffer should contain full field data. For more information about how to get the offset, see Section 6.1.11.5.
- [GetAsByteString\(\)](#): returns a UTF-8-formatted string as an [RFA_String](#).
- [GetAsCharString\(\)](#): returns a UCS2-formatted string in [List<char>](#). Each item in [List<char>](#) is a char which holds a character, and the size of the list is the total numbers of characters.

```
RMTEConverter conv = new RMTEConverter();
PartialUpdateReadIterator partialReadIt = new PartialUpdateReadIterator();

// ...

if ( dataBuffer.IsPartialUpdates )
{
    partialReadIt.Start( dataBuffer.GetBuffer() );

    while ( !partialReadIt.Off() )
    {
        fieldUpdatedBuf = partialReadIt.Value;
        offset = partialReadIt.Offset;
        conv.SetBuffer( fieldUpdatedBuf, offset );
        partialReadIt.Forth();
    }

    // For update msg display
    val = conv.GetAsCharString();
}
else
{
    conv.SetBuffer( dataBuffer.GetBuffer() );

    // For refresh msg display
    val = conv.GetAsCharString();
}
```

Example 14: RMTEConverter Usage

5.2.6 Exception

The Common Package also defines an Exception interface. All exceptions thrown by the RFA inherit from this Exception interface.

The most important methods exposed by the [Exception](#) interface are [Severity](#) and [Classification](#) property. The [Severity](#) property specifies the severity of the exception. The severity is defined as follows:

- [CommonErrorSeverityTypeEnum.Error](#) indicates an unrecoverable event such as missing required configuration parameters.
- [CommonErrorSeverityTypeEnum.Warning](#) indicates a recoverable event such as a loss of a connection to a back-end system.
- [CommonErrorSeverityTypeEnum.Information](#) indicates general information such as debugging information.
- The [Classification](#) property specifies the classification of the exception. The classification is as follows:
- [CommonErrorClassificationTypeEnum.Anticipated](#) indicates an error for which the API can commonly recover gracefully.
- [CommonErrorClassificationTypeEnum.Internal](#) indicates an internal program bug and theoretically should be fully preventable.
- [CommonErrorClassificationTypeEnum.External](#) indicates an error due to an external system or physical resource.
- [CommonErrorClassificationTypeEnum.IncorrectAPIUsage](#) indicates an unanticipated error due to incorrect usage of the API. The API cannot recover gracefully from these errors.
- [CommonErrorClassificationTypeEnum.ConfigurationError](#) indicates a missing configuration.

A number of interfaces inherit from the exception interface. The [InvalidUsageException](#) interface indicates invalid usage of the RFA (e.g., invalid input parameter, invalid call sequence). The [InvalidConfigurationException](#) indicates missing or incorrect configuration parameter. The [OutOfRangeException](#) interface is used by the Data package and indicates an error when a data value does not fit into an [Int](#) or [UInt](#).

Each interface that inherits from [Exception](#) provides a [Status](#) property. Type of this property is a [GeneralExceptionStatus](#) that indicates the general status of the interface that threw the exception. If the status is [GeneralExceptionStatus.StateEnum.OperationFailed](#), the interface is still useable if the problem is corrected. If the status is [GeneralExceptionStatus.StateEnum.InvalidObject](#), the interface should no longer be used. Applications can call the [StatusCode](#) property to get more information.

An example showing rudimentary exception handling is shown below; for the full example see the [CheckException\(\)](#) method in the consumer example distributed with RFA.

```
public static void CheckException(R Reuters.RFA.Common.Exception e)
{
    // Determine exception Severity
    RFA_String excpSeverityStr;
    switch (e.Severity)
    {
        case Reuters.RFA.Common.Exception.CommonErrorSeverityTypeEnum.Error:
            excpSeverityStr = new RFA_String("Error");
            break;
        case Reuters.RFA.Common.Exception.CommonErrorSeverityTypeEnum.Warning:
            excpSeverityStr = new RFA_String("Warning");
            break;
        case Reuters.RFA.Common.Exception.CommonErrorSeverityTypeEnum.Information:
            excpSeverityStr = new RFA_String("Information");

            break;
        default:
```

```

        excpSeverityStr = new RFA_String("UNKNOWN Severity");
        break;
    }

    // Determine exception Classification
    RFA_String excpClassificationStr;
    switch (e.Classification)
    {
        case Reuters.RFA.Common.Exception.CommonErrorClassificationTypeEnum.Anticipated:
            excpClassificationStr = new RFA_String("Anticipated");
            break;
        case Reuters.RFA.Common.Exception.CommonErrorClassificationTypeEnum.Internal:
            excpClassificationStr = new RFA_String("Internal");
            break;
        case Reuters.RFA.Common.Exception.CommonErrorClassificationTypeEnum.External:
            excpClassificationStr = new RFA_String("External");
            break;
        case Reuters.RFA.Common.Exception.CommonErrorClassificationTypeEnum.IncorrectAPIUsage:
            excpClassificationStr = new RFA_String("IncorrectAPIUsage");
            break;
        case Reuters.RFA.Common.Exception.CommonErrorClassificationTypeEnum.ConfigurationError:
            excpClassificationStr = new RFA_String("ConfigurationError");
            break;
        default:
            excpClassificationStr = new RFA_String("UNKNOWN Classification");
            break;
    }

    // Determine exception Type
    RFA_String excpTypeStr;
    RFA_String excpStatusText;
    RFA_String excpDetails = new RFA_String();

    switch (e.ErrorType)
    {
        case Reuters.RFA.Common.Exception.CommonErrorTypeEnum.InvalidUsageException:
        {
            excpTypeStr = new RFA_String("InvalidUsageException");
            InvalidUsageException actualException = e as InvalidUsageException;
            excpStatusText = actualException.Status.StatusText;
            break;
        }
        case Reuters.RFA.Common.Exception.CommonErrorTypeEnum.InvalidConfigurationException:
        {
            excpTypeStr = new RFA_String("InvalidConfigurationException");
            InvalidConfigurationException actualException = e as InvalidConfigurationException;
            excpStatusText = actualException.Status.StatusText;
            excpDetails = actualException.ParameterName + " ";
            excpDetails += actualException.ParameterValue;
            break;
        }
        case Reuters.RFA.Common.Exception.CommonErrorTypeEnum.SystemException:
        {
            excpTypeStr = new RFA_String("SystemException");

            Reuters.RFA.Common.SystemException actualException = e as
Reuters.RFA.Common.SystemException;
            excpStatusText = actualException.Status.StatusText;
            break;
        }
    }

```

```

        default:
            excpTypeStr = new RFA_String();
            excpStatusText = new RFA_String("Unknown Exception Type!");
            break;
    }

    // output exception information
    RFA_String tmpstr;
    tmpstr = new RFA_String("AN EXCEPTION HAS BEEN THROWN! The following information describes the
exception:");
    tmpstr.Append("\r\n");
    tmpstr.Append("Exception Type: ");
    tmpstr.Append(excpTypeStr);
    tmpstr.Append("\r\n");
    tmpstr.Append("Exception Severity: ");
    tmpstr.Append(excpSeverityStr);
    tmpstr.Append("\r\n");
    tmpstr.Append("Exception Classification: ");
    tmpstr.Append(excpClassificationStr);
    tmpstr.Append("\r\n");
    tmpstr.Append("Exception Status Text: ");
    tmpstr.Append(excpStatusText);
    tmpstr.Append("\r\n");
    if (!excpDetails.Empty())
    {
        tmpstr.Append(excpDetails);
        tmpstr.Append("\r\n");
    }

    AppUtil.Log(AppUtil.LEVEL.ERR, string.Format("ExceptionHandler.CheckException() EXCEPTION! {0}",
tmpstr.ToString()));
    MessageBox.Show(null, tmpstr.ToString(), "Exception Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}

```

Example 15: Exception Handling in RFA

5.2.7 Quality of Service

In network terminology, Quality of Service (QoS) is a means of classifying traffic into separate tiers to provide differentiated services within a network. With RFA, Quality of Service is a method of classifying services provided by the delivery infrastructure. QoS in RFA is divided into two dimensions:

- **Timeliness:** age of data
- **Rate:** period⁵ of change in data

Timeliness may have one of the following values:

- **Real-time** implies no delay is applied to the data.
- **Unspecified Delayed Timeliness** implies a fixed delay (e.g., 60 seconds) is applied to the data.
- **Unspecified Timeliness** implies the Timeliness has not been defined.
- Any positive number representing the actual delay in seconds

Rate may have one of the following values:

- **Tick-By-Tick** implies the application receives every update. The period (i.e., the time interval between successive events) varies based on the market activity.
- **Just-In-Time Filtered Rate** implies the period varies during different time intervals. In this case, RFA may deliver data with a fixed period at some time intervals while at other time intervals RFA may deliver data Tick-By-Tick.
- **Unspecified Rate** implies the Rate has not been defined.
- Any positive number representing the actual rate in milliseconds. These update will represent an aggregate of the updates that happened within that interval.

A Quality of Service always contains a single value in each dimension (e.g., **Timeliness = Real-time** and **Rate = Tick-By-Tick**). An application may check the actual QoS when receiving events.

As part of the service directory applications may receive `QualityOfServiceInfo` objects, which contain a `QoS` object and the number of concrete services that can provide that particular QoS. A `QualityOfServiceInfo` contains:

- **QualityOfService:** the actual QoS provided by the service.
- **Count:** 1 when the service is up, and 0 when the service is down. If the service is part of a service group, **Count** is the number of concrete services in the group that can provide that particular QoS.

⁵ Period is the time interval between two successive occurrences of a recurring event.

5.2.7.1 Quality of Service Request

An application requesting interest in an item can specify a desired QoS. The actual QoS it receives can be extracted from the event.

PARAMETER	VALUE
Best Timeliness	Real Time
Worst Timeliness	Delayed
Best Rate	Tick by Tick
Worst Rate	Slowest Rate

Table 17: Example Quality of Service Request

The goal for an application is to describe a range of values between Best and Worst of QoS that is acceptable. If the application wants a specific QoS it can set the Best values only.

Best and **Worst Timeliness** Request Properties may have one of two values: `QualityOfServiceRequest.RealTime` or `QualityOfServiceRequest.Delayed`. These values are used to set the upper and lower bounds for the timeliness dimension of QoS. If the application will only accept real-time QoS, both **Best** and **Worst Timeliness** will be set to `RealTime`. If the application doesn't have a preference, **Best** should be set to `QualityOfServiceRequest.RealTime`, and **Worst** should be set to `QualityOfServiceRequest.Delayed`. It follows that if the application only wants delayed; both **Best** and **Worst Timeliness** should be set to `QualityOfServiceRequest.Delayed`. If RFA is unable to provide the specified Timeliness (e.g., due to lack of availability or entitlements), it rejects the request.

Best and **Worst Rate** Request Properties may have one of the following values:

`QualityOfServiceRequest.TickByTick`, `QualityOfServiceRequest.FastestFilteredRate`, `QualityOfServiceRequest.JustInTimeFilteredRate`, `QualityOfServiceRequest.SlowestRate`, or any positive number reflecting the actual minimum period of change in data. These are the same as defined above with the exception of `QualityOfServiceRequest.FastestFilteredRate`, which implies 1 millisecond conflated updates, and `QualityOfServiceRequest.SlowestRate`, which implies the rate is unspecified through configuration (i.e., `"UnspecifiedRate"`).

In addition to these attributes, the application can also specify Stream Property whether the QoS should change following the initial item image. It has two values: **Static** or **Dynamic**:

- **Static** implies the QoS will not change following the initial item image. RFA determines the QoS of the initial image based on entitlements and availability.
- **Dynamic** implies the QoS may change after the initial item image. RFA determines the QoS of both the initial item image and subsequent updates (or images) based on entitlements, availability and network congestion.

5.2.7.2 Quality of Service Item Event

The application can obtain the actual QoS in the form of a `QualityOfService` present on the item event.

The **Timeliness** property indicates whether RFA is providing Real Time (`QualityOfService.RealTime`), Unspecified Timeliness (`QualityOfService.UnspecifiedTimeliness`), Unspecified Delayed Timeliness (`QualityOfService.UnspecifiedDelayedTimeliness`), or any positive number.

The **Rate** property indicates whether RFA is providing Tick-By-Tick (`QualityOfService.TickByTick`), Just-In-Time-Filtered (`QualityOfService.JustInTimeFilteredRate`), Unspecified Rate (`QualityOfService.UnspecifiedRate`), or any positive number.

5.2.7.3 Quality of Service Recovery

After the concrete service is chosen from a service group based on the requested QoS, RFA may have to switch to another service within the same service group. The service that RFA switches to is dependent on the Stream property of the requested QoS. If the Stream property is Static, RFA will only recover to any concrete service within the same service group that provides the exact Quality of Service that it was originally consuming as.⁶ If the Stream property is Dynamic, RFA will recover to any concrete service within the same service group that can provide a QoS in the requested range. The triggers that will cause RFA to switch are:

- **Recovery** when a concrete service in use becomes unavailable and a new one must be found.
- **Re-permission** when a DACS lock change is performed such that permission in use is taken away or when a DACS profile changes that can potentially expand or shrink the set of QoSs available to the user.

5.2.7.3.1 Recovery

When a concrete service or connection goes down, or when an item is marked `RespStatus.StreamStateEnum.Closed` or `RespStatus.StreamStateEnum.ClosedRecover` by the back end, RFA will attempt to connect to another concrete service within the same service group that will continue providing the item at a specific QoS per the static/dynamic constraints described above.

When RFA receives a trigger signaling that it needs to perform recovery logic, the item will be placed in a recovering state. RFA will periodically check to see if a concrete service within the same service group is available and able to provide the QoS needed. If one becomes available, the item will be moved to the requested state, and item events will resume.

5.2.7.4 Default Value

PROPERTY NAME	VALUE
BestTimeliness	RealTime
WorstTimeliness	Delayed
BestRate	TickByTick
WorstRate	SlowestRate
StreamProperty	Static

Table 18: Quality of Service Request Defaults

The default values above provide the most preferred QoS or lesser QoS depending on availability, entitlements and network congestion.⁷ An application may change these parameters to specify a different QoS.

⁶ This is regardless of the requested Quality of Service.

⁷ The default parameters provide QoS-unaware applications the ability to receive different QoS's based on available infrastructure and configuration.

5.2.7.5 Quality of Service Scenario

There may be times when the actual QoS is downgraded from the requested QoS, such as when a QoS range is provided, QoS is specified as Dynamic, and the following conditions occur:

- The desired QoS is not available on the network.
- The desired QoS is available, but the user is not entitled to it.
- The initial QoS provided can no longer be provided due to availability, entitlements or network congestion.

Table 19 depicts RFA's reaction to certain requested qualities of service. Note that the behavior depends on whether the service is obtainable (i.e., the service was not denied due to availability, entitlements, or network congestion).

REQUESTED QUALITY OF SERVICE	BEHAVIOR
Best Rate = TickByTick Worst Rate = TickByTick Best Timeliness = RealTime Worst Timeliness = RealTime	Attempts to find a service that provides exact real-time, Tick-By-Tick QoS.
Best Rate = TickByTick Worst Rate = SlowestRate Best Timeliness = RealTime Worst Timeliness = Delayed	Attempts to find a service that provides any QoS based on the order of precedence.
Best Rate = TickByTick Worst Rate = SlowestRate Best Timeliness = Delayed Worst Timeliness = Delayed	Attempts to find a service that provides any delayed QoS based on the order of precedence.
Best Rate = TickByTick Worst Rate = SlowestRate Best Timeliness = Delayed Worst Timeliness = RealTime	An exception is thrown when the Best is not better than the Worst.
Best Rate = FastestFilteredRate Worst Rate = SlowestRate Best Timeliness = RealTime Worst Timeliness = Delayed	Attempts to find any QoS Rate excluding TickByTick and any QoS Timeliness based on the order of precedence.
Best Rate = SlowestRate Worst Rate = SlowestRate Best Timeliness = RealTime Worst Timeliness = Delayed	Attempts to find a worst QoS Rate and any QoS Timeliness based on the order of precedence.

Table 19: Quality of Service: Request Parameters vs. Behavior

Chapter 6 Data Package

6.1 Data Package Concepts

The purpose of the **Data Package** is to provide interfaces for encoding data in OMM form to RWF form, and decoding data in RWF to OMM.

The OMM representation of data is used for programmatic manipulation, while the RWF representation of data is a more condensed form used to transmit the data on the wire. Encoded data can be transmitted more efficiently on the wire. Decoded data which has been converted from RWF to an OMM model can be manipulated more easily programmatically.

The Data Package provides interfaces for concrete representations of the [Data](#) interface defined in Common namespace. The interfaces are encapsulated in the Messages that flow between the service provider and the consumer. An application may use the data interfaces to access the actual content. The diagram below shows the relation between data types defined by the Data Package and the Data interface.

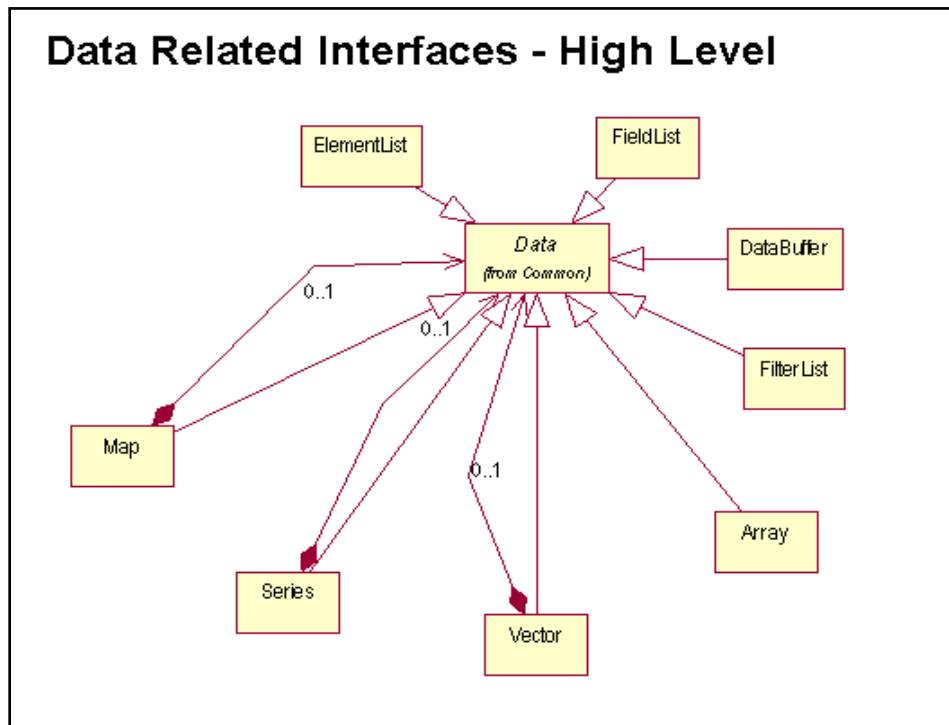


Figure 33: High-level Data relationships

The Data Package adds new data types and provides the corresponding read and write iterator interfaces to encode and decode containers and entries. Write iterators are used to encode the data and read iterators are used to decode the data.

The Data Package does not provide encoders/decoders for certain data types like Opaque, XML, and ANSI. These data types require external encoders/decoders.

The concepts related to the Data Package are listed in the following table.

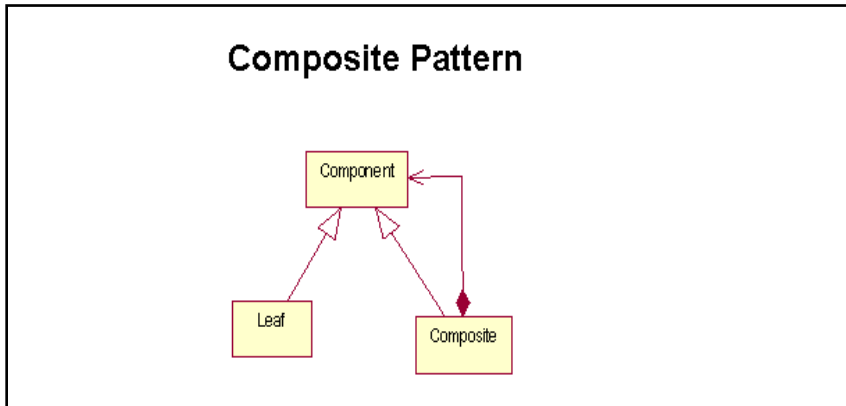
CONCEPT	DESCRIPTION
Data	Represents the actual content
Payload Data	The data that is contained in the message. For example, in Level I data, the Payload Data contains price discovery information.
Summary Data	Metadata that describes a Container's Entries. For example, Summary Data could specify the currency of each Entry's price or rules regarding the re-sorting of Entries.
Attribute Info	Data that typically identifies the associated Payload Data.
Permission Data	Authorization information with respect to a login context.
Leaf	<p>The simplest form of Data and represents the base data types.</p> <p>Only one type of Leaf is supported and is referred to as the Data Buffer. It may contain the following:</p> <ul style="list-style-type: none"> • Simple types like Int, UInt • Simple interfaces like Date, Time etc. • Data types requiring external parsers; e.g., XML, ANSI, etc.
Container	A more complex data representation than primitive data which contains Entries e.g., FilterList, Vector, Series etc.
Primitive	The lowest-level types of OMM data—e.g., Int, UInt, Real, Date, DateTime—that can be stored in containers.
Entry	The data contained within a Container, e.g., Field, Vector Entry; the Entry may contain another Container or a Leaf.
Nested Data	A Container contained within another Container to form a hierarchy.
Non-uniform Nesting	The nesting layer within a Container containing different types of nested data (e.g., a Map of FilterLists).
Uniform Nesting	The nesting layer within a Container containing the same type of nested data (e.g., a FieldList of FieldLists).
Data Decoding	Process of obtaining the actual content (raw data) represented by the data structures.
Data Encoding	Converting actual content (raw data) into data structures provided by the Data Package.
Iterators	A utility to step through the data and access individual entries.
Read Iterator	A utility to step through and decode data.
Write Iterator	A utility to step through and encode data.
Single Iterator	A single instance of a SingleReadIterator or SingleWriteIterator that can decode or encode multiple types multiple nested levels of data.
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.
Actions	Operations on Entries within a Container to manage update processing and fragments reconstruction. A Provider may specify actions. A Consumer needs to comply with the actions.
Standard Data	Data that unites entry content and entry definition, represented as one entity. 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 data formats.
Defined Data	<p>Data that separates entry content and entry definition. The content and definition is 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>Defined Data prevents the application from managing a dictionary of definitions and reduces bandwidth consumption through the ability to distribute entry definitions only once. Bandwidth consumption is further reduced in the case of fixed-width data, as distribution of the width must occur only once.</p>
Def ID	The Identifier referencing the Definition

CONCEPT	DESCRIPTION
Entry Definition	Description of the contained data (content) within an Entry
Entry Content	Contained data within an Entry

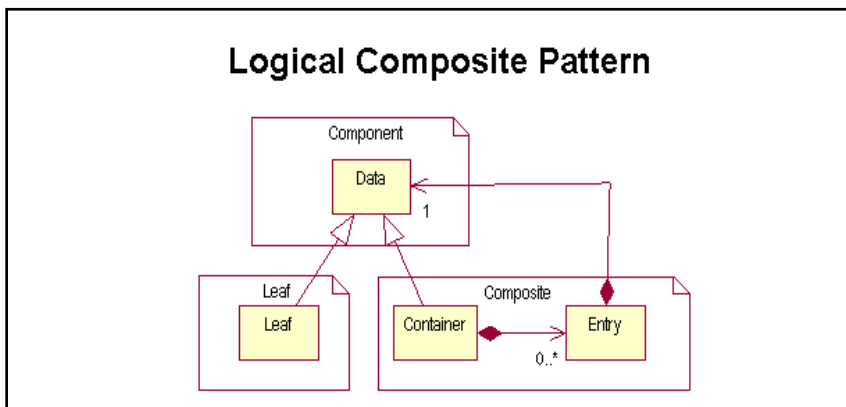
Table 20: Data Package Concepts

6.1.1 Logical Composite Pattern

Several interfaces in the Data Package closely resemble those of a Composite Pattern depicted in the figure below:

**Figure 34: Composite Pattern**

- Data resembles a Composite Pattern's Component. Data forms the common ancestor of several other Data Package interfaces, such as the Message Package interfaces and select Data Package interfaces.
- Leaf resembles a Composite Pattern's Leaf. Leaf includes simple types resembling numerics, reals, date, time, status, and QoS. The specific types within the leaf may be scoped to the Data Package or the Common Package.
- Both a Container and its contained Entries jointly resemble a Composite Pattern's Composite. They realize constructs such as field identifier value pairs, associative key value pairs or self-describing named value pairs. The Data Package includes iterator interfaces to encode and decode these Containers and Entries.

**Figure 35: Logical Composite Subsystem**

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 hierarchy.

Several methods in the Message and Data Packages may contain Data. This is not only to realize a nesting hierarchy but also to append branched hierarchies in an existing nesting hierarchy (e.g., to append additional metadata).

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. The table below provides high-level descriptions of the Leaf and Container types in RFA.

6.1.2 Data Uniformity

Some Containers house Entries that each contain Uniform, or Homogeneous, Data. Other Containers house Entries that each contain Non-uniform, or Heterogeneous, Data. Non-uniform Data contained in each Entry may differ, while uniform Data contained in each Entry is of the same type.

Some Containers may also house Summary Data. Summary data is always uniform with respect to payload data in the Entries. For example, if a Container's Entries contain payload data of type [FieldList](#) then the summary data of the Container must also be a [FieldList](#).

Whether payload, key, or summary data, uniform data always has a slightly higher bandwidth optimization than non-uniform data. This is because the type of contained data only needs to be encoded once.

A Consumer application typically does not need to be concerned whether the Container houses either uniform or non-uniform data. This is because a Consumer application does not usually encode Data.

However, a Provider application typically needs to be concerned whether a Container and its housed Entries contain uniform or non-uniform types of Data. For Containers that house Entries with uniform Data, the Provider application needs to ensure that the encoding of subsequent types of Data match the original type of Data. If the Provider application attempts to encode a different type, the API handles this situation as an error condition. Thus interface methods enforce their support for uniformity. This is true for payload, key and summary data.

A breakdown of container types and the uniformity of data they contain is shown below:

CONTAINER	UNIFORMITY OF CONTAINED DATA
Array	Uniform
ElementList	Uniform, Non-uniform
FieldList	Uniform, Non-uniform
FilterList	Uniform, Non-uniform
Map	Uniform
Series	Uniform
Vector	Uniform

Table 21: Data Container Uniformity

6.1.3 Data Types and Data Containers

RFA classifies all data types into primitives and containers. In general, a container requires further parsing to read, while a primitive can be read directly without additional parsing. A data item's classification specifies where it can be used. Section 6.1.3.1 depicts Data and DataBuffer type classification, while Section 6.1.3.2 shows where they can be contained. For a full list of primitives supported by RFA, see [DataBufferEnum](#) in [DataBuffer](#) class.

NOTE: OMM preserves the precision of encoded numeric values. Note that some conversions from OMM numeric types to primitive types (e.g., [Real](#) to [Float](#)) may result in a loss of precision. This is an case of a narrowing precision conversion. Also note that because the IEEE Standard for Floating-Point Arithmetic (IEEE 754) cannot represent some values exactly, rounding may occur when manually converting from the OMM representation to other data types, or converting via the provided utility methods.

6.1.3.1 Data Type Classification

TYPE	CLASSIFICATION	DATATYPE – DATABUFFERTYPE	INFORMATION
Int	Primitive	DataBuffer – Int	An 8-byte signed integer. Uses System.Int64 .
UInt	Primitive	DataBuffer – UInt	An 8-byte unsigned integer. Uses System.UInt64 .
Float	Primitive	DataBuffer – Float	A 4-byte value that has a range of -3.4e38 to +3.4e38 with an accuracy of 6 to 7 decimal digits. This value is compliant with the IEEE-754 standard. Uses System.Single . (See note following this table)
Double	Primitive	DataBuffer – Double	An 8-byte value that has a range of -1.7e308 to +1.7e308 with an accuracy of 14 to 15 digits. This value is compliant with the IEEE-754 standard. Uses System.Double . (See note following this table)
Real	Primitive	DataBuffer – Real	An 8-byte precision ⁸ (19-20 decimal places) fixed-placed representation of a numeric having a fractional or exponential part. The range of the fractional part is from $\frac{1}{2}$ to $\frac{1}{256}$. The range of the exponential part is from 10^{-14} to 10^7 . Uses Reuters.RFA.Data.Real . (See note following this table) The Reuters.RFA.Data.Real also supports infinity, negative infinity, and Not a Number (NaN) values. The application can use enumeration values defined in Reuters.RFA.Data.MagnitudeTypeEnum to set these values with Infinity, NegInfinity, and NotANumber.
Date	Primitive	DataBuffer – Date	A 4-byte value that represents a Gregorian date. Uses Reuters.RFA.Data.Date .
Time	Primitive	DataBuffer – Time	A 3, 5, 7, or 8-byte value that includes information for hours, minutes, seconds, and optional milliseconds, microseconds, and nanoseconds. Uses Reuters.RFA.Data.Time .
DateTime	Primitive	DataBuffer – DateTime	An 7, 9, 11, or 12-byte combination of a Date and a Time. Uses Reuters.RFA.Data.DateTime .
QualityOfService	Primitive	DataBuffer – QualityOfService	Indicates Quality Of Service Info. Quality Of Service Info contains a set of Quality Of Services (along with instance counts). Uses Reuters.RFA.Common.QualityOfServiceInfo .
RespStatus	Primitive	DataBuffer – RespStatus	Indicates a RespStatus. Uses Reuters.RFA.Common.RespStatus .
Enumeration	Primitive	DataBuffer – Enumeration	A 2-byte signed value that can be expanded to a language-specific string in an EnumTable Dictionary. Uses System.Int16 .
Buffer	Primitive	DataBuffer – Buffer	Indicates a Buffer. Uses Reuters.RFA.Common.Buffer .
StringASCII	Primitive	DataBuffer – StringASCII	Indicates 8-bit characters encoding using the Reuters Basic Character Set (RBCS). The first 128 characters are equivalent to the ASCII character set (ANSI X3.4-1968). Uses Reuters.RFA.Common.RFA_String .
StringUTF8	Primitive	DataBuffer – StringUTF8	Indicates a UTF-8 encoding of ISO 10646 (specified in Section 3.9 of the Unicode 4.0 standard and IETF's RFC 3629). Uses Reuters.RFA.Common.Buffer .

⁸ For Defined Data, fewer bytes may be used if the full precision is not needed.

TYPE	CLASSIFICATION	DATATYPE – DATABUFFERTYPE	INFORMATION
StringRMTEs	Primitive	DBus – StringRMTEs	Indicates an encoding with the A Multilingual Text Encoding Standard (RMTEs). RMTEs uses ISO 2022 escape sequences to select the character sets used. RMTEs provides support for RBCS, UTF-8, Japanese Latin and Katakana (JIS C 6220 - 1969), Japanese Kanji (JIS X 0208 - 1990), and Chinese National Standard (CNS 11643-1986). StringRMTEs also supports RREP sequences for character repetition and RHPA sequences for partial updates. Uses Reuters.RFA.Common.Buffer .
AnsiPage	Container	DBus – AnsiPage	Indicates AnsiPage. The RFA AnsiPage package can decode and encode AnsiPage with no required dictionary. Uses Reuters.RFA.Common.Buffer .
Opaque	Container	DBus – Opaque	Indicates a buffer is present and the type or structure should be agreed upon between the Provider and Consumer. Uses Reuters.RFA.Common.Buffer .
XML	Container	DBus – XML	Indicates XML-formatted data. Uses Reuters.RFA.Common.Buffer .
Array	Primitive	Array	A container of ordered entries. An Array can contain zero to N ⁹ primitive type entries , with zero indicating an empty Array.
ElementList	Container	ElementList	A container of flexible, self-describing named entries. An Element List can contain zero to N ¹⁰ entries, with zero indicating an empty Element List.
FieldList	Container	FieldList	A container of efficient, field-identifier, value-paired entries. The field identifier references attributes such as name and type in a field definition dictionary. A FieldList can contain zero to N ¹¹ entries, with zero indicating an empty FieldList.
FilterList	Container	FilterList	A container of loosely coupled entries. A FilterList can contain zero to N ¹² entries, with zero indicating an empty FilterList, though this type is typically limited by the number of available of filterId values.
Map	Container	Map	A container of key-value paired Entries. A Map can contain zero to N ¹³ entries, where zero entries indicate an empty Map. A single fragment of a Map may contain up to 64K Entries.

⁹ 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.

¹⁰ 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.

¹¹ 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.

¹² 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.

¹³ 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.

TYPE	CLASSIFICATION	DATATYPE – DATABUFFERTYPE	INFORMATION
Msg	Container	Msg	A container housing an OMM message.
Series	Container	Series	A container of uniform Entries, often used to represent table-based information, where no explicit indexing is present or required. A Series can contain zero to N ¹⁴ entries, where zero entries indicates an empty Series.
Vector	Container	Vector	A container of position-oriented, index-value, paired entries. Each entry's index is represented by an unsigned integer with a value between 0 and 2 ³⁰ . A single fragment of a Vector may contain 64K Entries. A Vector can contain zero to N ¹⁵ entries, where zero entries indicates an empty Vector.

Table 22: Data Type Classification

¹⁴ 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.

¹⁵ 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.

6.1.3.2 Data Containers

ENTRY TYPE	CONTAINS PRIMITIVES	LOGICAL COMPOSITE
ArrayEntry	Yes (except Array)	No
ElementEntry	Yes	Yes (except Msg)
FieldEntry	Yes	Yes (except Msg)
FilterEntry	No	Yes (except Msg)
MapEntry Key	Yes	No
MapEntry Value	No	Yes (except Msg)
AttribInfo Attrib	No	Yes (except Msg)
Msg Payload	No	Yes
SeriesEntry	No	Yes (except Msg)
VectorEntry	No	Yes (except Msg)

Table 23: Data Type Containment

6.1.4 Standard Data and Defined Data

Standard Data is Data that combines its Entry content and Entry definition. For Standard Data, the type of the contained Data may reside either within the Entry or in a dictionary. All Containers support Standard Data. One key benefit of Standard Data is its ease of use. Another benefit of Standard Data is its similarity with conventional market price data formats such as MarketFeed.

Defined Data is Data that separates Entry content and Entry definition. Thus the contained Data and Entry identifier are separated into discrete segments. A Data Definition defines the overall layout of data in the container which each entry conforms to. This eliminates the need for type information to be specified for each Entry in the container. The result is that the type information only needs to be transmitted once. This allows additional optimizations to be performed that result in fewer bytes transmitted on the network.

One key benefit of Defined Data is that it can considerably reduce bandwidth consumption, particularly for cases when the data structure continually repeats such as in the case of Level II, Time Series and Data Dictionaries. Additionally, Defined Data can further reduce bandwidth consumption in the case of fixed-width data, as distribution of the width must only occur once.

Typically, Consumer applications need to only know a few subtle differences between Standard Data and Defined Data. This is because the Data Package coherently presents Defined Data and Standard Data to an application that decodes data (i.e., typically a Consumer application).

Provider applications typically need to know the distinction between Standard Data and Defined Data in order to independently specify the Entry content and Entry definition, whereas Consumers usually only need to know a few subtle differences.

Containers which are able to house the definition portion of Defined Data (e.g. Map, Vector, Series) may have several definitions, each referenced by an identifier known as a [DefID](#). Containers able to house the Entry content portion of Defined Data may have zero or one structure of Entry content, also referenced by a [DefID](#).

Data Definitions and Defined Data are described in more detail in Section 6.1.9.

6.1.5 Container Property Breakdown

The table below depicts the above properties across all containers. For more information about fragmentation see Section 6.1.11.1.

CONTAINER TYPES	STANDARD DATA	DEFINED DATA	FRAGMENTATION
Filterlist	Optional	No	Optional
ElementList	Optional	Optional Set Defined Data	No
FieldList	Optional	Optional Set Defined Data	No
Series	Optional	Optional Set Definition(s)	Optional
Map	Optional	Optional Set Definition(s)	Optional
Vector	Optional	Optional Set Definition(s)	Optional
Array	Optional	No	No

Table 24: Container Properties

6.1.6 Leaf Properties

6.1.6.1 Blanking

Blanking is the indication of an empty `DataBuffer` with a specified type. For example, a Consumer application could react to a blank by clearing its respective displayed value.

A Provider application may specify a blank `DataBuffer` by using `SetBlankData()`. A Consumer application may determine a blank by using the `IsBlank()` property.

Some `DataBuffer` types support a blank while others do not. Additionally, some `DataBuffer` types may only support a blank in the case of Standard Data and not Defined Data.

NOTE: Blank does not imply that the application should clear the cache; the Message Package provides the `ClearCache` on the `RespMsg` for this purpose.

6.1.6.2 Array Widths

The Array container may house variable-width or fixed-width types. Variable-width implies the data contained in the Array Entries may be of different sizes, whereas fixed-width implies the data contained in the Array Entries are of the same size.

The benefit of variable-width size is the flexibility of different sized encoding, while the benefit of fixed-width size is the ability to minimize bandwidth and to manipulate the entire Array as a transparent buffer (i.e., able to operate on contained data entries via the fixed-size offset).

6.1.6.3 Encoded Types

An Encoded Type is the true packing on the wire. Consumer applications do not usually need to be concerned about Encoded Types because the Data Package presents all Encoded Types as a `DataBuffer` type. If desired, the Consumer application may identify the true Encoded Type via a specific method on the `DataBuffer` interface.

- For Standard Data, all `DataBuffer` encoded types are Length Specified (packed with a length indicating the size).
- For Set Defined Data, `DataBuffer` types can use `DataBufferEncodedEnum`, the purpose of which is to reduce bandwidth consumption (by not encoding the actual size on the wire) at the potential expense of a loss of flexibility (e.g., the loss of the full range of a numeric type). Applications should use the `FieldEntryDef` or `ElementEntryDef` interfaces to set the encoded type for an entry.

Encoded Types include:

- Length Specified: packed with a length indicating the size.
- Real Length: packed to save 1 byte on the wire at the expense that the Consumer may incur a slight impact on performance due to the Consumer implementation needing to partially decode the value even if the Consumer application has no desire for the value.
- `Bit8Value`, `Bit16Value`, `Bit32Value`, `Bit64Value`: packed to the specified number of bits. Range is limited per the specified number of bits.

6.1.7 Entry Properties

6.1.7.1 Permission Data

Permission Data is optional authorization information with respect to a login context. Typically applications use Permission Data to manipulate authorization locks. For more information see the *DACSLOCK API Developers Guide .NET Edition* and *DACSLOCK API Reference Manual .NET Edition*.

Typically, Permission Data is at stream scope. In this case Permission Data is accessible via the Manifest in the Message.

Some Entries also support Permission Data, which is available at Entry scope for applications that desire finer grain authorization. In this case, Permission Data is accessible via an Entry interface.

6.1.7.2 Rippling

The [FieldList](#) also supports rippling. Rippling is a property of a [FieldEntry](#) that replaces the value with a newly received value and propagates the former value to an alternate [FieldEntry](#). Rippling reduces bandwidth consumption since the Provider need not again distribute former values. Rather, the Consumer propagates the former value to the alternate FieldEntry as identified through a dictionary. It is a responsibility of the Consumer to ripple the [FieldEntry](#)s. RFA does not ripple any [FieldEntry](#)s on behalf of the application.

Though some [FieldEntry](#)s have the rippling property, specific [IndicationMaskFlag.DoNotRipple](#) values inhibit rippling. For details on specific RespTypeNum values that inhibit rippling, refer to the *AnsiPage Developers Guide .NET Edition*.

6.1.7.3 Entry Properties Comparison

The table below lists the above properties across all Entry types.

ENTRY	IDENTIFIER	IDENTIFIER TYPE	PERM DATA
FilterEntry	FilterID	byte	Optional
ElementEntry	Name	String	No
FieldEntry	FieldID	short	No
SeriesEntry	Implicit order	N/A	No
MapEntry	Key	Any Primitive	Optional
VectorEntry	Position	uint	Optional
ArrayEntry	Implicit order	N/A	No

Table 25: Entry Type Properties

6.1.7.4 Entry Actions

Actions are operations on Entries within a Container to manage changes (e.g., updates and status) and reconstruction of fragments. A Provider may specify actions. A Consumer needs to comply with the actions. Possible Actions are listed below in the table.

Filter, Map, and Vector Entries have explicit actions while others have implicit actions. Entries supporting explicit actions contain an enumeration—e.g., `VectorEntry.ActionEnum.Delete`—indicating the explicit action, while entries supporting implicit actions have no enumeration. Some Entries support different actions in specific cases. For example, a sorted Vector supports more actions than does an unsorted Vector.

Consuming applications must comply with these change processing rules whether the actions are implicit or explicit:

- Whether the action occurs in a Refresh or Update message, always apply an Add action, even if the associated refresh message has not been received. (Integrity on Final Refresh indicates that updates and status messages can be received prior to the final refresh message). Subsequent Add actions for the same item should be applied instead of the previous Add action.
- Apply any other actions received for that item after the Add.

The following table provides a description of explicit actions and association to Entries that support them.

ACTION	DESCRIPTION	APPLICABLE ENTRIES	IMPACT ON OTHER ENTRIES
Set	Replace the entry	VectorEntry, FilterEntry	None
Add	Append or replace the entry	MapEntry	None
Update	Partial change of the entry	VectorEntry, FilterEntry, MapEntry	None
Delete	Remove the entry.	MapEntry	None
		VectorEntry (sorted only)	Decreases any higher-ordered position by one.
Insert	Place the entry between other entries	VectorEntry (sorted only)	Increases any higher-ordered position by one. May leave gaps if previous lower-ordered position is unpopulated.
Clear	Empty the entry (Contains no data)	VectorEntry, FilterEntry	None

Table 26: Explicit Actions

Some entries that support implicit actions may have different actions depending on the type of contained Data. The following table provides a description of implicit actions.

ACTION	DESCRIPTION	ENTRIES	CONTAINED DATA
Update	Partial change of the entry (Determined by the Data Type of Contained Data)	ElementEntry, FieldEntry	Any Data Type excluding Array or RMTES string
Set	Append or replace (Determined by the Data Type of Contained Data)	ElementEntry, FieldEntry	Any DataBufferType or Array excluding RMTES string
	Replace	ArrayEntry	Not dependent on contained data
	Append	SeriesEntry	Not dependent on contained data

Table 27: Implicit Actions

For Providers, some Entries can optionally be time filtered (i.e., cached) dependent on the action. The following table provides a description of explicit and implicit actions that support time filtering per Entry.

ACTION	ENTRIES	TIME FILTERING
Set	FilterEntry, VectorEntry	Optional per Entry. Ignore previous explicit action and forward.
Add	MapEntry	Optional per Entry. Ignore previous explicit action and forward.
Update	MapEntry, VectorEntry FilterEntry	Optional per Entry. Need comply with contained data actions.
Delete	MapEntry	Optional per Entry. Ignore previous explicit action and forward.
	VectorEntry (sorted only)	No, but optionally may be accrued into a single Vector instance.
Insert	VectorEntry (sorted only)	No, but optionally may be accrued into a single Vector instance.
Clear	FilterEntry, VectorEntry	Optional per Entry. Ignore previous explicit action and forward

Table 28: Time Filtering based on Action

6.1.8 Data Types in Depth

6.1.8.1 DataBuffer

The DataBuffer is a Leaf that contains simple types including Int, UInt, Date, Time, DateTime, and Real. This is the only Leaf supported.

Below are the DataBuffer properties.

PROPERTY NAME	DESCRIPTION																						
EncodedType	Type of the encoded DataBuffer. Forwarding data may require knowledge of the true encoded type. All possible values of DataBufferEncodedEnum are as follows:																						
	<table><tr><th>DATABUFFERENCODED ENUM</th><th>DESCRIPTION</th></tr><tr><td>LengthSpecified</td><td>Indicates a specified length of encoded value. The length specified is the native type used for accessor methods on the DataBuffer interface.</td></tr><tr><td>Bit8Value</td><td>Indicates an 8-bit encoded value.</td></tr><tr><td>Bit16Value</td><td>Indicates a 16-bit encoded value.</td></tr><tr><td>Bit24Value</td><td>Indicates a 24-bit encoded value.</td></tr><tr><td>Bit32Value</td><td>Indicates a 32-bit encoded value.</td></tr><tr><td>Bit40Value</td><td>Indicates a 40-bit encoded value.</td></tr><tr><td>Bit56Value</td><td>Indicates a 56-bit encoded value.</td></tr><tr><td>Bit64Value</td><td>Indicates a 64-bit encoded value.</td></tr><tr><td>Bit72Value</td><td>Indicates a 72-bit encoded value.</td></tr><tr><td>RealLength</td><td>Indicates a RealLength encoded value. Benefit of a RealLength is that it saves 1 byte on the wire at the expense that the Consumer may incur a slight impact on performance. A Real offers a RealLength EncodedType.</td></tr></table>	DATABUFFERENCODED ENUM	DESCRIPTION	LengthSpecified	Indicates a specified length of encoded value. The length specified is the native type used for accessor methods on the DataBuffer interface.	Bit8Value	Indicates an 8-bit encoded value.	Bit16Value	Indicates a 16-bit encoded value.	Bit24Value	Indicates a 24-bit encoded value.	Bit32Value	Indicates a 32-bit encoded value.	Bit40Value	Indicates a 40-bit encoded value.	Bit56Value	Indicates a 56-bit encoded value.	Bit64Value	Indicates a 64-bit encoded value.	Bit72Value	Indicates a 72-bit encoded value.	RealLength	Indicates a RealLength encoded value. Benefit of a RealLength is that it saves 1 byte on the wire at the expense that the Consumer may incur a slight impact on performance. A Real offers a RealLength EncodedType.
	DATABUFFERENCODED ENUM	DESCRIPTION																					
	LengthSpecified	Indicates a specified length of encoded value. The length specified is the native type used for accessor methods on the DataBuffer interface.																					
	Bit8Value	Indicates an 8-bit encoded value.																					
	Bit16Value	Indicates a 16-bit encoded value.																					
	Bit24Value	Indicates a 24-bit encoded value.																					
	Bit32Value	Indicates a 32-bit encoded value.																					
	Bit40Value	Indicates a 40-bit encoded value.																					
	Bit56Value	Indicates a 56-bit encoded value.																					
	Bit64Value	Indicates a 64-bit encoded value.																					
	Bit72Value	Indicates a 72-bit encoded value.																					
RealLength	Indicates a RealLength encoded value. Benefit of a RealLength is that it saves 1 byte on the wire at the expense that the Consumer may incur a slight impact on performance. A Real offers a RealLength EncodedType.																						

PROPERTY NAME	DESCRIPTION																												
DataBufferType	<p>Type of DataBuffer. The associated set accessor allows specification of type information after the actual value has been specified. All possible values of DataBufferEnum are as follows:</p> <table> <tr> <th>DATABUFFER ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>NoDataBuffer</td><td>Indicates neither the DataBuffer value nor DataBuffer type information is present.</td></tr> <tr> <td>UnknownDataBuffer</td><td>Indicates that DataBuffer value is present but no DataBuffer type information is present. On the DataBuffer interface, the application that retrieves a Buffer to extract the raw information Applications may use this value to obtain the contained data on a FieldEntry if the type is unknown. In this case the application retrieves the contained data as a DataBuffer that will be of type UnknownDataBuffer.</td></tr> <tr> <td>Int</td><td>An 8-byte signed integer.</td></tr> <tr> <td>UInt</td><td>An 8-byte unsigned integer.</td></tr> <tr> <td>Float</td><td>A 4-byte value that has a range of -3.4e38 to +3.4e38 with an accuracy of 6 to 7 decimal digits. This value is compliant with the IEEE-754 standard.</td></tr> <tr> <td>Double</td><td>An 8-byte value that has a range of -1.7e308 to +1.7e308 with an accuracy of 14 to 15 digits. This value is compliant with the IEEE-754 standard.</td></tr> <tr> <td>Real</td><td>An 8-byte precision (19-20 decimal places) fixed-placed representation of a numeric having a fractional or exponential part. For Defined Data, fewer bytes may be used if the full precision is not needed. The range of the fractional part is from 1/2 to 1/256. The range of the exponential part is from 10-14 to 10+7. . Scope is Reuters.RFA.Data.Real.</td></tr> <tr> <td>Date</td><td>A 4-byte value that represents a Gregorian date. . Scope is Reuters.RFA.Data.Date.</td></tr> <tr> <td>Time</td><td>A 3- or 5-byte value that includes the information for hours, minutes, seconds, and optional milliseconds. . Scope is Reuters.RFA.Data.Time.</td></tr> <tr> <td>DateTime</td><td>An 7- or 9-byte combination of a Date and a Time. Scope is Reuters.RFA.Data.DateTime.</td></tr> <tr> <td>QualityOfServiceInfo</td><td>Indicates Quality Of Service Info. Quality Of Service Info contains a set of Quality Of Services (along with instance counts). Scope is Reuters.RFA.Common.QualityOfServiceInfo.</td></tr> <tr> <td>RespStatus</td><td>Indicates a RespStatus. Scope is Reuters.RFA.Common.RespStatus.</td></tr> <tr> <td>Enumeration</td><td>A 2-byte signed value that can be expanded to a language specific string in an EnumTable Dictionary.</td></tr> </table>	DATABUFFER ENUM	DESCRIPTION	NoDataBuffer	Indicates neither the DataBuffer value nor DataBuffer type information is present.	UnknownDataBuffer	Indicates that DataBuffer value is present but no DataBuffer type information is present. On the DataBuffer interface, the application that retrieves a Buffer to extract the raw information Applications may use this value to obtain the contained data on a FieldEntry if the type is unknown. In this case the application retrieves the contained data as a DataBuffer that will be of type UnknownDataBuffer.	Int	An 8-byte signed integer.	UInt	An 8-byte unsigned integer.	Float	A 4-byte value that has a range of -3.4e38 to +3.4e38 with an accuracy of 6 to 7 decimal digits. This value is compliant with the IEEE-754 standard.	Double	An 8-byte value that has a range of -1.7e308 to +1.7e308 with an accuracy of 14 to 15 digits. This value is compliant with the IEEE-754 standard.	Real	An 8-byte precision (19-20 decimal places) fixed-placed representation of a numeric having a fractional or exponential part. For Defined Data, fewer bytes may be used if the full precision is not needed. The range of the fractional part is from 1/2 to 1/256. The range of the exponential part is from 10-14 to 10+7. . Scope is Reuters.RFA.Data.Real .	Date	A 4-byte value that represents a Gregorian date. . Scope is Reuters.RFA.Data.Date .	Time	A 3- or 5-byte value that includes the information for hours, minutes, seconds, and optional milliseconds. . Scope is Reuters.RFA.Data.Time .	DateTime	An 7- or 9-byte combination of a Date and a Time. Scope is Reuters.RFA.Data.DateTime .	QualityOfServiceInfo	Indicates Quality Of Service Info. Quality Of Service Info contains a set of Quality Of Services (along with instance counts). Scope is Reuters.RFA.Common.QualityOfServiceInfo .	RespStatus	Indicates a RespStatus. Scope is Reuters.RFA.Common.RespStatus .	Enumeration	A 2-byte signed value that can be expanded to a language specific string in an EnumTable Dictionary.
DATABUFFER ENUM	DESCRIPTION																												
NoDataBuffer	Indicates neither the DataBuffer value nor DataBuffer type information is present.																												
UnknownDataBuffer	Indicates that DataBuffer value is present but no DataBuffer type information is present. On the DataBuffer interface, the application that retrieves a Buffer to extract the raw information Applications may use this value to obtain the contained data on a FieldEntry if the type is unknown. In this case the application retrieves the contained data as a DataBuffer that will be of type UnknownDataBuffer.																												
Int	An 8-byte signed integer.																												
UInt	An 8-byte unsigned integer.																												
Float	A 4-byte value that has a range of -3.4e38 to +3.4e38 with an accuracy of 6 to 7 decimal digits. This value is compliant with the IEEE-754 standard.																												
Double	An 8-byte value that has a range of -1.7e308 to +1.7e308 with an accuracy of 14 to 15 digits. This value is compliant with the IEEE-754 standard.																												
Real	An 8-byte precision (19-20 decimal places) fixed-placed representation of a numeric having a fractional or exponential part. For Defined Data, fewer bytes may be used if the full precision is not needed. The range of the fractional part is from 1/2 to 1/256. The range of the exponential part is from 10-14 to 10+7. . Scope is Reuters.RFA.Data.Real .																												
Date	A 4-byte value that represents a Gregorian date. . Scope is Reuters.RFA.Data.Date .																												
Time	A 3- or 5-byte value that includes the information for hours, minutes, seconds, and optional milliseconds. . Scope is Reuters.RFA.Data.Time .																												
DateTime	An 7- or 9-byte combination of a Date and a Time. Scope is Reuters.RFA.Data.DateTime .																												
QualityOfServiceInfo	Indicates Quality Of Service Info. Quality Of Service Info contains a set of Quality Of Services (along with instance counts). Scope is Reuters.RFA.Common.QualityOfServiceInfo .																												
RespStatus	Indicates a RespStatus. Scope is Reuters.RFA.Common.RespStatus .																												
Enumeration	A 2-byte signed value that can be expanded to a language specific string in an EnumTable Dictionary.																												

PROPERTY NAME	DESCRIPTION																
	<table> <tr> <th>DATABUFFER ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>Buffer</td><td>Indicates a Buffer. Scope is Reuters.RFA.Common.Buffer.</td></tr> <tr> <td>StringASCII</td><td>Indicates an 8-bit characters encoding using the Refinitiv Basic Character Set (RBCS). RBCS is also used for Marketfeed. The first 128 characters are equivalent to the ASCII character set (ANSI X3.4-1968). Scope is Reuters.RFA.Common.RFA_String.</td></tr> <tr> <td>StringUTF8</td><td>Indicates a UTF-8 encoding of ISO 10646 (specified in section 3.9 of the Unicode 4.0 standard and IETF's RFC 3629). Scope is Reuters.RFA.Common.Buffer.</td></tr> <tr> <td>StringRMTEs</td><td>Indicates an encoding with the A Multilingual Text Encoding Standard. RMTEs uses ISO 2022 escape sequences to select the character sets used. RMTEs provides support for the Refinitiv Basic Character Set, UTF-8, Japanese Latin and Katakana (JIS C 6220 - 1969), Japanese Kanji (JIS X 0208 - 1990), and Chinese National Standard (CNS 11643-1986). StringRMTEs also supports RREP sequences for character repetition and RHPA sequences for partial updates. Scope is Reuters.RFA.Common.Buffer.</td></tr> <tr> <td>Opaque</td><td>Indicates a buffer is present and type should be agreed upon between Provider and Consumer. Scope is Reuters.RFA.Common.Buffer.</td></tr> <tr> <td>XML</td><td>Indicates XML. Scope is Reuters.RFA.Common.Buffer.</td></tr> <tr> <td>ANSI_Page</td><td>Indicates AnsiPage. The RFA AnsiPage package can decode and encode AnsiPage, with no required dictionary. Scope is Reuters.RFA.Common.Buffer.</td></tr> </table>	DATABUFFER ENUM	DESCRIPTION	Buffer	Indicates a Buffer. Scope is Reuters.RFA.Common.Buffer .	StringASCII	Indicates an 8-bit characters encoding using the Refinitiv Basic Character Set (RBCS). RBCS is also used for Marketfeed. The first 128 characters are equivalent to the ASCII character set (ANSI X3.4-1968). Scope is Reuters.RFA.Common.RFA_String .	StringUTF8	Indicates a UTF-8 encoding of ISO 10646 (specified in section 3.9 of the Unicode 4.0 standard and IETF's RFC 3629). Scope is Reuters.RFA.Common.Buffer .	StringRMTEs	Indicates an encoding with the A Multilingual Text Encoding Standard. RMTEs uses ISO 2022 escape sequences to select the character sets used. RMTEs provides support for the Refinitiv Basic Character Set, UTF-8, Japanese Latin and Katakana (JIS C 6220 - 1969), Japanese Kanji (JIS X 0208 - 1990), and Chinese National Standard (CNS 11643-1986). StringRMTEs also supports RREP sequences for character repetition and RHPA sequences for partial updates. Scope is Reuters.RFA.Common.Buffer .	Opaque	Indicates a buffer is present and type should be agreed upon between Provider and Consumer. Scope is Reuters.RFA.Common.Buffer .	XML	Indicates XML. Scope is Reuters.RFA.Common.Buffer .	ANSI_Page	Indicates AnsiPage. The RFA AnsiPage package can decode and encode AnsiPage, with no required dictionary. Scope is Reuters.RFA.Common.Buffer .
DATABUFFER ENUM	DESCRIPTION																
Buffer	Indicates a Buffer. Scope is Reuters.RFA.Common.Buffer .																
StringASCII	Indicates an 8-bit characters encoding using the Refinitiv Basic Character Set (RBCS). RBCS is also used for Marketfeed. The first 128 characters are equivalent to the ASCII character set (ANSI X3.4-1968). Scope is Reuters.RFA.Common.RFA_String .																
StringUTF8	Indicates a UTF-8 encoding of ISO 10646 (specified in section 3.9 of the Unicode 4.0 standard and IETF's RFC 3629). Scope is Reuters.RFA.Common.Buffer .																
StringRMTEs	Indicates an encoding with the A Multilingual Text Encoding Standard. RMTEs uses ISO 2022 escape sequences to select the character sets used. RMTEs provides support for the Refinitiv Basic Character Set, UTF-8, Japanese Latin and Katakana (JIS C 6220 - 1969), Japanese Kanji (JIS X 0208 - 1990), and Chinese National Standard (CNS 11643-1986). StringRMTEs also supports RREP sequences for character repetition and RHPA sequences for partial updates. Scope is Reuters.RFA.Common.Buffer .																
Opaque	Indicates a buffer is present and type should be agreed upon between Provider and Consumer. Scope is Reuters.RFA.Common.Buffer .																
XML	Indicates XML. Scope is Reuters.RFA.Common.Buffer .																
ANSI_Page	Indicates AnsiPage. The RFA AnsiPage package can decode and encode AnsiPage, with no required dictionary. Scope is Reuters.RFA.Common.Buffer .																
String	Select types that are converted to a string representation. The associated set accessor converts select types from a string representation.																
Buffer	Data as a Buffer .																
Int	Data as an integer. The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit64Value.																
UInt	Data as an unsigned integer. The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit64Value.																
Float	Data as a Float . The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit32Value.																
Double	Data as a Double . The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit64Value.																
Real	Data as a Real . The associated set accessor allows specification of select encodedTypes of LengthSpecified or RealLength. The benefit of a RealLength is that it saves 1 byte on the wire at the expense that the Consumer may incur a slight impact on performance.																
Date	Data as a Date . The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit32Value.																
Time	Data as a Time . The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit32Value.																

PROPERTY NAME	DESCRIPTION
DateTime	Data as a DateTime . The associated set accessor allows specification of select encodedTypes of LengthSpecified or Bit64Value.
QualityOfServiceInfo	Data as a QualityOfService Info.
RespStatus	Status associated with the response. RespStatus is typically available only on Refresh or Status.
Enumeration	Data as an Enumeration.

Table 29: DataBuffer properties

6.1.8.2 Array

An [Array](#) is a simple Container of ordered Entries each known as an [ArrayEntry](#). The Entry has no identifier but rather dependent on the implicit order within the Container.

An [Array](#) can contain zero to 65,535 primitive type entries, where zero entries indicates an empty [Array](#). 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.

Entries can be varied- or fixed-width. In the later case, an application may optionally access the entire set of entries via the [EncodedBuffer](#) property and index them directly.

Below is the [Array](#) structure.

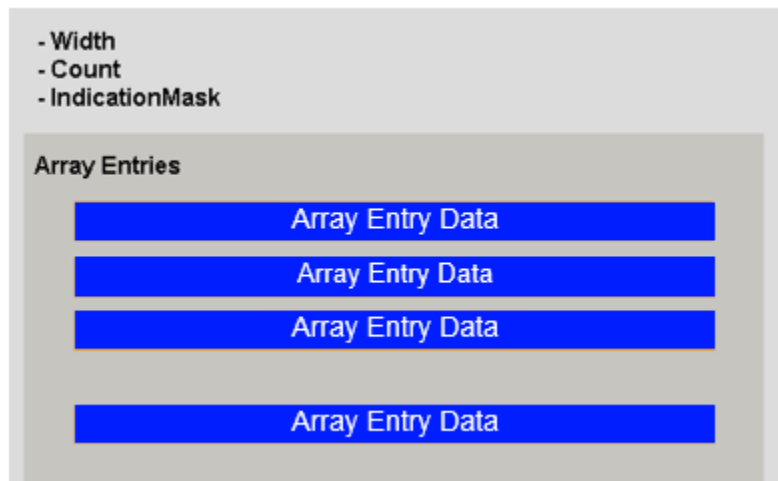


Figure 36: Array Structure

The table below shows the specific [Array](#) and [ArrayEntry](#) properties.

PROPERTY NAME	DESCRIPTION
Count	Returns the number of ArrayEntries.
IndicationMask	Returns a mask indicating the properties on the Array . The only possible value for IndicationMaskFlag is FixedWidth , which indicates fixed-width data contained in ArrayEntries. Its default value is 0 (Unlimit) .
Width	Returns the width of the data within the ArrayEntries for FixedWidth Arrays. The width is dependent on the specific type (and EncodedType) of data within the ArrayEntries.

Table 30: Array properties

PROPERTY NAME	DESCRIPTION
Data	The contained Data. For an entry obtained from the read iterator, the Session Layer internally populates the contained Data. Once the application specifies the Data property on any entry instance, the Data property returns a reference to the data. In this case, the reference is only valid if the data has not gone out of scope. Encoded Range: $0 \leq x < 64K$

Table 31: Array Entry properties

6.1.8.3 FieldList

A [FieldList](#) is a Container of identifier/value pairs Entries each known as a [FieldEntry](#). The identifier of Entry is a number. The contained Entry Data supports non-uniform types.

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.

For Standard Data, the identifier references the type attribute from a field definition dictionary. For Defined Data, the Defined Data definition contains the type attribute. Whether Standard Data or Defined Data, a field definition dictionary may also include other attributes such as maximum size and conversion information.

Fields are encoded with length. This allows decoding of Fields without the need of a field definition dictionary; however, interpretation of the Field requires a field definition dictionary.

Figure 37 illustrates the **FieldList** structure.

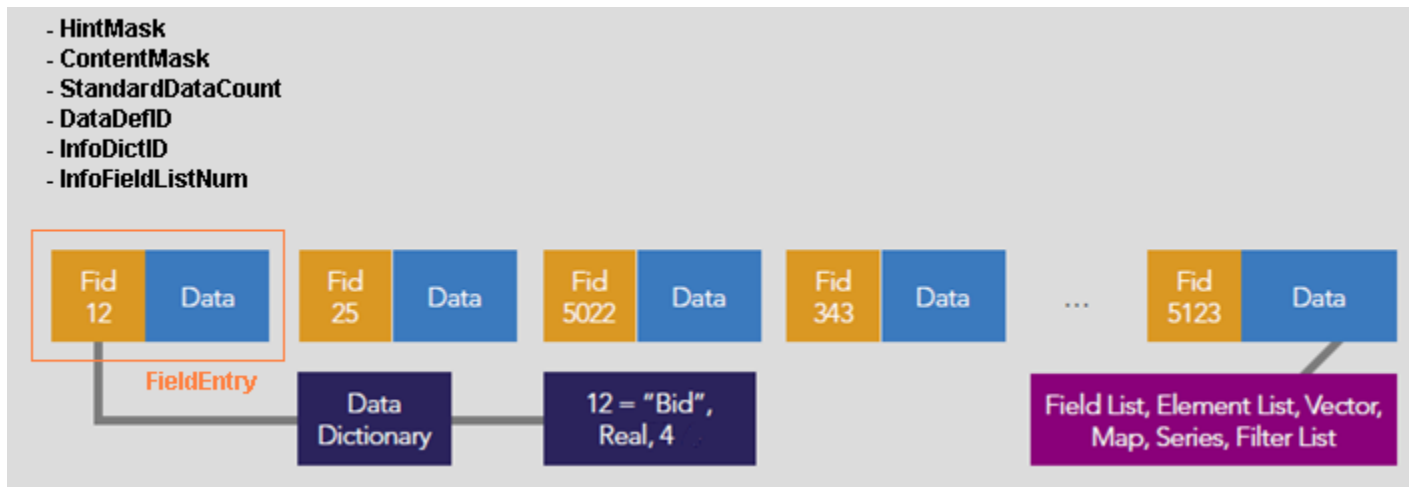


Figure 37: FieldList Structure

The following table shows specific **FieldList** and **FieldEntry** properties.

PROPERTY NAME	DESCRIPTION
HintMask	Indicates presence of optionally contained interfaces and attributes. The only possible value for HintMaskFlag is Info . Info indicates that the FieldList contains a Dictionary ID and a FieldList number (also known as a template number). Info is present in the initial refresh and any refresh following a ClearCache (as determined by the IndicationMask in the RespMsg).
ContentMask	Consumer read-only, related type information. Possible values of ContentMaskFlag include: <ul style="list-style-type: none"> • LocalDataDef: Indicates whether any DefinedData content is locally defined. If false and DefinedData is true, a consumer application needs to specify a definition FieldListDef with the same DataDefID in the start method on FieldListReadIterator. • DefinedData: Indicates whether the FieldEntry was created from DefinedData opposed to StandardData. If true, use the Data property to obtain the underlying data. Otherwise obtain the type information from a dictionary and use the GetData() method with a formal parameter.
StandardDataCount	The number of StandardData FieldEntries. The content of DefinedData FieldEntries may exist though do not have a count. The application needs to decode DefineData FieldEntries to determine its count.
DataDefID	The DataDef identifier, referencing the Defined Data content. The default value is 0. The Defined Data definition must have been formerly specified in an alternate container (at local or global scope). Typically a consumer application needs only check the ContentMask flags to determine if the Defined Data content is present (via the DefinedData flag) and if the Defined Data definition is at local scope (via the LocalDataDef flag), else global scope. The range of the DataDefID triggers the value of the ContentMaskFlag.LocalDataDef . <ul style="list-style-type: none"> • Range: $0 \leq x < 32K$ • Local Scoped Range: $0 \leq x < 16$ • Global Scoped Range: $16 \leq x < 32K$
InfoDictID	The Dictionary identifier used to reference a dictionary that contains Field identifiers and fields associated to each Field identifier. One of the fields associated to each Field identifier is type information that can be used to obtain StandardData contained in the FieldEntries. Use of a dictionary requires the Dictionary identifier to be present in the first refresh of the initial RespMsg or first refresh following a ClearCache (as indicated by the ClearCache flag accessible via the RespMsg). A FieldList can support multiple dictionaries. A FieldEntry with the reserved Field identifier of zero indicates a temporary change to the Dictionary identifier. In this case, the value of the Dictionary identifier is then in the contained data of this FieldEntry . This temporary change is solely scoped to this FieldList instance or until a subsequent FieldEntry additionally has a Field Identifier of zero.
InfoFieldListNum	A FieldList Number (also known as a template number or record template). Use of a dictionary requires the FieldList Number to be present in the first refresh of the initial RespMsg or first refresh following a ClearCache (as indicated by the ClearCache flag accessible via the RespMsg).

Table 32: FieldList properties

PROPERTY NAME	DESCRIPTION
ContentMask	Returns a mask indicating the contents available on the Field. The only possible value for ContentMaskFlag is DefinedData , which indicates whether the FieldEntry was created from DefinedData as opposed to StandardData. If true, use the Data property to obtain the underlying data. Otherwise obtain the type information from a dictionary and use the GetData() method with a formal parameter.
Data	The contained data.
FieldID	The Field identifier. A dictionary provides the association from the Field identifier to the type of contained data. A reserved value of zero indicates a dictionary change. Range: $-32K < x < 32K$ excluding zero.

Table 33: FieldEntry properties

6.1.8.4 FieldList Info

As indicated by the presence of `FieldList.HintMask.Info`, the `FieldList` might contain a dictionary ID and a field list number (also known as a template number). The dictionary ID references a specific dictionary that contains field identifiers with associated data types and other information needed to decode entries from the `FieldList`. If specified, `FieldList` information needs to be present in the item's initial `RespMsg`. An item's subsequent messages will use the dictionary ID and field list number as specified on the initial refresh. If not present on the initial refresh, the value of dictionary ID defaults to 1.

6.1.8.5 ElementList

An `ElementList` is a Container of flexible self-describing named Entries which is known as `ElementEntry`. The Entry identifier is a name of type string.

An `ElementList` 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 `ElementEntry` has a maximum encoded length of 65,535 bytes. These limitations can change in subsequent releases.

Below is the `ElementList` structure.

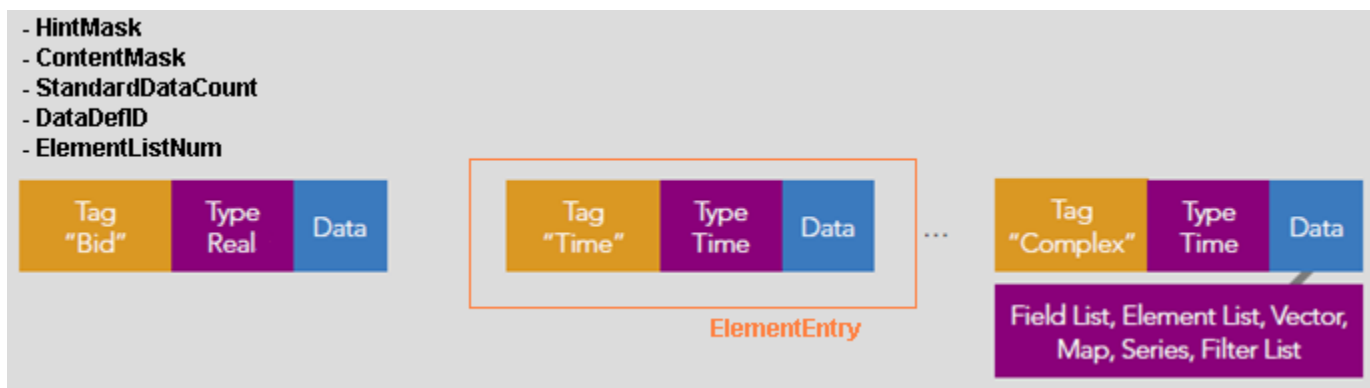


Figure 38: ElementList Structure

The following table shows the specific [ElementList](#) and [ElementEntry](#) properties.

PROPERTY NAME	DESCRIPTION						
HintMask	<p>Indicates presence of optionally contained interfaces and attributes. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>ElementListNum</td><td>Indicates that the ElementList contains an ElementList number (also known as a template number). The ElementList number is present in the initial refresh and any refresh following a ClearCache (determined by the IndicationMask in the RespMsg).</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	ElementListNum	Indicates that the ElementList contains an ElementList number (also known as a template number). The ElementList number is present in the initial refresh and any refresh following a ClearCache (determined by the IndicationMask in the RespMsg).		
HINTMASK FLAG	DESCRIPTION						
ElementListNum	Indicates that the ElementList contains an ElementList number (also known as a template number). The ElementList number is present in the initial refresh and any refresh following a ClearCache (determined by the IndicationMask in the RespMsg).						
ContentMask	<p>Consumer read-only, related type information. All possible values of ContentMaskFlag are as follows:</p> <table> <tr> <th>CONTENTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DefinedData</td><td>Indicates whether ElementList contains the DefinedData content that requires either a local or global DataDef.</td></tr> <tr> <td>LocalDataDef</td><td>Indicates whether any DefinedData content is locally defined. If false and DefinedData Flag is true, a consumer application needs to specify a definition (ElementListDef) with the same DataDefID in the start method on ElementListReadIterator.</td></tr> </table>	CONTENTMASK FLAG	DESCRIPTION	DefinedData	Indicates whether ElementList contains the DefinedData content that requires either a local or global DataDef.	LocalDataDef	Indicates whether any DefinedData content is locally defined. If false and DefinedData Flag is true, a consumer application needs to specify a definition (ElementListDef) with the same DataDefID in the start method on ElementListReadIterator .
CONTENTMASK FLAG	DESCRIPTION						
DefinedData	Indicates whether ElementList contains the DefinedData content that requires either a local or global DataDef.						
LocalDataDef	Indicates whether any DefinedData content is locally defined. If false and DefinedData Flag is true, a consumer application needs to specify a definition (ElementListDef) with the same DataDefID in the start method on ElementListReadIterator .						
StandardDataCount	The number of StandardData ElementEntries. The content of DefinedData ElementEntries may exist though do not have a count. The application need decode DefineData ElementEntries to determine its count.						
DataDefID	<p>The DataDef identifier associated to the Defined Data content. The Defined Data definition need have been formerly specified in an alternate container (at local or global scope). Typically a consumer application need only check the ContentMask flags to determine if Defined Data content is present (via DefinedData flag) and if Defined Data definition is at local scope (via LocalDataDef flag), else global scope. The range of the DataDefID triggers the value of the ContentMaskFlag.LocalDataDef.</p> <p>Range: $0 \leq x < 32K$. Local Scoped Range: $0 \leq x < 16$. Global Scoped Range: $16 \leq x < 32K$.</p>						
ElementListNum	An ElementList Number. (also known as a Template number).						

Table 34: ElementList properties

PROPERTY NAME	DESCRIPTION				
ContentMask	<p>A mask indicating the contents available on the ElementEntry. All possible values of ContentMaskFlag are as follows:</p> <table> <tr> <th>CONTENTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DefinedData</td><td>Indicates the presence of defined data</td></tr> </table>	CONTENTMASK FLAG	DESCRIPTION	DefinedData	Indicates the presence of defined data
CONTENTMASK FLAG	DESCRIPTION				
DefinedData	Indicates the presence of defined data				
Data	<p>The contained Data. For an entry obtained from the read iterator, the Session Layer internally populates the contained Data. Once the application specifies the Data property on any entry instance, the Data property returns a reference to the data specified by the Data property. In this case, the reference is only valid if the data has not gone out of scope.</p> <p>Encoded Range: $0 \leq x < 64K$</p>				
Name	<p>The name identifier.</p> <p>Range: $0 \leq x < 32K$</p>				

Table 35: ElementEntry properties

6.1.8.6 Series

A [Series](#) is a Container of accruable repetitive structured Entries each known as [SeriesEntry](#). The Entry has no identifier but rather dependent on the implicit order within the Container.

A [Series](#) can contain zero to 65,535 entries, where zero entries indicates an empty Series. 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.

Below is the [Series](#) structure.

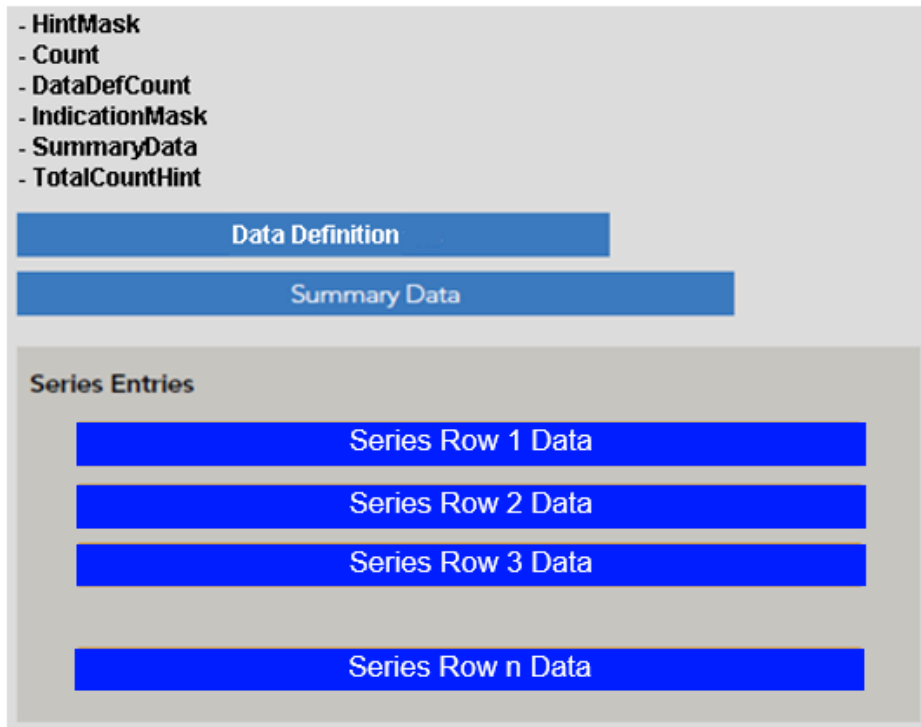


Figure 39: Series Structure

The following table shows the specific [Series](#) and [SeriesEntry](#) properties.

PROPERTY NAME	DESCRIPTION								
HintMask	Indicates presence of optionally contained interfaces and attributes. All possible values of HintMaskFlag are as follows: <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>TotalCountHint</td><td>Flag indicates the presence of a TotalCountHint.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	TotalCountHint	Flag indicates the presence of a TotalCountHint.				
HINTMASK FLAG	DESCRIPTION								
TotalCountHint	Flag indicates the presence of a TotalCountHint.								
Count	The number of SeriesEntries. SeriesEntries solely contains the StandardData content.								
DataDefCount	The number of DataDefs. Consumer applications should first use this count to determine if there are any DataDefs. If the value is non-zero, the application should then iterate through the DataDefs. An application needs to iterate through all DataDefs (via the DataDefReadIterator) prior to decoding any SeriesEntries. By iterating through the DataDefs, the application can determine if there are any DataDefs that are non-local and thus need be cloned.								
IndicationMask	A mask indicating specific properties of a Series . All possible values of IndicationMaskFlag are as follows: <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DataDef</td><td>Indicates the Series that contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator. For encoding, an application needs iterate and encode DataDefs using the DataDefWriteIterator.</td></tr> <tr> <td>SummaryData</td><td>Indicates the Data that contains Summary Data. Summary Data must have the same DataType as contained entries.</td></tr> <tr> <td>Entries</td><td>Indicates the Data that contains entries.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	DataDef	Indicates the Series that contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator . For encoding, an application needs iterate and encode DataDefs using the DataDefWriteIterator .	SummaryData	Indicates the Data that contains Summary Data. Summary Data must have the same DataType as contained entries.	Entries	Indicates the Data that contains entries.
INDICATIONMASK FLAG	DESCRIPTION								
DataDef	Indicates the Series that contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator . For encoding, an application needs iterate and encode DataDefs using the DataDefWriteIterator .								
SummaryData	Indicates the Data that contains Summary Data. Summary Data must have the same DataType as contained entries.								
Entries	Indicates the Data that contains entries.								
SummaryData	The Summary Data in the Series. Summary Data must have the same DataType as contained entries.								
TotalCountHint	SeriesEntries solely contains the StandardData content.								
DataDef	Data definitions for defined data about the Series . This interface is designed for SingleReadIterator to use data definitions to decode the defined data later.								

Table 36: Series properties

PROPERTY NAME	DESCRIPTION
Data	The contained Data. For an entry obtained from the read iterator, the Session Layer internally populates the contained Data. Once the application specifies the Data property on any entry instance, the Data property returns a reference to the data specified by the Data property. In this case, the reference is only valid if the data has not gone out of scope. Encoded Range: 0 <= x < 64K

Table 37: SeriesEntry properties

6.1.8.7 Vector

A **Vector** is a Container of position-oriented highly manipulable Entries each known as a **VectorEntry**. The Entry identifier is a position within the **Vector**.

A **Vector** can contain zero to 65,535 entries, where zero entries indicates an empty Vector. 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.

Below is the **vector** structure.

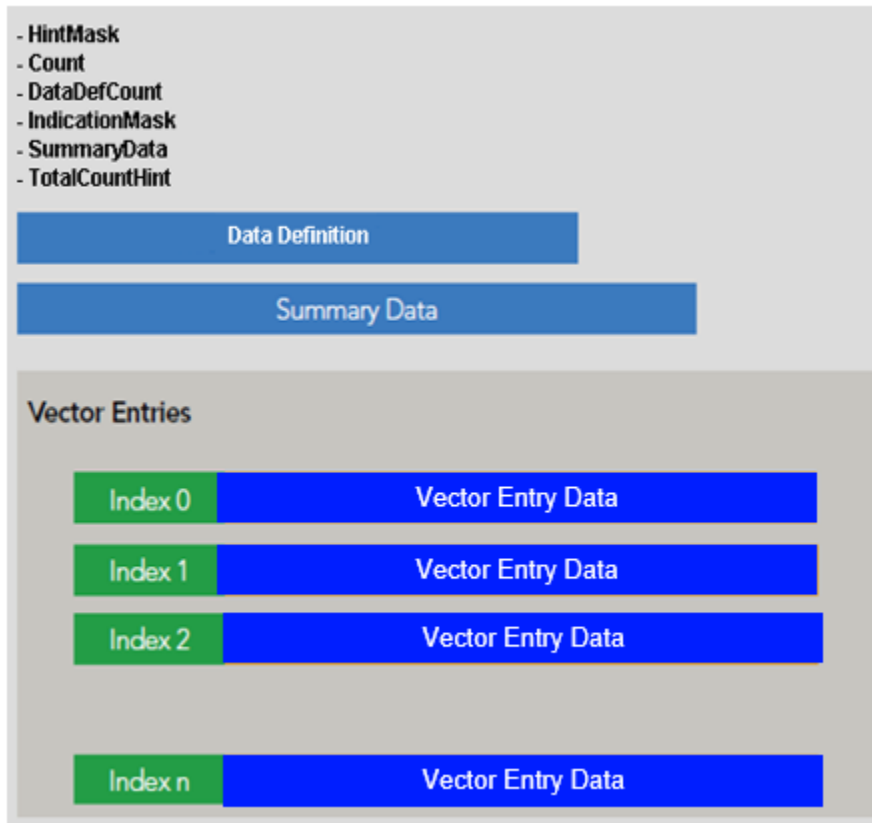


Figure 40: Vector Structure

The table below shows the specific **Vector** and **VectorEntry** properties.

PROPERTY NAME	DESCRIPTION												
HintMask	<p>Indicates the presence of optionally contained interfaces and attributes. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>TotalCountHint</td><td>Flag indicates the presence of a TotalCountHint.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	TotalCountHint	Flag indicates the presence of a TotalCountHint.								
HINTMASK FLAG	DESCRIPTION												
TotalCountHint	Flag indicates the presence of a TotalCountHint.												
Count	The number of VectorEntries. VectorEntries solely contains the StandardData content.												
DataDefCount	The number of DataDefs. Consumer applications should first use this count to determine if there are any DataDefs. If the value is non-zero, the application should then iterate through the DataDefs. An application needs to iterate through all DataDefs (via the DataDefReadIterator) prior to decoding any VectorEntries. By iterating through the DataDefs, the application can determine if there are any DataDefs that are non-local and thus need be cloned.												
IndicationMask	<p>A mask indicating the properties on the Vector. All possible values of IndicationMaskFlag are as follows:</p> <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DataDef</td><td>Indicates the Vector that contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator. For encoding, an application needs to iterate and encode DataDefs using the DataDefWriteIterator.</td></tr> <tr> <td>SummaryData</td><td>Indicates the Data that contains Summary Data. The Summary Data must have the same DataType as contained entries.</td></tr> <tr> <td>Entry</td><td>Indicates the Data that contains entries.</td></tr> <tr> <td>PermissionDataPerEntry</td><td>Provider-specified indication that permission data may reside in the Entry. This value needs to be true if PermissionData resides in any Entry. The PermissionDataFlag on the Entry indicates whether permission data is present on this particular Entry.</td></tr> <tr> <td>Sortable</td><td>Indicates that insert and delete are available. Typically an application needs to create a doubly-linked list to support Sortable whereas a singularly-linked is otherwise satisfactory.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	DataDef	Indicates the Vector that contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator . For encoding, an application needs to iterate and encode DataDefs using the DataDefWriteIterator .	SummaryData	Indicates the Data that contains Summary Data. The Summary Data must have the same DataType as contained entries.	Entry	Indicates the Data that contains entries.	PermissionDataPerEntry	Provider-specified indication that permission data may reside in the Entry. This value needs to be true if PermissionData resides in any Entry. The PermissionDataFlag on the Entry indicates whether permission data is present on this particular Entry.	Sortable	Indicates that insert and delete are available. Typically an application needs to create a doubly-linked list to support Sortable whereas a singularly-linked is otherwise satisfactory.
INDICATIONMASK FLAG	DESCRIPTION												
DataDef	Indicates the Vector that contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator . For encoding, an application needs to iterate and encode DataDefs using the DataDefWriteIterator .												
SummaryData	Indicates the Data that contains Summary Data. The Summary Data must have the same DataType as contained entries.												
Entry	Indicates the Data that contains entries.												
PermissionDataPerEntry	Provider-specified indication that permission data may reside in the Entry. This value needs to be true if PermissionData resides in any Entry. The PermissionDataFlag on the Entry indicates whether permission data is present on this particular Entry.												
Sortable	Indicates that insert and delete are available. Typically an application needs to create a doubly-linked list to support Sortable whereas a singularly-linked is otherwise satisfactory.												
SummaryData	The Summary Data in the Vector . The Summary Data must have the same DataType as contained entries.												
TotalCountHint	<p>Provider-specified hint that indicates the total number of entries spanning all refreshes. It is useful solely for multi-part refreshes.</p> <p>Valid Range: $0 \leq x < 2^{30}$.</p>												
DataDef	Data definitions for defined data about the Vector . This interface is designed for SingleReadIterator to use data definitions to decode the defined data later.												

Table 38: Vector properties

PROPERTY NAME	DESCRIPTION				
HintMask	<p>Indicates the presence of optionally contained interfaces and attributes. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>PermissionData</td><td>Indicates whether the message contains Permission Data.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	PermissionData	Indicates whether the message contains Permission Data.
HINTMASK FLAG	DESCRIPTION				
PermissionData	Indicates whether the message contains Permission Data.				

PROPERTY NAME	DESCRIPTION												
Data	<p>The contained Data. For an entry obtained from the read iterator, the Session Layer internally populates the contained Data. Once the application specifies the Data property on any entry instance, the Data property returns a reference to the data specified by the Data property. In this case, the reference is only valid if the data has not gone out of scope.</p> <p>Encoded Range: $0 \leq x < 64K$</p>												
Position	The position of the VectorEntry (0-based, $< 2^{30}$).												
Action	<p>The action to occur at the specified position. All possible values of ActionEnum are as follows:</p> <table> <tr> <th>ACTION ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>Update</td><td>Indicates a partial change of the entry. No impact to other entries.</td></tr> <tr> <td>Set</td><td>Indicates to replace the entry. No impact to other entries.</td></tr> <tr> <td>Clear</td><td>Indicates to empty the entry. No impact to other entries. Contains no data.</td></tr> <tr> <td>Insert</td><td>Indicates to place the entry in between other entries. Increases any higher-ordered position by one. May leave gaps if previous lower-ordered position is unpopulated. Only valid if the Sortable Flag on the IndicationMask is true.</td></tr> <tr> <td>Delete</td><td>Indicates to remove the entry. Decreases any higher-ordered position by one. Only valid if the Sortable Flag on the IndicationMask is true. Contains no data.</td></tr> </table>	ACTION ENUM	DESCRIPTION	Update	Indicates a partial change of the entry. No impact to other entries.	Set	Indicates to replace the entry. No impact to other entries.	Clear	Indicates to empty the entry. No impact to other entries. Contains no data.	Insert	Indicates to place the entry in between other entries. Increases any higher-ordered position by one. May leave gaps if previous lower-ordered position is unpopulated. Only valid if the Sortable Flag on the IndicationMask is true.	Delete	Indicates to remove the entry. Decreases any higher-ordered position by one. Only valid if the Sortable Flag on the IndicationMask is true. Contains no data.
ACTION ENUM	DESCRIPTION												
Update	Indicates a partial change of the entry. No impact to other entries.												
Set	Indicates to replace the entry. No impact to other entries.												
Clear	Indicates to empty the entry. No impact to other entries. Contains no data.												
Insert	Indicates to place the entry in between other entries. Increases any higher-ordered position by one. May leave gaps if previous lower-ordered position is unpopulated. Only valid if the Sortable Flag on the IndicationMask is true.												
Delete	Indicates to remove the entry. Decreases any higher-ordered position by one. Only valid if the Sortable Flag on the IndicationMask is true. Contains no data.												
PermissionData	The permission expression, which contains authorization information.												

Table 39: VectorEntry properties

6.1.8.8 Map

A [Map](#) is a Container of manipulable associative key-value pair entries each known as a [MapEntry](#). The identifier is a key of type Data.

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.

Below is the [Map](#) structure.

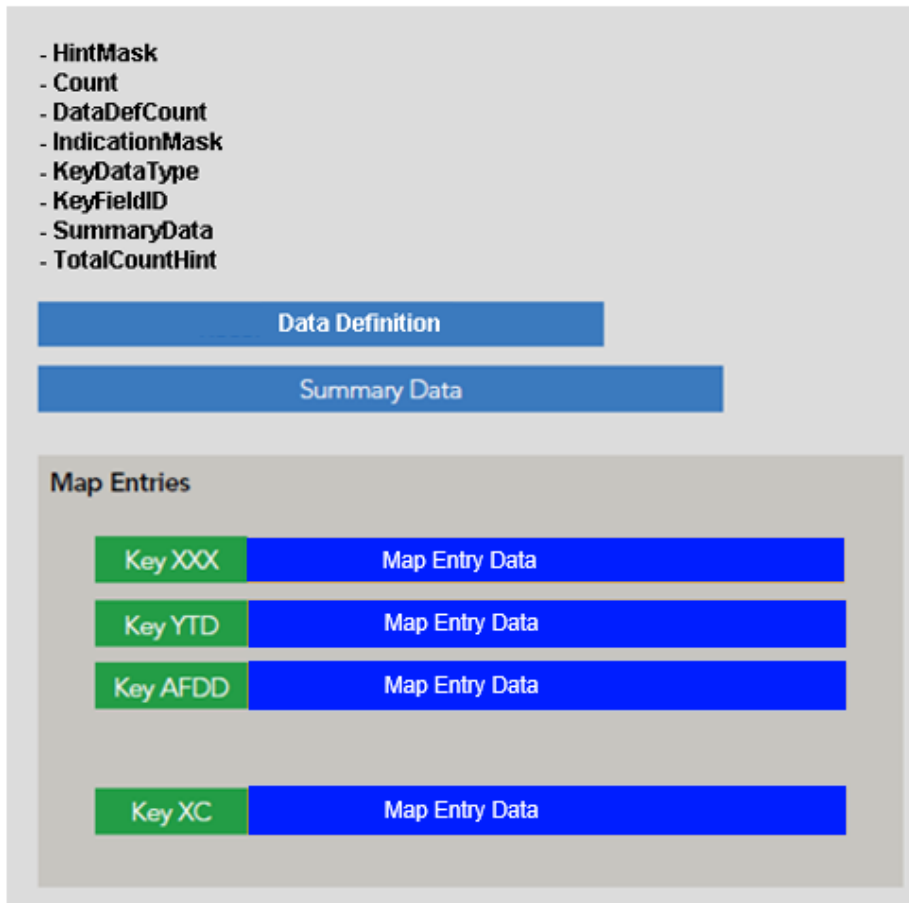


Figure 41: Map Structure

The table below shows [Map](#) and [MapEntry](#) properties.

PROPERTY NAME	DESCRIPTION										
HintMask	<p>Indicates the presence of optionally contained interfaces and attributes. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>TotalCountHint</td><td>Flag indicates the presence of a TotalCountHint.</td></tr> <tr> <td>KeyFieldID</td><td>Flag indicates the presence of a KeyFieldID.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	TotalCountHint	Flag indicates the presence of a TotalCountHint.	KeyFieldID	Flag indicates the presence of a KeyFieldID.				
HINTMASK FLAG	DESCRIPTION										
TotalCountHint	Flag indicates the presence of a TotalCountHint.										
KeyFieldID	Flag indicates the presence of a KeyFieldID.										
Count	The number of MapEntries. MapEntries solely contains the StandardData content.										
DataDefCount	The number of DataDefs. Consumer applications should first use this count to determine if there are any DataDefs. If the value is non-zero, the application should then iterate through the DataDefs. An application needs to iterate through all DataDefs (via the DataDefReadIterator) prior to decoding any MapEntries. By iterating through the DataDefs, the application can determine if there are any DataDefs that are non-local and thus need be cloned.										
IndicationMask	<p>A mask indicating the properties on the Map. All possible values of IndicationMaskFlag are as follows:</p> <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DataDef</td><td>Indicates the Map contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator. For encoding, an application needs to iterate and encode DataDefs using the DataDefWriteIterator.</td></tr> <tr> <td>SummaryData</td><td>Indicates the Data contains Summary Data. Summary Data must have the same DataType as contained entries.</td></tr> <tr> <td>Entries</td><td>Indicates the Data contains entries.</td></tr> <tr> <td>PermissionDataPerEntry</td><td>Provider-specified indication that permission data may reside in the Entry. This value needs to be true if PermissionData resides in any Entry. The PermissionDataFlag on the Entry indicates whether permission data is present on this particular Entry.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	DataDef	Indicates the Map contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator . For encoding, an application needs to iterate and encode DataDefs using the DataDefWriteIterator .	SummaryData	Indicates the Data contains Summary Data. Summary Data must have the same DataType as contained entries.	Entries	Indicates the Data contains entries.	PermissionDataPerEntry	Provider-specified indication that permission data may reside in the Entry. This value needs to be true if PermissionData resides in any Entry. The PermissionDataFlag on the Entry indicates whether permission data is present on this particular Entry.
INDICATIONMASK FLAG	DESCRIPTION										
DataDef	Indicates the Map contains Data Definitions. For decoding, an application needs to iterate over the DataDefs via the DataDefReadIterator . For encoding, an application needs to iterate and encode DataDefs using the DataDefWriteIterator .										
SummaryData	Indicates the Data contains Summary Data. Summary Data must have the same DataType as contained entries.										
Entries	Indicates the Data contains entries.										
PermissionDataPerEntry	Provider-specified indication that permission data may reside in the Entry. This value needs to be true if PermissionData resides in any Entry. The PermissionDataFlag on the Entry indicates whether permission data is present on this particular Entry.										
KeyDataType	Provider-specified type of key. Merely provides access to the key type information prior to obtaining the actual keys. As indicated in the concepts section, the commonality of key data is homogeneous.										
KeyFieldID	The key field ID.										
SummaryData	The Summary Data in the Map. The Summary Data must have the same DataType as contained entries.										
TotalCountHint	<p>Provider-specified hint that indicates the total number of entries spanning all refreshes. Useful solely for multi-part refreshes.</p> <p>Valid Range: $0 \leq x < 2^{30}$.</p>										
DataDef	Data definitions for defined data about the Map . This interface is designed for SingleReadIterator to use data definitions to decode the defined data later.										

Table 40: Map properties

PROPERTY NAME	DESCRIPTION				
HintMask	<p>Indicates the presence of optionally contained interfaces and attributes. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>PermissionData</td><td>Indicates whether the message contains Permission Data.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	PermissionData	Indicates whether the message contains Permission Data.
HINTMASK FLAG	DESCRIPTION				
PermissionData	Indicates whether the message contains Permission Data.				

PROPERTY NAME	DESCRIPTION								
Data	<p>The contained Data. For an entry obtained from the read iterator, the Session Layer internally populates the contained Data. Once the application specifies the Data property on any entry instance, the Data property returns a reference to the data specified by the Data property. In this case, the reference is only valid if the data has not gone out of scope.</p> <p>Encoded Range: $0 \leq x < 64K$</p> <p>NOTE: You must call KeyData before calling Data, otherwise your operation will not properly decode the payload data.</p>								
KeyData	<p>The contained Key.</p> <p>NOTE: You must call KeyData before calling Data, otherwise your operation will not properly decode the payload data.</p>								
Action	<p>The action occurring at the specified position. All possible values of ActionEnum are as follows:</p> <table> <tr> <th>ACTION ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>Update</td><td>Indicates a partial change of the entry. No impact to other entries.</td></tr> <tr> <td>Add</td><td>Indicates to append or replace the entry. No impact to other entries.</td></tr> <tr> <td>Delete</td><td>Indicates to remove the entry. No impact to other entries. Contains no data.</td></tr> </table>	ACTION ENUM	DESCRIPTION	Update	Indicates a partial change of the entry. No impact to other entries.	Add	Indicates to append or replace the entry. No impact to other entries.	Delete	Indicates to remove the entry. No impact to other entries. Contains no data.
ACTION ENUM	DESCRIPTION								
Update	Indicates a partial change of the entry. No impact to other entries.								
Add	Indicates to append or replace the entry. No impact to other entries.								
Delete	Indicates to remove the entry. No impact to other entries. Contains no data.								
PermissionData	The permission expression, which contains authorization information.								

Table 41: MapEntry properties

6.1.8.9 FilterList

A **FilterList** is a Container of loosely coupled Entries each known as **FilterEntry**. The Entry identifier is a numeric known as a **FilterID**. The **FilterID** corresponds to the binary value of a selectable flag specified by a Consumer on the **AttribInfo.HintMaskFlag.DataMask**.

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.

The contained Entry Data supports heterogeneous or homogeneous types. Homogeneous data always has a slightly higher bandwidth optimization than heterogeneous data.

Below is the **FilterList** structure.

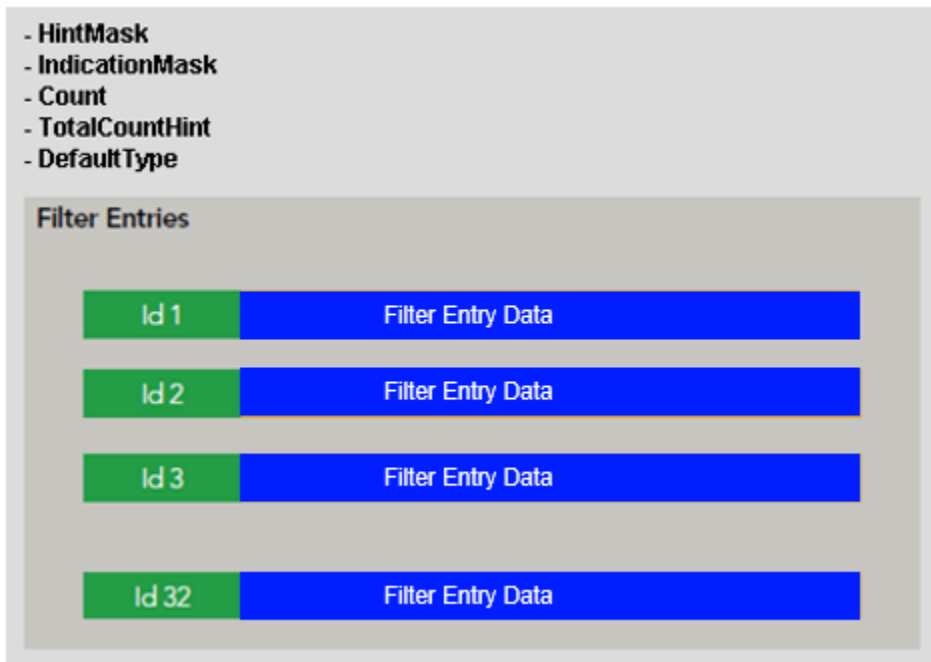


Figure 42: FilterList Structure

The table below shows the specific `FilterList` and `FilterEntry` properties.

PROPERTY NAME	DESCRIPTION				
HintMask	<p>Indicates the presence of optionally contained interfaces and attributes. All possible values of <code>HintMaskFlag</code> are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>TotalCountHint</td><td>Flag indicating the <code>TotalCountHint</code> is specified by the provider.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	TotalCountHint	Flag indicating the <code>TotalCountHint</code> is specified by the provider.
HINTMASK FLAG	DESCRIPTION				
TotalCountHint	Flag indicating the <code>TotalCountHint</code> is specified by the provider.				
IndicationMask	<p>A mask indicating the properties on the <code>FilterList</code>. All possible values of <code>IndicationMaskFlag</code> are as follows:</p> <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>PermissionDataPerEntry</td><td>Provider-specified indication that permission data may reside in the Entry. This value needs to be true if <code>PermissionData</code> resides in any Entry. The <code>PermissionDataFlag</code> in the Entry indicates whether permission data is present in this particular Entry.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	PermissionDataPerEntry	Provider-specified indication that permission data may reside in the Entry. This value needs to be true if <code>PermissionData</code> resides in any Entry. The <code>PermissionDataFlag</code> in the Entry indicates whether permission data is present in this particular Entry.
INDICATIONMASK FLAG	DESCRIPTION				
PermissionDataPerEntry	Provider-specified indication that permission data may reside in the Entry. This value needs to be true if <code>PermissionData</code> resides in any Entry. The <code>PermissionDataFlag</code> in the Entry indicates whether permission data is present in this particular Entry.				
Count	The number of <code>FilterEntries</code> . <code>FilterEntries</code> solely contains the <code>StandardData</code> content.				
TotalCountHint	Provider-specified hint that indicates the total number of entries spanning all refreshes. It is useful solely for multi-part refreshes. Valid Range: $0 \leq x < 2^{30}$.				
DefaultType	<p>The default type of data across all Entries. This type may be <code>DataType</code> or <code>DataBufferType</code>. When the type of all Entries matches this type, the container is a homogeneous container. When one or more Entries differ from this type, the container is a heterogeneous container. A homogeneous container has a more optimized encoding.</p> <p>Valid Range: $0 \leq x < 256$. Reserved Range: $0 \leq x < 256$.</p>				

Table 42: FilterList properties

PROPERTY NAME	DESCRIPTION								
HintMask	<p>Indicates the presence of optionally contained interfaces and attributes. All possible values of <code>HintMaskFlag</code> are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>PermissionData</td><td>Indicates whether the message contains Permission Data.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	PermissionData	Indicates whether the message contains Permission Data.				
HINTMASK FLAG	DESCRIPTION								
PermissionData	Indicates whether the message contains Permission Data.								
Data	<p>The contained Data. For an entry obtained from the read iterator, the Session Layer internally populates the contained Data. Once the application specifies the <code>Data</code> property on any entry instance, the <code>Data</code> property returns a reference to the data specified by the <code>Data</code> property. In this case, the reference is only valid if the data has not gone out of scope.</p> <p>Encoded Range: $0 \leq x < 64K$</p>								
FilterID	The <code>FilterEntry</code> identifier.								
Action	<p>The action to occur at the specified position. All possible values of <code>ActionEnum</code> are as follows:</p> <table> <tr> <th>ACTION ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>Update</td><td>Indicates a partial change of the entry. No impact to other entries.</td></tr> <tr> <td>Set</td><td>Indicates to replace the entry. No impact to other entries.</td></tr> <tr> <td>Clear</td><td>Indicates to empty the entry. No impact to other entries. Contains no data.</td></tr> </table>	ACTION ENUM	DESCRIPTION	Update	Indicates a partial change of the entry. No impact to other entries.	Set	Indicates to replace the entry. No impact to other entries.	Clear	Indicates to empty the entry. No impact to other entries. Contains no data.
ACTION ENUM	DESCRIPTION								
Update	Indicates a partial change of the entry. No impact to other entries.								
Set	Indicates to replace the entry. No impact to other entries.								
Clear	Indicates to empty the entry. No impact to other entries. Contains no data.								
PermissionData	The permission expression, which contains authorization information.								

Table 43: FilterEntry properties

6.1.9 Data Definition

A Data Definition is a container for Entry definitions. These Entry definitions contain an Entry identifier along with the type of contained Data.

There are two types of Data Definition: `ElementListDef` and `FieldListDef` which are descendants of the `DataDef` interface. The `FieldList`, `ElementList`, `Series`, `Map` and `Vector` containers may house Defined Data Definitions.

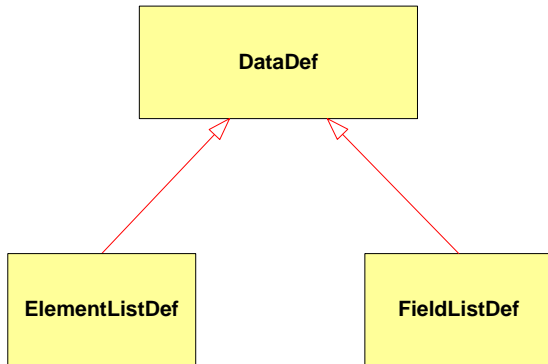


Figure 43: Defined Data Definitions

Below are the `DataDef` properties.

PROPERTY NAME	DESCRIPTION
DefType	Specifies the type of data definition. There are two values: <code>ElementListDef</code> and <code>FieldListDef</code> .

Table 44: DataDef properties

6.1.9.1 ElementList Definition

The ElementList Definition is a descendant of the Data Definition that describes an [ElementList](#).

Below is the ElementList Definition structure.

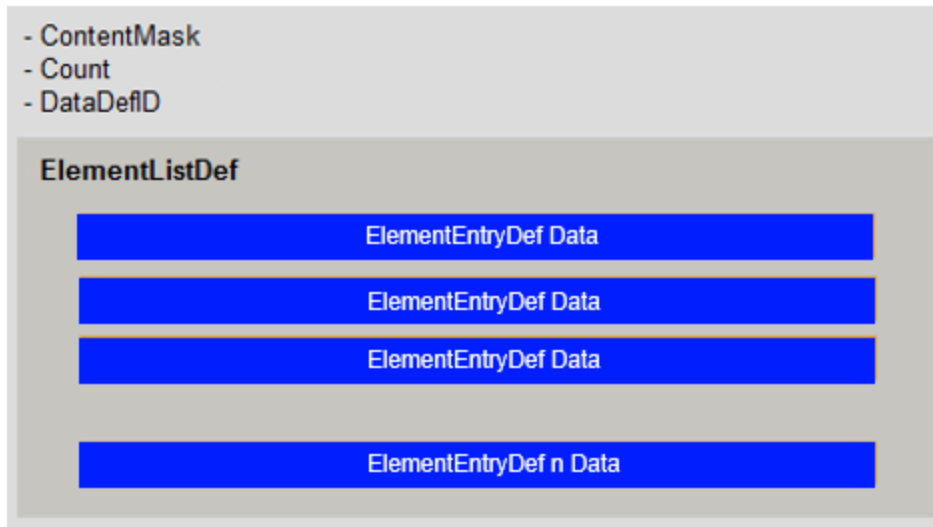


Figure 44: ElementListDef Structure

The table below shows the specific [ElementListDef](#) and [ElementEntryDef](#) properties.

PROPERTY NAME	DESCRIPTION
ContentMask	Consumer read-only, related type information. The only possible value for ContentMaskFlag is LocalDataDef , which indicates whether this ElementListDef is a local DefinedData definition. If false, the Consumer application should clone this DefinedData definition as Provider is likely to send the DefinedData content having the same DefID in a subsequent message.
Count	The number of ElementEntryDefs.
DataDefID	The identifier associated to the data definition. A FieldList DataDef ID and ElementList DataDef ID have separate namespaces. The Range triggers the value of the ContentMaskFlag.LocalDataDef . Valid Range: $0 \leq x < 32K$. Local Scoped Range: $0 \leq x < 16$. A value of 0 is not present on the wire, and thus reduces bandwidth consumption. Global Scoped Range: $16 \leq x < 32K$.

Table 45: ElementListDef properties

PROPERTY NAME	DESCRIPTION
EncodedType	The actual EncodeType such as LengthSpecified , Bit8 or ReallLength .
Type	The contained type, either DataType or DataBufferType .
Name	The name identifier.

Table 46: ElementEntryDef properties

6.1.9.2 FieldList Definition

The FieldList Definition is a descendant of the Data Definition interface that describes a [FieldList](#).

Below is the [FieldList](#) definition structure.

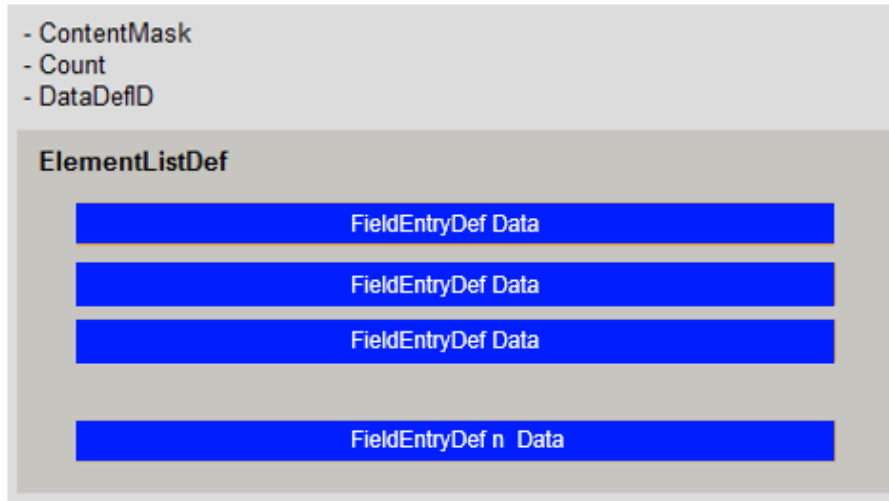


Figure 45: FieldListDef Structure

The following table shows the specific [FieldListDef](#) and [FieldEntryDef](#) properties.

PROPERTY NAME	DESCRIPTION				
ContentMask	Consumer read-only, related type information. All possible values of ContentMaskFlag are as follows: <table border="1"> <thead> <tr> <th>CONTENTMASK FLAG</th><th>DESCRIPTION</th></tr> </thead> <tbody> <tr> <td>LocalDataDef</td><td>Indicates whether this ElementListDef is a local DefinedData definition. If false, the Consumer application should clone this DefinedData definition, as Provider is likely to send DefinedData content having the same DefID in a subsequent message.</td></tr> </tbody> </table>	CONTENTMASK FLAG	DESCRIPTION	LocalDataDef	Indicates whether this ElementListDef is a local DefinedData definition. If false, the Consumer application should clone this DefinedData definition, as Provider is likely to send DefinedData content having the same DefID in a subsequent message.
CONTENTMASK FLAG	DESCRIPTION				
LocalDataDef	Indicates whether this ElementListDef is a local DefinedData definition. If false, the Consumer application should clone this DefinedData definition, as Provider is likely to send DefinedData content having the same DefID in a subsequent message.				
Count	The number of FieldEntryDefs.				
DataDefID	The identifier associated to the data definition. A FieldList DataDef ID and ElementList DataDef ID have separate namespaces. The Range triggers the value of the ContentMaskFlag.LocalDataDef . Valid Range: $0 \leq x < 32K$. Local Scoped Range: $0 \leq x < 16$. A value of 0 is not present on the wire, and thus reduces bandwidth consumption. Global Scoped Range: $16 \leq x < 32K$.				

Table 47: FieldListDef properties

PROPERTY NAME	DESCRIPTION
EncodedType	The actual EncodeType such as LengthSpecified, Bit8 or RealLength.
Type	The contained type, either DataType or DataBufferType .
FieldID	A dictionary provides the association from the Field identifier to the type of data. A reserved value of zero indicates a dictionary change.

Table 48: FieldEntryDef properties

6.1.10 Masks

Several of Data Package interfaces use Masks. A Mask is a set of flags that convey information to a Consumer or Provider. Masks differ in that a Consumer, Provider and/or implementation may specify the flags within a Mask. In addition to Hint Mask and Indication Mask the Data Package also has the following masks:

- Content Mask: Consumer- specified or Provider-specified mask that describes the structure of information—e.g., whether the information is structured from Defined Data, such as [DefinedData](#).
- Iterator Mask: Consumer-specified or Provider-specified mask that governs selection criteria on an iterator.

6.1.11 Advanced Distribution Concepts

6.1.11.1 Fragmentation

Some Containers allow fragmentation across multiple Data instances. Each Data instance is known as a fragment and typically occurs in a unique Message. Encoding applications may fragment data by distributing subsets of entries across multiple refresh messages.

Containers that support fragmentation provide a `TotalCountHint`. A `TotalCountHint` is a Provider's suggestion of the total number of Entries within a Container across all fragments. Only the first fragment of the container must contain the `TotalCountHint`. Each Entry may either be Standard Data or Defined Data. A Consumer may choose to use the `TotalCountHint` to pre-allocate sufficient memory to store all received and expected Entries.

A fragmented container is available only to the entries at the outermost container of the payload data. This implies that an entry at the outermost container of the payload data cannot be further fragmented across messages.

6.1.11.2 PartialUpdateReadIterator

The [PartialUpdateReadIterator](#) provides a sequential extraction of partial field updates from update messages. Applications can then apply the partial field updates on raw full field data cached earlier from refresh messages. The interface usually works with the [RMTEConverter](#) interface from the Common package to convert encoded RMTES strings to strings of format UTF8 or UCS2.

Before a client uses the [PartialUpdateReadIterator](#) interface to parse partial field updates, the client can call the [IsPartialUpdates\(\)](#) property from the [DataBuffer](#) interface to determine if the update message contains partial field updates. If the update message contains partial field updates, the [PartialUpdateReadIterator](#) interface needs to parse the partial field updates.

[PartialUpdateReadIterator](#) behaves like a standard iterator with the following exceptions:

- The `Value` property returns an RFA [Buffer](#) which contains the partial field update string or full field update string at the current position. The string is an RMTES-encoded string.
- The `Offset()` property returns the offset for the partial field update or returns -1 for a full field update or error. The offset is a 0-based offset which is from the start of the cached field buffer being updated and indicates the point at which to apply the update's partial field data element. The cached field buffer contains the full field RMTES-encoded string.

For details on converting RMTES encoded strings to strings of Unicode formats, refer to section 5.2.5, [RMTEConverter](#).

6.1.11.3 Working with Encoded Buffers

The `EncodedBuffer` property provides a way to access data as a raw encoded Buffer. Typically applications need not access this raw encoded Buffer as the Data package provides interfaces to fully encode and decode all data. However, there are some cases where an application may want to access this raw encoded Buffer. These include an encoding application that wants to know the size of the currently encoded data, a hybrid application that wants to pass data through, record/playback utilities, data integrity test applications, and performance test applications.

For the `EncodedBuffer` property, applications need to ensure that the Buffer has the same type as the concrete Data instance.

6.1.11.4 Versioning Support

Refinitiv Wire Format (RWF) supports versioning to allow future enhancements and extensions. RWF version information includes major and minor versions.

RFA Data and Msg interfaces provide versioning functionality to allow applications to encode data with a connection's negotiated RWF version. Versioning support applies only to OMM connection types and OMM message domain models.

In scenarios where multiple versions exist, the RFA application must follow these guidelines to optimize usage and performance:

- Before setting `EncodedBuffer` on a `Data` or `Msg` object to allow RFA to hold pre-encoded data in the buffer, the client application must set the version information by calling `SetAssociatedMetaInfo()`. The client application can get the `MajorVersion/MinorVersion` property on the `Data` or `Msg` object to retrieve the relevant version information, cache it, and later pass it to a `SetAssociatedMetaInfo()` call.
- To optimize RFA performance, encode all opaque data (as set by the `Attrib` property call on an `AttribInfo` object or by the `Payload` property call on a `Msg` object) with the negotiated version number. To encode data with the negotiated version, a client application must call the `SetAssociatedMetaInfo()` method with the handle argument to pass the negotiated version to its data objects.

NOTE: The Login request is an exception because it is sent before a connection is established (and hence before a version is negotiated). If the Login request contains opaque data in its `AttribInfo` object, RFA encodes opaque data for a login request using the default version predefined by ETA. In this case, there is no negotiated version on the connection.

- The handle passed in the `SetAssociatedMetaInfo()` function on a `Data` object can be a login handle for:
 - A login handle (see the non-interactive provider example below.)
 - A client session handle (see the RDM provider and generic message provider examples below.)
 - An open item stream handle (see the generic message consumer example below.)
- The session configuration parameter **OMMAAllowMultiVersion** sets whether the client application supports multiple versions between physical connections in one session. If **OMMAAllowMultiVersion** is set to **NotAllowMultiVer**, then the client supports only the negotiated version from the first established connection (called the “first come, first served rule”). Different versions will be disconnected.

In this case, client applications might not receive data because the first established connection might not provide proper services or capability suitable to the client's needs. If a hybrid application has different versions between the 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 can either change RFA and Infrastructure components to use the same RWF version, or change the value of **OMMAAllowMultiVersion** to **AllowMultiVer**.

- If **OMMAAllowMultiVersion** is set to `AllowMultiVer`, the client application supports all connections, regardless of whether the versions are the same or different.
- When a consumer establishes a connection with the provider, the lowest common version is considered the negotiated version for both consumer and provider on one connection.
- The RFA adapter performs version checking. If the version that encodes data in a message does not match the connection's negotiated version, the RFA adapter decodes the message and re-encodes it with the negotiated version before sending it out on the wire. This creates latency which can negatively affect performance if the client application deals with large volumes of data. For better performance, applications should pass version information to the `Data` object when encoding data in a client application to prevent the RFA Adapter from decoding and re-encoding the data.
- There are two ways to pass version information to data objects such as `Map`, `Vector`, `Series`, `FieldList`, `ElementList`, `FiterList`, or `Array`. One way is to call `SetAssociatedMetaInfo()` on data objects by passing the handle. Another way is to call `SetAssociatedMetaInfo()` on nested data objects by passing the version information received through the `MajorVersion/MinorVersion` property on the outer data objects. Passing version information to data objects is subject to two constraints:
 - A data object and its nested data must be encoded with the same version.
 - All RWF data in an OMM message, including its pre-encoded `DataBuffer`, its payload, and its pre-encoded opaque data in `AttributInfo`, must be encoded with the same version.
- `Map`, `Vector`, and `Series` support summary data. In scenarios with multiple versions, the client application needs to declare or initialize these objects with the copy flag set to true. Otherwise, set `SummaryData` property will throw an exception if the version used for encoding the data is different from the version of the object which holds the summary data.
- One session supports multiple connections. If the session allows multiple versions, one of the negotiated versions will be used for non-interactive provider applications to encode data for publication across all connections. The particular version that the session uses will not change until the application restarts. The RFA adapter might need to re-encode data with the proper negotiated version associated with the connection before sending the message out to wire. The non-interactive provider application will pass version information through the login handle.
- For other OMM connection types (except the `RSSL_NIPROV` connection), the client application always encodes its data with the connection's negotiated version according to the results of the `SetAssociatedMetaInfo()` method.
- Passing version information occurs mainly on the data-encoding side of the client application. Some RFA example applications have versioning support implementations. For more information, see the `StartProvider_Interactive`, `StarterProvider_NonInteractive`, `HybridApp`, `StarterConsumer_GenericMsg`, and `StarterProvider_GenericMsg` examples.

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

```
if (loginHandle != 0)
{
    dataBodyMap.SetAssociatedMetaInfo(loginHandle);
}

encoder.EncodeDirectoryDataBody(dataBodyMap, cfgVariables.ServiceName, cfgVariables.VendorName, qos);

private void EncodeDirectoryMap(Map map, RFA_String svcName, RFA_String vendName, QualityOfService qos)
{
    //...

    FilterList filterList = new FilterList();
    filterList.SetAssociatedMetaInfo(map.MajorVersion, map.MinorVersion);
}
```

```
EncodeDirectoryFilterList(filterList, svcName, vendName, QoS);
mapEntry.Data = filterList;
```

Example 16: StarterProvider_NonInteractive Passing Version Information

```
if (clientSessionHandle != 0)
{
    dataBody.SetAssociatedMetaInfo(clientSessionHandle);
}
encoder.EncodeDirectoryDataBody(dataBody, cfgVariables.ServiceName, cfgVariables.VendorName, QoS);
respMsg.Payload = dataBody;
```

Example 17: StarterProvider_Interactive Passing Version Information

```
private bool SendGenericMsg(RFA_String dataValue)
{
    if (itemHandle != 0)
    {
        ElementList payLoad = new ElementList();
        payLoad.SetAssociatedMetaInfo(itemHandle);
        ElementListWriteIterator iter = new ElementListWriteIterator();
        ElementEntry entry = new ElementEntry();
        DataBuffer buffer = new DataBuffer();
        RFA_String ename = new RFA_String();
        RFA_String val = new RFA_String();

        iter.Start(payLoad);

        ename.Set("Data");
        val.Set(dataValue.ToString());
        buffer.SetFromString(val, DataBuffer.DataBufferEnum.StringAscii);
        entry.Name = ename;
        entry.Data = buffer;
        iter.Bind(entry);
        iter.Complete();
    }
}
```

Example 18: StarterConsumer_GenericMsg Passing Version Information

6.1.11.5 Reusable Objects

Containers (e.g. [Map](#), [Series](#)), Entry (e.g. [MapEntry](#), [ArrayEntry](#)) and Definition (e.g. [ElementEntryDef](#)) interfaces also provide the [clear\(\)](#) method to allow client application to clear data objects and its contained interfaces. These cleared data objects then may be reused without re-declaring them.

The use of [clear\(\)](#) method to reuse any Entry, Entry Definition, [Array](#), [ElementList](#) or [FieldList](#) is optional. It may be skipped in a high performance application.

NOTE: It is mandatory to use the [clear\(\)](#) method to reuse [ElementListDef](#), [FieldListDef](#), [FilterList](#), [Map](#), [Series](#), or [Vector](#).

```

DataBuffer dataBuffer = new DataBuffer();

dataBuffer.SetInt((long)12345, DataBuffer.DataBufferEncodedEnum.Bit16Value);
dataBuffer.Clear();    // the use of clear() in this case s optional
dataBuffer.SetUInt((ulong)345, DataBuffer.DataBufferEncodedEnum.Bit8Value);

Map map = new Map();
MapEntry mapEntry = new MapEntry();
MapWriteIterator mapWit = new MapWriteIterator();

// encode map
mapWit.Start( map );

// encode mapEntry

mapWit.Bind( mapEntry );

mapEntry.Clear() ;    // the use of clear() in this case is optional

// encode mapEntry

mapWit.Bind( mapEntry );
mapWit.Complete();

map.Clear();    //the use of clear() in this case is NOT optional

// encode map again

```

Example 19: Reusable object

6.2 Data Package Usage

6.2.1 Data Encoding

The raw content needs to be encoded into data structures that can be embedded in messages flowing between sender and receiver. The data structure would either be in the form of a Container or a Leaf.

6.2.1.1 General Data Encoding Process

Containers are comprised of individual Entries. Each entry can be made of a Leaf or another nested Container. In general, encoded data can consist of one level or multiple levels of arbitrary nesting hierarchy.

Applications may encode data via a single-pass depth traversal or double-pass breadth traversal. In the case of singlepass depth traversal, the application sequentially iterates down and encodes data at the lowest tier in a hierarchy before encoding any subsequent data at a higher tier in a hierarchy. In the case of double-pass breadth traversal, the application iterates to any tier and selectively binds, or encodes and inserts, pre-encoded lower-level tiers. The advantage of singlepass depth traversal is the performance as it avoids an additional copy. The advantage of double-pass breadth traversal is that the application may combine already-encoded data structures (e.g., cached structures) and can produce a smaller encoded buffer on the wire at the expense of more processing cost (i.e., memory allocation, copying) by the application.

The Data package uses iterators to step through each entry in a container. Write iterators are used to encode and insert, or bind, entries in a container. An iterator can be defined to be container type-specific or container type-independent. Container type-independent iterators are called Single Iterators because only one iterator instance is needed to encode the message payload. In general, the application must encode containers in the following order:

1. Initialization of a container including its [HintMask](#), [IndicationMask](#) and other properties where appropriate; and
2. Encoding of [DataDef](#) (if any) and
3. Setting of [SummaryData](#) (if any) and
4. Encoding data by using the container-specific write iterator or [SinglewriteIterator](#). The following pictures show this order.

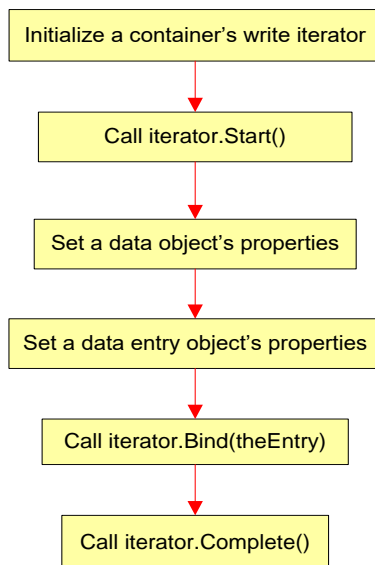


Figure 46: Using container-specific write iterator to encode data

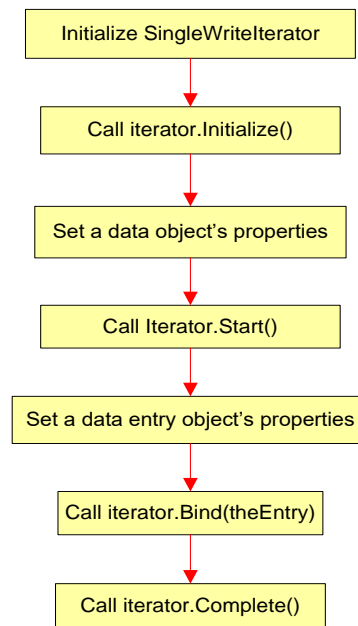


Figure 47: Using SingleWriteIterator to encode data

Encoding is done via the use of write iterators. The write iterators generally expose the interfaces listed in the table below. For more details of interface, refer to the *RFA Reference Manual .NET Edition*.

FUNCTION NAME	DESCRIPTION
Start()	Sets the position to the beginning of the data type. Parameters passed in are the data object to be encoded, regardless of whether the data is Standard or Defined, and the estimated number of entries that will be encoded.
Bind()	Encodes and inserts a DataEntry at the current position.
Complete()	Completes encoding a data type. Each Start() must be matched with a Complete() call. No more Bind() calls are allowed after this method has been called.

Table 49: Write Iterator Interface Functions

The single-pass depth traversal is also known as the pre-bind of post-encoded data while the double-pass breadth traversal is known as the post-bind of pre-encoded data. These names come from the relative order of binding and encoding data.

For pre-bind of post-encoded data, an empty entry is bound to a container using that container-specific write iterator, and then it is populated with data. In the case of the post-bind of pre-encoded data, an entry is first populated with data, and then it is bound to a container using that container-specific write iterator.

NOTE: The [DataBuffer](#) type is not allowed to be pre-bound. For details on which data types can be bound to a container, refer to Section 3.6.3.

6.2.1.2 Encoding Data Definition

There are two data types enable to contain the data definition namely `ElementListDef` and `FieldListDef`. The `ElementListDef` or `FieldListDef` objects need to be encoded by the `ElementListDefWriteIterator` or `FieldListDefWriteIterator` interface respectively. The following section depicts how to encode `ElementListDef` and `FieldListDef`.

6.2.1.2.1 Encoding ElementList Definition

An `ElementList` can contain Standard Data or Defined Data. For Defined Data, `ElementListDef` describes the defined data. An `ElementList` is a Container of flexible self-describing `ElementEntry`. `ElementEntryDef` is the definition of an `ElementEntry`.

`ElementEntryDef` interface provides the ability to encode the definition of an `ElementEntry` by calling the `Name` and `Type` properties. The type to be set for an `ElementEntry` is either `DataType` or `DataBufferType`.

```
RFA_String fids = new RFA_String();
fids.Set("FIDS");

ElementEntryDef elementEntryDef = new ElementEntryDef();
elementEntryDef.Name = fids;
elementEntryDef.Type = Reuters.RFA.Data.DataEnum.Array;
```

Example 20: Encoding ElementEntryDef

Each `ElementListDef` has an identifier which can be set by the `DataDefID` property. An `ElementList` uses the identifier to tell which `ElementListDef` is being used by it.

`ElementListDefWriteIterator` is an `ElementListDef` encode iterator. It provides the ability to encode element list data definition by calling the `Start()`, `Bind()`, and `Complete()` methods.

- The `Start()` method sets position to the beginning of the `ElementListDef`.
- The `Bind()` method binds an `ElementEntryDef` at the current position.
- The `Complete()` method completes encoding of the `ElementListDef`. Once the method is called, no more bind calls are allowed to encode more definitions on the `ElementListDef`.

```
void EncodeElementListDef()
{
    ElementListDefWriteIterator elementListDefWit = new ElementListDefWriteIterator();
    ElementListDef elementListDef = new ElementListDef();
    ElementEntryDef elementEntryDef = new ElementEntryDef();

    elementListDef.DataDefID = 1;    //set data definition identifier

    elementListDefWit.Start(elementListDef);    //start position of the elementListDef

    RFA_String fids = new RFA_String();
    RFA_String values = new RFA_String();
    RFA_String displays = new RFA_String();
    fids.Set("FIDS");
    values.Set("VALUES");
    displays.Set("DISPLAYS");

    elementEntryDef.Name = fids;

    elementEntryDef.Type = DataEnum.Array;
```

```

    elementListDefWit.Bind(elementEntryDef);    // bind def to current position of the elementListDef

    elementEntryDef.Name = values;
    elementEntryDef.Type = DataEnum.Array;
    elementListDefWit.Bind(elementEntryDef);    // bind def to current position of the elementListDef

    elementEntryDef.Name = displays;
    elementEntryDef.Type = DataEnum.Array;
    elementListDefWit.Bind(elementEntryDef);    // bind def to current position of the elementListDef

    elementListDefWit.Complete();    // complete encoding of the elementListDef
}

```

Example 21: Encoding ElementListDef

6.2.1.2.2 Encoding FieldList Definition

Like an [ElementList](#), a [FieldList](#) also can contain Standard Data or Defined Data. For Defined Data, [FieldListDef](#) describes the defined data. A [FieldList](#) is a Container of flexible self-describing [FieldEntry](#). [FieldEntryDef](#) is the definition for a [FieldEntry](#). Encoding a [FieldList](#) definition is similar to encoding an [ElementList](#) definition

The following example illustrates how to create a [FieldListDef](#) object.

```

void EncodeFieldListDef()
{
    FieldListDefWriteIterator fieldListWit = new FieldListDefWriteIterator();
    FieldListDef fieldListDef = new FieldListDef();
    FieldEntryDef fieldEntryDef = new FieldEntryDef();

    fieldListDef.DataDefID = 1;

    fieldListWit.Start(fieldListDef);

    fieldEntryDef.FieldID = 1;
    fieldEntryDef.Type = DataBuffer.DataBufferEnum.StringAscii;
    fieldListWit.Bind(fieldEntryDef);

    fieldEntryDef.FieldID = 2;
    fieldEntryDef.Type = DataBuffer.DataBufferEnum.Real;
    fieldListWit.Bind(fieldEntryDef);

    fieldEntryDef.FieldID = 3;
    fieldEntryDef.Type = DataBuffer.DataBufferEnum.Real;
    fieldListWit.Bind(fieldEntryDef);

    fieldListWit.Complete();
}

```

Example 22: Encoding FieldListDef

6.2.1.3 Encoding DataBuffer

The `DataBuffer` is a Leaf and represents base data types. Base data types include types such as `UInt` and simple interfaces such as `Date`, `Time` and `DateTime`. The raw content is encoded into `DataBuffer` using methods and properties.

The true encoding type is defined by `DataBufferEncodedEnum` (`Bit8Value`, `Bit16Value`...). It is passed as an argument for the `SetXXX` methods or `Set` properties.

```
DataBuffer dataBuffer = new DataBuffer();
int fid = 30;
dataBuffer.SetInt(fid, Reuters.RFA.Data.DataBufferEncodedEnum.Bit16Value);
```

Example 23: Encoding DataBuffer

The type of the `DataBuffer` content is indicated using `DataBufferEnum` (`UInt`, `Buffer`, `StringAscii`...) and can be set in the following ways:

- The `Set` property implementation internally sets the `DataBuffer` type when setting the value.

```
Real real = new Real();
real.MagnitudeType = MagnitudeTypeEnum.ExponentNeg2;
real.Value = 3235;
dataBuffer.Real = real;    //internal set the DataBuffer type as Real.
```

Example 24: Encoding DataBuffer

- The `DataBuffer` type can be passed as an argument to the `SetXXX` method. See the *RFA Reference Manual .NET Edition* for details on the `DataBuffer` types that can be passed as arguments to the `SetFromString()` and `SetBuffer()` methods.

```
RFA_String vendorName = new RFA_String();
vendorName.Set("Reuters");
dataBuffer.SetFromString(vendorName, DataBufferEnum.StringAscii);
```

Example 25: Encoding DataBuffer

An application may specify a blank `DataBuffer` type using `SetBlankData()`.

6.2.1.4 Encoding Array

Array is a Container of implicitly position-oriented Entries. An **Array** Container may house variable-width or fixed-width. Variable-width implying the data contained in the Array Entries may be of different sizes, whereas fixed-width implying the data contained in the Array Entries is of the same size. An application needs to call both the **width** and **IndicationMask** properties to set the fixed size of the data contained across the Array Entries.

The following are examples of encoding **Array**.

```
void EncodeArray()
{
    ArrayWriteIterator arrWIt = new
        ArrayWriteIterator();
    Array array = new Array();
    ArrayEntry arrayEntry = new ArrayEntry();
    DataBuffer dataBuffer = new DataBuffer();
    uint mType;

    arrWIt.Start(array);

    // Specify Dictionary as a capability
    mType =
        RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DICTIONARY;

    dataBuffer.UInt = (UInt32)mType;
    arrayEntry.Data = dataBuffer;
    arrWIt.Bind(arrayEntry);

    // Specify Market Price as a capability
    mType =
        RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE;

    arrayEntry.Clear();    //
    dataBuffer.UInt = mType as UInt32;
    arrayEntry.Data = dataBuffer;
    arrWIt.Bind(arrayEntry);

    arrWIt.Complete();
}
```

**Example 26: Encoding Array with
ArrayWriterIterator**

```
void EncodeArray()
{
    SingleWriteIterator swIt = new
        SingleWriteIterator();
    Array array = new Array();
    ArrayEntry arrayEntry = new ArrayEntry();
    uint mType;

    swIt.Initialize(array);
    swIt.Start(array, DataBuffer.DataBufferEnum.UInt);

    // Specify Dictionary as a capability
    mType =
        RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DICTIONARY;

    swIt.Bind(arrayEntry);
    swIt.SetUInt(mType);

    // Specify Market Price as a capability
    mType =
        RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE;

    arrayEntry.Clear();
    swIt.Bind(arrayEntry);
    swIt.SetUInt(mType);

    swIt.Complete();
}
```

**Example 27: Encoding Array with
SingleWriterIterator**

6.2.1.5 Encoding ElementList

An `ElementList` contains self-describing Element Entries. The data type in each `ElementEntry` may differ. Each `ElementEntry` includes Name and Data. The Data is either a Data type or `DataBuffer` type. Encoding an `ElementEntry` is done by calling the `Name` and `Data` properties.

If used, the `ElementListNum` must be set before calling the `ElementListWriteIterator.Start()` method as well.

The following are examples of encoding `ElementList`.

```
Void EncodeElementList()
{
    ElementListWriteIterator elWit = new
    ElementListWriteIterator();
    ElementList eleList = new ElementList();
    ElementEntry eleEntry = new ElementEntry();
    DataBuffer dataBuffer = new DataBuffer();
    RFA_String nameStr = new RFA_String();
    RFA_String valStr = new RFA_String();

    elWit.Start(eleList); // ElementList contain
        StandardData

    nameStr.Set("ServiceName");
    eleEntry.Name = nameStr;
    valStr.Set("DIRECT_FEED");
    dataBuffer.SetFromString(valStr,
        DataBuffer.DataBufferEnum.StringAscii);

    eleEntry.Data = dataBuffer;
    elWit.Bind(eleEntry); // bind pre-encoded element

    eleEntry.Clear();
    nameStr.Set("ProviderName");
    eleEntry.Name = nameStr;
    valStr.Set("Reuter");
    dataBuffer.SetFromString(valStr,
        DataBuffer.DataBufferEnum.StringAscii);

    eleEntry.Data = dataBuffer;
    elWit.Bind(eleEntry); // bind pre-encoded element

    elWit.Complete();
}
```

**Example 28: Encoding ElementList with
ElementListWriteIterator**

```
Void EncodeElementList()
{
    SingleWriteIterator swIt = new
    SingleWriteIterator();
    ElementList eleList = new ElementList();
    ElementEntry eleEntry = new ElementEntry();

    RFA_String nameStr = new RFA_String();
    RFA_String valStr = new RFA_String();

    swIt.Initialize(eleList);
    swIt.Start(eleList); // ElementList contain
        StandardData

    nameStr.Set("ServiceName");
    eleEntry.Name = nameStr;
    valStr.Set("DIRECT_FEED");

    swIt.Bind(eleEntry); // bind post-encoded element
    swIt.SetString(valStr,
        DataBuffer.DataBufferEnum.StringAscii);

    eleEntry.Clear();
    nameStr.Set("ProviderName");
    eleEntry.Name = nameStr;
    valStr.Set("Reuter");

    swIt.Bind(eleEntry); // bind post-encoded element
    swIt.SetString(valStr,
        DataBuffer.DataBufferEnum.StringAscii);

    swIt.Complete();
}
```

**Example 29: Encoding ElementList with
SingleListWriteIterator**

If `ElementList` contains Defined Data, call the `DataDefID` property to set an `ElementListDef` identifier first before calling the `ElementListWriteIterator.Start()` or `SingleWriteIterator.Start()` method following the example below. This example uses the `elementListDef` variable in section **Error! Reference source not found..2.1.2.1 Encoding ElementList Definition**.

```
elementList.DataDefID = 1;

elWIter.Start(elementList,
    IteratorMaskFlag.DefinedData, elementListDef);
    // the type of data content is DefinedData

// encode element here
...
// encode element here
...

elWIter.Complete();
```

Example 30: Encoding ElementList and ElementListDef with ElementListWriteIterator

```
elementList.DataDefID = 1;

swIter.Start(elementList,
    IteratorMaskFlag.DefinedData, elementListDef);
    // the type of data content is DefinedData

// encode element here
...

// encode element here
...

swIter.Complete();
```

Example 31: Encoding ElementList and ElementListDef with SingleListWriteIterator

6.2.1.6 Encoding FieldList

Unlike `ElementList`, a `FieldList` has the `SetInfo()` method to allow an application to set the dictionary ID and the `FieldList` Template Number for standard data. The dictionary which the dictionary ID refers to contains the information to help to obtain standard data in each `FieldEntry`. The `FieldList`'s `SetInfo()` and `DataDefID` property have to be called before calling the `WriteIterator.Start()` method.

Unlike `ElementEntry`, a `FieldEntry` has `FieldID` (not Name) and Data. `FieldID` is a `FieldEntry` identifier.

The following are examples of encoding `FieldList`.

```
void EncodeFieldList()
{
    FieldListWriteIterator fieldListWit = new
    FieldListWriteIterator();
    FieldList fieldList = new FieldList();
    FieldEntry fieldEntry = new FieldEntry();
    DataBuffer dataBuffer = new DataBuffer();
    Reuters.RFA.Data.Real trdPrice = new Real();
    trdPrice.Value = 100;
    trdPrice.MagnitudeType =
        MagnitudeTypeEnum.ExponentNeg2;

    fieldListWit.Start(fieldList); // FieldList contains
        StandardData

    dataBuffer.Int = 9;
    fieldEntry.FieldID = 2; // RDNDISPLAY
    fieldEntry.Data = dataBuffer;
    fieldListWit.Bind(fieldEntry);

    dataBuffer.Clear();
    fieldEntry.Clear();
    dataBuffer.Real = trdPrice;
    fieldEntry.FieldID = 6; // TRDPRC_1
    fieldEntry.Data = dataBuffer;
    fieldListWit.Bind(fieldEntry);

    fieldListWit.Complete();
}
```

Example 32: Encoding FieldList with FieldListWriteIterator

```
void EncodeFieldList()
{
    SingleWriteIterator swIt = new
    SingleWriteIterator();
    FieldList fieldList = new FieldList();
    FieldEntry fieldEntry = new FieldEntry();

    swIt.Initialize(fieldList);
    swIt.Start(fieldList); // FieldList contains
        StandardData

    fieldEntry.FieldID = 2; // RDNDISPLAY
    swIt.Bind(fieldEntry);
    swIt.SetInt(9);

    fieldEntry.Clear();
    fieldEntry.FieldID = 6; // TRDPRC_1
    swIt.Bind(fieldEntry);
    swIt.SetReal(100, MagnitudeTypeEnum.ExponentNeg2);

    swIt.Complete();
}
```

Example 33: Encoding FieldList with SingleWriteIterator

If [FieldList](#) contains Defined Data, call the [DataDefID](#) property to set an [FieldListDef](#) identifier first before calling the [FieldListWriteIterator.Start\(\)](#) or [SingleWriteIterator.Start\(\)](#) method following the examples below. The example uses the `fieldListDef` variable in section 6.2.1.2.2, Encoding FieldList Definition.

```
FieldList.DataDefID = 1;

fieldListWriter.Start(fieldList,
    IterationMaskFlag.DefinedData, fieldListDef);
    // the type of data content is DefinedData
// encode fields
...
// encode fields
...
fieldListWriter.Complete();
```

Example 34: Encoding FieldList and FieldListDef with FieldListWriteIterator

```
fieldList.DataDefID = 1;

swiMarketPrice.Start(elementList,
    IterationMaskFlag.DefinedData, fieldListDef);
    // the type of data content is DefinedData
// encode fields
...
// encode fields
...
swiMarketPrice.Complete();
```

Example 35: Encoding FieldList and FieldListDef with SingleWriteIterator

6.2.1.7 Encoding FilterList

[FilterList](#) is a simpler Container that does not support Defined Data, Summary Data, except for Permission Data and Entries. [FilterList](#) can be a homogeneous container or a heterogeneous container. If the [FilterList](#) is a homogeneous container, the client application can call the [DefaultType](#) property to set the default type of data across all Entries. The type may be a Data Type or a DataBuffer type.

[FilterEntry](#) has an ***Identifier*** property which is a numeric known as a [FilterID](#). The [FilterID](#) corresponds to the binary value of a selectable flag specified by a consumer on the [AttribInfo.HintMaskFlags.DataMask](#).

The following are examples of encoding `FilterList`.

```
void EncodeFilterList(FilterList filterList,
    RFA_String svcName, RFA_String vendName,
    QualityOfService QoS)
{
    FilterListWriteIterator filterListWIt = new
    FilterListWriteIterator();
    FilterList filterList = new FilterList();
    FilterEntry filterEntry = new FilterEntry();

    filterListWIt.Start(filterList);

    filterList.TotalCountHint = 2; // Specify 2 filter
    entries

    filterEntry.FilterId =
        RDM.Directory.SERVICE_DIRECTORY_FILTER_IDS.SER
        VICE_
        INFO_ID;
    filterEntry.Action = FilterEntry.ActionEnum.Set;

    //Encode ElementList for Service Info

    filterEntry.Data = elementList as
        Reuters.RFA.Common.Data;
    filterListWIt.Bind(filterEntry);

    filterEntry.Clear();
    filterEntry.FilterId =
        RDM.Directory.SERVICE_DIRECTORY_FILTER_IDS.SER
        VICE_
        STATE_ID;
    filterEntry.Action = FilterEntry.ActionEnum.Set;

    //Encode ElementList for Service State

    filterEntry.Data = elementList as
        Reuters.RFA.Common.Data;
    filterListWIt.Bind(filterEntry);

    filterListWIt.Complete();
}
```

Example 36: Encoding FilterList with FilterListWriteIterator

```
void EncodeFilterList(FilterList filterList,
    RFA_String svcName, RFA_String vendName,
    QualityOfService QoS)
{
    SingleWriteIterator swIt = new
    SingleWriteIterator();
    FilterList filterList = new FilterList();
    FilterEntry filterEntry = new FilterEntry();

    swIt.Initialize(filterList);
    swIt.Start(filterList, DataEnum.ElementList);

    filterList.TotalCountHint = 2; // Specify 2 filter
    entries

    filterEntry.FilterId =
        RDM.Directory.SERVICE_DIRECTORY_FILTER_IDS.SERVICE_
        INFO_ID;
    filterEntry.Action = FilterEntry.ActionEnum.Set;

    //Encode ElementList for Service Info

    swIt.Bind(filterEntry);
    swIt.SetData(elementList as
        Reuters.RFA.Common.Data);

    filterEntry.Clear();
    filterEntry.FilterId = RDM.Directory.
        SERVICE_DIRECTORY_FILTER_IDS.SERVICE_STATE_ID;
    filterEntry.Action = FilterEntry.ActionEnum.Set;

    //Encode ElementList for Service State

    swIt.Bind(filterEntry);
    swIt.SetData(elementList as
        Reuters.RFA.Common.Data);

    swIt.Complete();
}
```

Example 37: Encoding FilterList with SingleWriteIterator

6.2.1.8 Encoding Map

A `Map` is a Container of manipulable associative key-value pair entries each known as a `MapEntry`. The data type in each `MapEntry` has to be the same. A `MapEntry` can contain key data, permission data, and load data.

MapEntry contains key data which identifies associated load data. A **MapEntry** key supports homogeneous types. An application can set the **MapEntry** key type by calling the **KeyDataType** property on a **Map**.

MapEntry has a property called **Actions** which are operations on MapEntries within a Map to manage change processing of fragments. MapEntries supporting explicit actions contain an enumeration indicating the explicit action.

The **IndicationMask** enumeration from the **Map** interface indicates specific properties of a **Map**.

PermissionDataPerEntry indication mask from **Map** indicates that the permission data is present in the **MapEntry**. Client application can call **PermissionData** property by passing in a Data Buffer to set permission data on the **MapEntry**.

DataDef mask indicates that a **Map** contains Data Definition list. The application must encode DataDefs first, Summary Data second and Entries last. **DataDefwriteIterator** interface provides the ability to iterate and encode Data Definitions on a Container, such as a **Map**.

SummaryData mask indicates that **Map** houses Summary Data. Summary Data is typically metadata that describes **Map** Entries. Summary Data is always homogeneous with respect to Load Data in the Entries. For example, if a Map's Entries contain Load Data of type **ElementList**, then the Summary Data of the **Map** needs to be an **ElementList**, and encoding the Summary Data is just like encoding an **ElementList**. The **SummaryData** flag must be set in the Container's **IndicationMask** by the application before encoding summary data, if the **SummaryData** is to be set on the Container.

Entries indicationMask indicates that **Map** contains MapEntries. Encoding MapEntries on **Map** is similar to encoding ElementEntries on **ElementList**.

The following piece of code depicts encoding a Map's Data Definition. In this example uses the elementListDef variable from section **Error! Reference source not found..2.1.2.1,Encoding ElementList Definition**.

```
DataDefwriteIterator dwIt = new
    DataDefwriteIterator();

map.IndicationMask = map.IndicationMask |
    Map.IndicationMaskFlag.DataDef;

//Encode ElementListDef
...

dwIt.Start(map);
dwIt.Bind(elementListDef);
dwIt.Complete();
```

Example 38: Encoding Map's Data Definition with DataDefWriteIterator

```
SinglewriteIterator swIt = new SinglewriteIterator();
List<DataDef> defDB = new List<DataDef>();

//Initialize iterator with a map object with a size
swIt.Initialize(map,320000); //iterator will encode
    all
data into this map.

map.IndicationMask = map.IndicationMask |
    Map.IndicationMaskFlag.DataDef;

//Encode ElementListDef
...

defDB.Add(elementListDef);
map.DataDef = defDB;
swIt.Start(map, DataEnum.FieldList);
```

Example 39: Encoding Map's Data Definition with SingleWriteIterator

The following piece of code depicts encoding a Map's Summary Data. For example, Summary Data is encoded after calling the Map's `SummaryData` property

```
void EncodeSummaryData()
{
    FieldListWriteIterator fwIt = new
    FieldListWriteIterator();
    FieldList summary = new FieldList();
    FieldEntry fieldEntry = new FieldEntry();
    DataBuffer dataBuffer = new DataBuffer();

    map.IndicationMask =
        Map.IndicationMaskFlag.SummaryData;
    map.KeyDataType =
        DataBuffer.DataBufferEnum.StringAscii;
    map.KeyFieldID = 0;
    map.TotalCountHint = 3;

    fwIt.Start(summary);

    fieldEntry.FieldID = 1;
    dataBuffer.Int = 64;
    fieldEntry.Data = dataBuffer;
    fwIt.Bind(fieldEntry);

    fieldEntry.FieldID = 15;
    dataBuffer.Enumeration = 840;
    fieldEntry.Data = dataBuffer;
    fwIt.Bind(fieldEntry);

    fwIt.Complete();

    map.SummaryData = summary;
}
```

Example 40: Encoding Map's SummaryData with FieldListWriterterator

```
void EncodeSummaryData()
{
    SinglewriteIterator swIt = new
    SinglewriteIterator();

    FieldList summary = new FieldList();
    FieldEntry fieldEntry = new FieldEntry();

    swIt.Initialize(map, 320000);

    map.IndicationMask =
        Map.IndicationMaskFlag.SummaryData;
    map.KeyDataType =
        DataBuffer.DataBufferEnum.StringAscii;
    map.KeyFieldID = 0;
    map.TotalCountHint = 3;

    swIt.Start(map, DataEnum.FieldList);

    swIt.StartSummaryData();

    swIt.Start(summary);

    fieldEntry.FieldID = 1;
    swIt.Bind(fieldEntry);
    swIt.SetInt(64);

    fieldEntry.FieldID = 15;
    swIt.Bind(fieldEntry);
    swIt.SetEnum(840);

    swIt.Complete();

    swIt.Complete();

    swIt.Complete();
}
```

Example 41: Encoding Map's SummaryData with SingleWriterterator

The following piece of code depicts setting of data in a [Map](#).

```
void EncodeMap()
{
    MapWriteIterator mapWIt = new MapWriteIterator();
    Map map = new Map();
    MapEntry mapEntry = new MapEntry();
    RFA_String svcName = new RFA_String("IDN_RDF");
    DataBuffer keyDataBuffer = new DataBuffer();

    mapWIt.Start(map);

    map.KeyDataType =
        DataBuffer.DataBufferEnum.StringAscii;
    map.TotalCountHint = 1; // Provides a hint to the
                           // consumer about how many MapEntry's are to be
                           // provided

    mapEntry.Action = MapEntry.ActionEnum.Add;
    keyDataBuffer.SetFromString(svcName,
        DataBuffer.DataBufferEnum.StringAscii);
    mapEntry.KeyData = keyDataBuffer;

    // Encode FilterList

    mapEntry.Data = filterList as
        Reuters.RFA.Common.Data;

    mapWIt.Bind(mapEntry);

    mapWIt.Complete();
}
```

Example 42: Encoding Map with MapWriteIterator

```
Void EncodeMap()
{
    SingleWriteIterator swIt = new
        SingleWriteIterator();
    Map map = new Map();
    MapEntry mapEntry = new MapEntry();
    RFA_String svcName = new RFA_String("IDN_RDF");
    DataBuffer keyDataBuffer = new DataBuffer();

    swIt.Initialize(map, 320000);

    map.IndicationMask = Map.IndicationMaskFlag.Entries;
    map.KeyDataType =
        DataBuffer.DataBufferEnum.StringAscii;
    map.TotalCountHint = 1; // Provides a hint to the
                           // consumer about how many MapEntry's are to be
                           // provided

    swIt.Start(map, DataEnum.FilterList);

    mapEntry.Action = MapEntry.ActionEnum.Add;
    keyDataBuffer.SetFromString(svcName,
        DataBuffer.DataBufferEnum.StringAscii);
    mapEntry.KeyData = keyDataBuffer;

    // Encode FilterList

    swIt.Bind(mapEntry);

    swIt.SetData(filterList);

    swIt.Complete();
}
```

Example 43: Encoding Map with SingleWriteIterator

6.2.1.9 Encoding Series

[Series](#) is a Container of accruable repetitive structured Entries each known as [SeriesEntry](#). Like [Map](#), [Series](#) also contains data definitions, Summary Data and Entries except Permission Data.

[SeriesEntry](#) interface does not provide an [Action](#) property because Series is designed as a Container of implicitly indexed entries. Encoding Series and SeriesEntry is similar to encoding Map and Map Entry.

For details on encoding the data definition and summary data, refer to Section 6.2.1.8.

The following piece of code depicts setting of data in a [Series](#).

```
void EncodeSeries()
{
    SeriesWriteIterator seriesWIt = new
    SeriesWriteIterator();
    Series series = new Series();
    SeriesEntry seriesEntry = new SeriesEntry();

    seriesWIt.Start(series);

    //Encode FieldList

    seriesEntry.Data = fieldList;
    seriesWIt.Bind(seriesEntry);

    //Encode FieldList

    seriesEntry.Clear();
    seriesEntry.Data = fieldList;
    seriesWIt.Bind(seriesEntry);

    seriesWIt.Complete();
}
```

Example 44: Encoding Series with SeriesWriteIterator

```
void EncodeSeries()
{
    SinglewriteIterator sWIt = new
    SinglewriteIterator();
    Series series = new Series();
    SeriesEntry seriesEntry = new SeriesEntry();

    sWIt.Initialize(series );

    series.IndicationMask =
    Series.IndicationMaskFlag.Entries;

    sWIt.Start(series, DataEnum.FieldList);

    //Encode FieldList

    sWIt.Bind(seriesEntry);
    sWIt.SetData(fieldList);

    //Encode FieldList

    sWIt.Bind(seriesEntry);
    sWIt.SetData(fieldList);

    sWIt.Complete();
}
```

Example 45: Encoding Series with SingleWriteIterator

6.2.1.10 Encoding Vector

[Vector](#) is a Container of position-oriented highly manipulable Entries each known as a [VectorEntry](#). Like [Map](#), [Vector](#) also contains data definitions, Summary Data, Entries, and Permission Data.

[VectorEntry](#) is indexed by position. Therefore, the [VectorEntry](#) interface provides a more powerful [Action](#) property to manage update processing and reconstruction of fragments. For the supported actions enumeration, refer to the *RFA Reference Manual .NET Edition*.

A client application can set a [VectorEntry](#)'s position by calling the [Position](#) property.

Encoding [Vector](#) and [VectorEntry](#) is similar to encoding [Map](#) and [MapEntry](#). For details on encoding the data definition and summary data, refer to the Section 6.2.1.8.

The following piece of code depicts setting of data in a [Vector](#).

```

void EncodeVector()
{
    VectorWriteIterator vectorWIt = new
    VectorWriteIterator();
    Vector vector = new Vector();
    VectorEntry vectorEntry = new VectorEntry();

    vectorWIt.Start(vector);

    //Encode FieldList

    vectorEntry.Action = VectorEntry.ActionEnum.Set;
    vectorEntry.Data = fieldList;
    vectorWIt.Bind(vectorEntry);

    //Encode FieldList

    vectorEntry.Clear();
    vectorEntry.Action = VectorEntry.ActionEnum.Set;
    vectorEntry.Data = fieldList;
    vectorWIt.Bind(vectorEntry);

    vectorWIt.Complete();
}

```

**Example 46: Encoding Vector with
VectorWriterIterator**

```

Void EncodeVector()
{
    SingleWriteIterator sWIt = new
    SingleWriteIterator();
    Vector vector = new Vector();
    VectorEntry vectorEntry = new VectorEntry();

    sWIt.Initialize(vector);

    vector.IndicationMask =
    Vector.IndicationMaskFlag.Entry;

    sWIt.Start(vector, DataEnum.FieldList);

    //Encode FieldList

    vectorEntry.Action = VectorEntry.ActionEnum.Set;
    sWIt.Bind(vectorEntry);
    sWIt.SetData(fieldList);

    //Encode FieldList

    vectorEntry.Clear();
    vectorEntry.Action = VectorEntry.ActionEnum.Set;
    sWIt.Bind(vectorEntry);
    sWIt.SetData(fieldList);

    sWIt.Complete();
}

```

**Example 47: Encoding Vector with
SingleWriterIterator**

6.2.2 Data Decoding

The data structures embedded in the message flowing between the consumer and provider needs to be decoded to access the raw content. The data structure would be either in the form of a container or a leaf.

6.2.2.1 General Data Decoding Process

The container is made of individual entries. Based on the type of the container, the entry may contain standard data content or defined data content. Decoding of defined data content requires data definitions. Data definitions may be locally scoped or globally scoped. Locally scoped data definitions would reside in the parent container. Globally scoped definitions can be obtained from the application implemented cache.

The Leaf represents simple types of data, simple interfaces, etc. The Data Package provides read iterators for decoding data content and data definitions in containers; and access methods to decode the Leaf.

The decoding process depends upon the characteristics of the data structure and thus varies for the individual data structures. It requires identifying the type of the data structure and decoding based on the type.

The Data package provides two processes to step through each entry in a container as follows:

1. Use `foreach` loop to iterate through every entry in a container. Moreover, every container provides the `GetEnumerator()` method to get an enumerator for iterating through it by using the methods of `IEnumerator`. However, this decoding process does not support decoding `FieldList` and `ElementList` which required globally defined data (`FieldListDef` for `FieldList` and `ElementListDef` for `ElementList`). The following example demonstrates how to use `foreach` loop to iterate through a `FieldList`.

```
foreach (FieldEntry fieldEntry in fieldList)
{
    fieldId = fieldEntry.FieldID;
    RDMFidDef fidDef = rdmFieldDictionary.GetFidDef(fieldId);
    Reuters.RFA.Common.Data dataEntry = fieldEntry.GetData(fidDef.OMMType);
    if (dataEntry.DataType == DataEnum.DataBuffer)
    {
        DataBuffer dataBuffer = dataEntry as DataBuffer;
        Console.WriteLine("\tFieldEntry: {0,-10} {1,-8}\t", fidDef.Name, "(" + fieldId + ")");
        Console.WriteLine(dataBuffer.GetAsString().ToString());
    }
}
```

Example 48: Use foreach loop to iterate through FieldEntry in FieldList

NOTE: Using `foreach` loop can degrade application performance as an enumerator is instantiated at the beginning and released at the end of the loop.

5. Use Read Iterators to decode entries in a container. An iterator can be defined to be a container type-specific or a container type-independent. Container type-independent iterators are called Single Iterators because only one iterator instance is needed to decode the message payload. In general, the application must decode data in the following order:
 1. Identifying the Type of the Data Structure
 2. Identifying the Type of Data properties
 3. Decoding of `DataDefs`;(if any) and
 4. Decoding of `SummaryData`;(if any) and

6. Decode data by using the container-specific read iterator or [SingleReadIterator](#). The following pictures show this order.

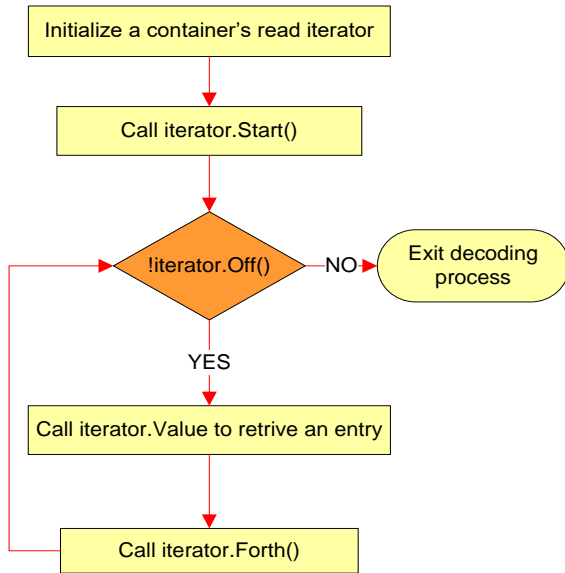


Figure 48: Decoding data with container-specific read iterator

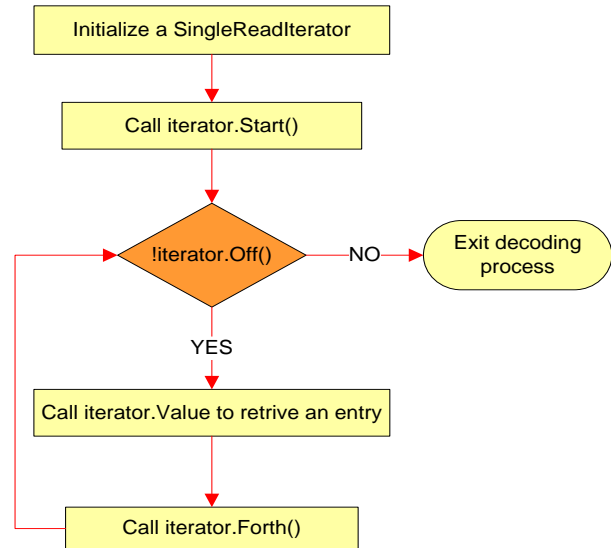


Figure 49: Decoding data with SingleReadIterator

Decoding is done via the use of read iterators. Read iterators generally expose the interfaces listed in the table below. For more interface details, see the *RFA Reference Manual .NET Edition*.

FUNCTION/PROPERTY NAME	DESCRIPTION
Start()	Sets the position to the beginning of the data.
Off()	Returns true if the position exceeds the data.
Forth()	Advances the position forward by one entry.
Value	Returns the DataEntry at the current position.

Table 50: Read Iterator Functions

6.2.2.2 Data Type Identifying

Data represents the raw content and resides either in a Container or in an Entry within a Container. The purpose of the data could be Summary Data, Key Data, Permission Data, or Payload Data.

The data retrieved from a Container or from an Entry needs to be decoded to get the actual content. The simplest form of data structure type to be decoded is the [DataBuffer](#). The [DataBuffer](#) content is obtained using appropriate accessor methods. Other types (e.g., containers like [FilterList](#), [FieldList](#)) have to be recursively decoded until the [DataBuffer](#) is retrieved.

Since data structure can be identified by the `DataType` property, the application can be implemented with switch case in [DecodeData\(\)](#) which allows structure-specific decoding.

```
void DecodeData(Reuters.RFA.Common.Data data)
{
    // decode based on data type
    switch(data.DataType)
    {
        case DataEnum.DataBuffer:
        {
            DataBuffer dataBuffer = data as DataBuffer;
            DecodeDataBuffer(dataBuffer);
            break;
        }
        case DataEnum.Vector:
        {
            Vector vector = data as Vector;
            DecodeVector(vector);
            break;
        }
        case DataEnum.FilterList:
        {
            FilterList filterList = data as FilterList;
            DecodeFilterList(filterList);
            break;
        }
        case DataEnum.FieldList:
        {
            FieldList fieldList = data as FieldList;
            DecodeFieldList(fieldList);
            break;
        }
        case DataEnum.ElementList:
        {
            ElementList elementList = data as ElementList;
            DecodeElementList(elementList);
            break;
        }
        case DataEnum.Series:
        {
            Series series = data as Series;
            DecodeSeries(series);
            break;
        }
        case DataEnum.Map:
        {
            Map map = data as Map;
            DecodeMap(map);
            break;
        }
    }
}
```

```

case DataEnum.Array:
{
    Array array = data as Array;
    DecodeArray(array);
    break;
}
case DataEnum.Msg:
{
    Msg msg = data as Msg;
    DecodeMsg(msg);
    break;
}
case DataEnum.NoData:
{
    Console.WriteLine("Unable to decodeData NoData");
    break;
}
default:
    Console.WriteLine("DecodeData() - Unsupported data type");
    break;
}
}

```

Example 49: Data type identifying

6.2.2.3 Decoding Data Definition

The `ElementListDef` or `FieldListDef` object has to be decoded with `ElementListDefReadIterator` or `FieldListDefReadIterator` interface respectively. The following section depicts how to decode `ElementListDef` and `FieldListDef`.

6.2.2.3.1 Decoding ElementList Definition

The following example illustrates how to decode a `ElementListDef` object.

```

void DecodeElementListDef(ElementListDef elementListDef)
{
    ElementListDefReadIterator elementListDefRit = new ElementListDefReadIterator();

    elementListDefRit.Start(elementListDef);

    while (!elementListDefRit.Off())
    {
        DecodeElementEntryDef(elementListDefRit.Value);
        elementListDefRit.Forth();
    }
    ...
}

```

Example 50: Decoding ElementListDef

6.2.2.3.2 Decoding FieldListDef Definition

The following example illustrates how to decode a `FieldListDef` object.

```
void DecodeFieldListDef(FieldListDef fieldListDef)
{
    FieldListDefReadIterator fieldListDefRit = new FieldListDefReadIterator();

    fieldListDefRit.Start(fieldListDef);

    while (!fieldListDefRit.Off())
    {
        DecodeFieldEntryDef(fieldListDefRit.Value);
        fieldListDefRit.Forth();
    }
}
```

Example 51: Decoding FieldListDef

6.2.2.4 Decoding Data Buffer

The `DataBuffer` is a Leaf and is the simplest form of data structure. It represents the simple types of data (e.g., Integer, String, Date, etc.) The content represented by a `DataBuffer` is retrieved using appropriate accessor methods.

The application implements the function `DecodeDataBuffer()` to decode the content in the `DataBuffer`. This function extracts the content using accessor methods based upon the data type.

The `DataBuffer` type is determined via the `DataBufferType` property. In the example below, the `Int` property extracts the 64-bit integer from the `DataBuffer` and dumps it to the screen.

```
void DecodeDataBuffer(DataBuffer dataBuffer)
{
    switch(dataBuffer.DataBufferType)
    {
        case DataBuffer.DataBufferEnum.Int:
            Console.WriteLine(dataBuffer.Int);
            break;
        case DataBuffer.DataBufferEnum.UInt:
            Console.WriteLine(dataBuffer.UInt);
            break;
        ...
    }
}
```

Example 52: Decoding DataBuffer

For `DataBuffer` types such as strings and buffers, the method `dataBuffer.GetBuffer().ByteArray` is used to extract the data and `GetBuffer().Size` is used to determine the length.

```
case DataBuffer.DataBufferEnum.StringAscii:
case DataBuffer.DataBufferEnum.StringUTF8:
case DataBuffer.DataBufferEnum.StringRMTEs:
{
    RFA_String sData = new RFA_String(dataBuffer.GetBuffer().ByteArray);
    Console.Write(sData.ToString());
    break;
}
```

Example 53: Decoding DataBuffer String type

`DataBuffer` types such as ANSI, XML, and opaque require external parsers to decode the data content; the example application retrieves the buffer containing the data and outputs it to the screen.

```
// Databuffer types requiring external parsers;
// For simplicity, the buffer is output to the screen

case DataBuffer.DataBufferEnum.ANSI_Page:
case DataBuffer.DataBufferEnum.XML:
case DataBuffer.DataBufferEnum.Opaque:
{
    RFA_String sData = new RFA_String(dataBuffer.GetBuffer().ByteArray);
    Console.Write(sData.ToString());
    break;
}
```

Example 54: Decoding DataBuffer ANSI, XML, Opaque type

Certain data types such as Time, Date are represented as interfaces. Appropriate property in the interfaces is used to access the details (e.g. `Time.Second` about the data).

```
case DataBuffer.DataBufferEnum.Time:
{
    Console.Write( dataBuffer.Time.Hour + ":" + dataBuffer.Time.Minute + ":" + dataBuffer.Time.Second);
    break;
}
```

Example 55: Decoding DataBuffer Time type

6.2.2.5 Decoding Array

The `Array` is decoded using the iterator (`ArrayReadIterator` or `SingleReadIterator`). The iterator is initialized by calling `Start()`. A while loop is implemented to iterate through the Entries, extract, and decode them. The `Off()` method, which returns a Boolean, is used to determine whether the iterator has advanced past the end of the data. If the data is valid at the current position, it is extracted using the `Value` property and decoded using `DecodeArrayEntry()` implemented by the application. The iterator is advanced to the next position using `Forth()`.

The example below illustrates decoding of `Array`.

```
void DecodeArray(Reuters.RFA.Data.Array array)
{
    ArrayReadIterator arrayRIt = new
        ArrayReadIterator();

    arrayRIt.Start(array);

    while (!arrayRIt.Off())
    {
        DecodeArrayEntry(arrayRIt.Value);
        arrayRIt.Forth();
    }
}
```

Example 56: Decoding Array with ArrayReadIterator

```
void DecodeArray(Reuters.RFA.Data.Array array)
{
    SingleReadIterator sRIt = new SingleReadIterator();

    sRIt.Start(array);

    while (!sRIt.Off())
    {
        DecodeArrayEntry(sRIt.Value as ArrayEntry);
        sRIt.Forth();
    }
}
```

Example 57: Decoding Array with SingleReadIterator

6.2.2.5.1 Decoding ArrayEntry

Data content is retrieved from the `ArrayEntry` using the `Data` property and decoded using the application-implemented `DecodeData()` function described in section 6.2.2, Data Decoding. The example below illustrates decoding of `ArrayEntry`.

```
void DecodeArrayEntry(ArrayEntry arrayEntry)
{
    DecodeData(arrayEntry.Data);
}
```

Example 58: Decoding ArrayEntry

6.2.2.6 Decoding ElementList

The `ElementList` contains entries that represent either Standard Data content or Defined Data content. Thus, in a Container, all the entries may represent Standard Data content, all the entries may represent Defined Data content, some entries represent Defined Data content, and some entries represent Standard Data content.

Mandatory details for decoding Standard Data content are contained in `ElementEntry`, so additional information is not required.

The following example explains decoding of `ElementList`. The decoding process is similar to that for other Containers except for the `Start()` method which expects additional information; in the form of a mask about the kind of Entry to be retrieved, i.e. Entry with Standard Data content, Entry with Defined Data content or both.

```
void DecodeElementList(ElementList elementList)
{
    ElementListReadIterator eLRit = new
    ElementListReadIterator();

    eLRit.Start(elementList);

    while (!eLRit.Off())
    {
        DecodeElementEntry(eLRit.Value);
        eLRit.Forth();
    }
}
```

**Example 59: Decoding ElementList with
ElementListReadIterator**

```
void DecodeElementList(ElementList elementList)
{
    SingleReadIterator sRit = new SingleReadIterator();

    sRit.Start(elementList);

    while (!sRit.Off())
    {
        DecodeElementEntry(sRit.Value as ElementEntry);
        sRit.Forth();
    }
}
```

**Example 60: Decoding ElementList with
SingleWriteIterator**

Decoding of Defined Data content requires data definitions to process the content. The definitions may either have local scope or global scope. Globally scoped definitions can be obtained from the application's cache. Locally scoped definitions can be obtained from the parent Container `Map`, `Vector` or `Series`.

In the following example, the mask is initialized with `IterationMaskFlag.DefinedData | IterationMaskFlag.StandardData` to retrieve both kind of Entries. This is also the default mask used by the `FieldListReadIterator` and `SingleReadIterator`. The application uses the `ContentMask` property and checks for the presence of Entries with Defined Data content using the `ElementList.ContentMaskFlag.DefinedData` flag. The scope of the data definitions required for decoding the Entries with Defined Data content is obtained from the content mask using `ElementList.ContentMaskFlag.LocalDataDef`. For globally scoped data definitions the application chooses to retrieve only Entries with Standard Data content by setting the mask to `StandardData`.

```
void DecodeElementListDef(ElementList elementList)
{
    ElementListReadIterator eLRit = new
    ElementListReadIterator();

    byte mask = IterationMaskFlag.DefinedData |
                IterationMaskFlag.StandardData;

    if ((elementList.ContentMask &
        ElementList.ContentMaskFlag.DefinedData) != 0
        &&
        (elementList.ContentMask &
        ElementList.ContentMaskFlag.LocalDataDef) !=
        0)
    {
        mask = IterationMaskFlag.DefinedData;
    }

    eLRit.Start(elementList, mask);
    ...
}
```

**Example 61: Decoding DataDef with
ElementListReadIterator**

```
void DecodeElementListDef(ElementList elementList)
{
    SingleReadIterator sRit = new SingleReadIterator();

    byte mask = IterationMaskFlag.DefinedData |
                IterationMaskFlag.StandardData;

    if ((elementList.ContentMask &
        ElementList.ContentMaskFlag.DefinedData) != 0
        &&
        (elementList.ContentMask &
        ElementList.ContentMaskFlag.LocalDataDef) !=
        0)
    {
        mask = IterationMaskFlag.DefinedData;
    }

    sRit.Start(elementList, mask);
    ...
}
```

**Example 62: Decoding DataDef with
SingleReadIterator for ElementList**

6.2.2.6.1 Decoding ElementEntry

The data content is retrieved from an Element Entry using the `Data` property and decoded using the application-implemented `DecodeData()` function described in detail in section 6.2.2, Data Decoding.

```
void DecodeElementEntry(ElementEntry elementEntry)
{
    DecodeData(elementEntry.Data);
}
```

Example 63: Decoding ElementEntry

6.2.2.7 Decoding FieldList

The `FieldList` contains entries that represent either Standard Data content or Defined Data content. Thus, in a Container, all the entries may represent Standard Data content, all the entries may represent Defined Data content, some entries represent Defined Data content, and some entries represent Standard Data content.

Decoding of Standard Data content in a Field Entry requires the data type and can be obtained from a dictionary. If `FieldList.HintMask` indicates the presence of `FieldList.HintMask.Info`, the application needs to retrieve the dictionary ID and field list number using the `getInfoDictID()` and `getInfoFieldListNum()` interfaces before decoding any entries from this `FieldList`.

The example below explains decoding of `FieldList`. The decoding process is similar to that for other Containers except for the `start()` method which expects additional information, in the form of a mask, about the kind of Entry to be retrieved, i.e. Entry with Standard Data content, Entry with Defined Data content or both.

The `start()` method accepts three parameters. The first parameter is the Container to be decoded. The second parameter is a mask specifying the kind of Entry to be retrieved with a default value to retrieve both kinds of Entries. The third parameter is the list of definitions to be used to decode Entries with Defined Data content. This is used for passing global data definitions and has a default value of 0. The application uses the `start()` method and specifies two arguments.

Rest of the decoding is similar to that of other Containers. A while loop is implemented to iterate through the Entries, extract and decode them. The `off()` method, which returns a Boolean, is used to determine whether the iterator has advanced past the end of the data. If the data is valid at the current position, it is extracted using the `value` property and decoded using `DecodeFieldEntry()` implemented by the application. The iterator is advanced to the next position using `Forth()`.

```
void DecodeFieldList(FieldList fieldList)
{
    FieldListReadIterator fieldListRit = new
        FieldListReadIterator();

    fieldListRit.Start(fieldList);

    while (!fieldListRit.Off())
    {
        DecodeFieldEntry(fieldListRit.value);
        it.Forth();
    }
}
```

Example 64: Decoding FieldList with FieldListReadIterator

```
void DecodeFieldList(FieldList fieldList)
{
    SingleReadIterator sRit = new SingleReadIterator();

    sRit.Start(fieldList);

    while (!sRit.Off())
    {
        DecodeFieldEntry(sRit.Value as FieldEntry);
        sRit.Forth();
    }
}
```

Example 65: Decoding FieldList with SingleReadIterator

Decoding Defined Data content requires data definitions to process the content. The definitions may either have local scope or global scope. Globally scoped definitions can be obtained from the application's cache. Locally scoped definitions can be obtained from the parent Container `Map`, `Vector` or `Series`.

In the following example, the mask is initialized with `IterationMaskFlag.DefinedData | IterationMaskFlag.StandardData` to retrieve both kind of Entries. This is also the default mask used by the `FieldListReadIterator` or `SingleReadIterator`. The application uses `ContentMask` property and checks for the presence of Entries with Defined Data content using the `FieldList.ContentMaskFlag.DefinedData` flag. The scope of the data definitions required for decoding the Entries with Defined Data content is obtained from the content mask using `FieldList.ContentMaskFlag.LocalDataDef`. For globally scoped data definitions the application chooses to retrieve only Entries with Standard Data content by setting the mask to `StandardData`.

```
void DecodeFieldList(FieldList fieldList)
{
    FieldListReadIterator fieldListRIt = new
    FieldListReadIterator();

    byte mask = IterationMaskFlag.DefinedData |
        IterationMaskFlag.StandardData;

    if ((fieldList.ContentMask &
        FieldList.ContentMaskFlag.DefinedData) != 0 &&
        (fieldList.ContentMask &
        FieldList.ContentMaskFlag.LocalDataDef) != 0)
    {
        mask = IterationMaskFlag.DefinedData;
    }

    fieldListRIt.Start(fieldList, mask, fieldListDef);
    ...
}
```

Example 66: Decoding FieldList with IteratorMask by FieldListReadIterator

```
void DecodeFieldList(FieldList fieldList)
{
    SingleReadIterator sRIt = new SingleReadIterator();

    byte mask = IterationMaskFlag.DefinedData |
        IterationMaskFlag.StandardData;

    if ((fieldList.ContentMask &
        FieldList.ContentMaskFlag.DefinedData) != 0 &&
        (fieldList.ContentMask &
        FieldList.ContentMaskFlag.LocalDataDef) != 0)
    {
        mask = IterationMaskFlag.DefinedData;
    }

    sRIt.Start(fieldList, mask, fieldListDef);
    ...
}
```

Example 67: Decoding FieldList with IteratorMask by SingleReadIterator

6.2.2.7.1 Decoding FieldEntry

The `FieldEntry` represents either Defined Data content or Standard Data content. Decoding the `FieldEntry` is based upon the content it represents. The content mask is obtained using the `ContentMask` property and contains the kind of the Entry.

Entry representing Defined Data content is indicated by the presence of `FieldEntry.ContentMaskFlag.DefinedData` in the content mask. Data content from such an Entry is retrieved using the `GetData()` or `Data` property.

```
void DecodeFieldEntry(FieldEntry fieldEntry)
{
    if ((input.ContentMask & FieldEntry.ContentMaskFlag.DefinedData) != 0)
    {
        DecodeData(fieldEntry.Data);
    }
    else
    {
        RFA_String fieldName = new RFA_String();
        try
        {
            RDMFidDef fieldDef = rdmFieldDict.GetFidDef(input.FieldID);
            if (fieldDef != null)
            {
                if (fieldDef.FieldId == 0)
                {
                    fieldName.Append(fieldDef.Name);
                    DecodeData(fieldEntry.GetData(DataBuffer.DataBufferEnum.Int));
                }
                else
                {
                    fieldName.Append(fieldDef.Name);
                    DecodeData(fieldEntry.GetData(fieldDef.OMMType));
                }
            }
        }
        catch (InvalidUsageException)
        {
            Console.WriteLine("Field ID" + fieldEntry.FieldId.ToString() + "does not exist in data dictionary");
        }
    }
}
```

Example 68: Decoding FieldEntry

Entries representing standard data content require type of the data to retrieve the actual content. The data type can be obtained from a local dictionary which the application uses for decoding purposes. The data content is retrieved from the entry using the `Data` property and specifying the type as input.

The data content retrieved from an Entry is decoded using the application-implemented `DecodeData()` function described in detail in Section 6.2.2.

6.2.2.8 Decoding FilterList

The `FilterList` is decoded using the iterator (`FilterListReadIterator` or `SingleReadIterator`). The iterator is initialized by calling `Start()`. A while loop is implemented to iterate through the Entries, extract and decode them. The `off()` method, which returns a Boolean, is used to determine whether the iterator has advanced past the end of the data. If the data is valid at the current position, it is extracted using the `Value` property and decoded using `DecodeFilterEntry()` implemented by the application. The iterator is advanced to the next position using `Forth()`.

```
void DecodeFilterList(FilterList filterList)
{
    FilterListReadIterator filterListRit = new
        FilterListReadIterator();

    filterListRit.Start(filterList);

    while (!filterListRit.off())
    {
        DecodeFilterEntry(filterListRit.Value);
        filterListRit.Forth();
    }
}
```

Example 69: Decoding FilterList with FilterListReadIterator

```
void DecodeFilterList(FilterList filterList)
{
    SingleReadIterator sRit = new SingleReadIterator();

    sRit.Start(filterList);

    while (!sRit.off())
    {
        DecodeFilterEntry(sRit.Value as FilterEntry);
        sRit.Forth();
    }
}
```

Example 70: Decoding FilterList with SingleReadIterator

6.2.2.8.1 Decoding FilterEntry

Data content is retrieved from the `FilterEntry` using the `Data` property and decoded using the application-implemented `DecodeData()` function described in section 6.2.2, Data Decoding.

```
void DecodeFilterEntry(FilterEntry filterEntry)
{
    DecodeData(filterEntry.Data);
}
```

Example 71: Decoding FilterEntry

6.2.2.9 Decoding Map

The **Map** may contain definitions. If the definitions exist, they need to be decoded prior to decoding the data contained in the Entries. The following example illustrates the decoding of **Map**.

In the example below, the hint mask is retrieved using the **IndicationMask** property and checked for the presence of Summary Data using the **Map.IndicationMaskFlag.SummaryData** flag. If **Map** contains Summary Data, it is decoded.

The Map Entries are retrieved using the iterator (**MapReadIterator** or **SingleReadIterator**). The iterator is initialized by calling **Start()** and specifying the Map as the input parameter. A while loop is implemented to iterate through the Entries, extract and decode them. The **off()** method, which returns a Boolean, determines whether the iterator has advanced past the end of the data. If the data is valid at the current position, it is extracted using the **value** property and decoded using **DecodeMapEntry()** implemented by the application. The iterator is advanced to the next position using **Forth()**.

```
void DecodeMap(Map map)
{
    MapReadIterator mapRIt = new MapReadIterator();

    mapRIt.Start(map);

    // decode summary data
    if ((map.IndicationMask &
        Map.IndicationMaskFlag.SummaryData) != 0)
    {
        DecodeData(map.SummaryData);
    }

    // contains data definitions; decode data
    // definitions
    if (map.DataDefCount > 0)
    {
        DecodeDataDef(map);
    }

    while (!mapRIt.Off())
    {
        DecodeMapEntry(mapRIt.Value);
        mapRIt.Forth();
    }
}
```

Example 72: Decoding Map with MapReadIterator

```
void DecodeMap(Map map)
{
    SingleReadIterator sRIt = new SingleReadIterator();

    sRIt.Start(map);

    // decode summary data
    if ((input.IndicationMask &
        Map.IndicationMaskFlag.SummaryData) != 0)
    {
        DecodeData(map.SummaryData);
    }

    // contains data definitions; decode data
    // definitions
    if (map.DataDefCount > 0)
    {
        DecodeDataDef(map.DataDef);
    }

    while (!sRIt.Off())
    {
        DecodeMapEntry(sRIt.Value as MapEntry);
        sRIt.Forth();
    }
}
```

Example 73: Decoding Map with SingleReadIterator

The application uses the method `DataDefCount` property to verify presence of data definitions. If a positive value is returned, it indicates presence of data definitions. The data definitions are decoded using the application implemented `DecodeDataDef()`. The following example illustrates the decoding defined data in `Map`. For the decoding `ElementListDef` and `FieldListDef`, refer to section 6.2.2.3, Decoding Data Definition.

```
void DecodeDataDef(Reuters.RFA.Common.Data data)
{
    DataDefReadIterator defRIt = new
        DataDefReadIterator();

    defRIt.Start(data);

    while (!defRIt.Off())
    {
        DataDef def = defRIt.Value;

        if (def.DefType ==
            DataDef.DefTypeEnum.ElementListDef)
        {
            DecodeElementListDef(def as ElementListDef);
        }
        else
        {
            DecodeFieldListDef(def as FieldListDef);
        }
        defRIt.Forth();
    }
}
```

Example 74: Decoding Map's DataDef with DataDefReadIterator

```
void DecodeDataDef(List<DataDef> listDataDef)
{
    int defCount = listDataDef.Count;

    for (int i = 0; i < defCount; ++i)
    {
        DataDef def = listDataDef[i];

        if (def.DefType ==
            DataDef.DefTypeEnum.ElementListDef)
        {
            DecodeElementListDef(def as ElementListDef);
        }
        else
        {
            DecodeFieldListDef(def as FieldListDef);
        }
    }
}
```

Example 75: Decoding Map's DataDef with List of DataDef

Summary Data is contained in `Map`. It is typical metadata that describes Container's Entries. For example, Summary Data could specify the currency of each Entry's price or rules regarding the resorting of Entries.

In the following example, the indication mask is retrieved from the Container using the `IndicationMask` property and verified for the presence of Summary Data using the `IndicationMaskFlag.SummaryData`. If Summary Data exists, it is retrieved using the `SummaryData` property and decoded in the same manner as the Payload Data. The section "Data Decoding" contains more details on data decoding. The example below illustrates the decoding summary data in `Map`.

```
if ( (map.IndicationMask & Map.IndicationMaskFlag.SummaryData) != 0 )
    DecodeData(map.SummaryData);
```

Example 76: Decoding Summary Data

Key data typically identifies the payload data and is contained in the Map Entry.

NOTE: You must call `KeyData` before calling `Data`, otherwise your operation will not properly decode the payload data.

In the following example, key data is retrieved from a Map using the `keyData` property (and then decoded in the same way as payload data). For further details on decoding data, refer to Section 6.2.2.

```
DecodeData(map.KeyData);
```

Example 77: Decoding Key Data

6.2.2.9.1 Decoding a MapEntry

Data content is retrieved from the Map Entry using the `Data` property and decoded using the application-implemented `DecodeData()` function described in section 6.2.2, Data Decoding.

NOTE: You must call `keyData` before calling `Data`, otherwise your operation will not properly decode the payload data.

```
void DecodeMapEntry(MapEntry mapEntry)
{
    Data keyData = mapEntry.KeyData;
    DecodeData(mapEntry.Data);
}
```

Example 78: Decoding MapEntry

6.2.2.10 Decoding Series

The [Series](#) may contain definitions like [Map](#) and [vector](#). If the definitions exist, they need to be decoded prior to decoding the data contained in the Entries.

The Series Entries are retrieved using the iterator ([SeriesReadIterator](#) or [SingleReadIterator](#)). The iterator is initialized by calling [Start\(\)](#) and specifying the [Series](#) as the input parameter. A while loop is implemented to iterate through the Entries, extract and decode them. The [Off\(\)](#) method, which returns a Boolean, determines whether the iterator has advanced past the end of the data. If the data is valid at the current position, it is extracted using [Value](#) property and decoded using [DecodeSeriesEntry\(\)](#) implemented by the application. The iterator is advanced to the next position using [Forth\(\)](#).

The following example illustrates the decoding of [Series](#).

```
void DecodeSeries(Series series)
{
    SeriesReadIterator serRIt = new
        SeriesReadIterator();

    serRIt.Start(series);

    if ((series.IndicationMask &
        Series.IndicationMaskFlag.SummaryData) != 0)
    {
        DecodeData(series.SummaryData);
    }

    if (series.DataDefCount > 0) // contains data def
    {
        DecodeDataDef(series);
    }

    while (!seriesRIt.Off())
    {
        DecodeSeriesEntry(serRIt.Value);
        serRIt.Forth();
    }
}
```

Example 79: Decoding Series with SeriesReadIterator

```
void DecodeSeries(Series series)
{
    SingleReadIterator sRI = new SingleReadIterator();

    sRI.Start(series);

    if ((series.IndicationMask &
        Series.IndicationMaskFlag.SummaryData) != 0)
    {
        DecodeData(series.SummaryData);
    }

    if (series.DataDefCount > 0) // contains data def
    {
        DecodeDataDef(series.DataDef);
    }

    while (!sRI.Off())
    {
        DecodeSeriesEntry(sRI.Value as SeriesEntry);
        sRI.Forth();
    }
}
```

Example 80: Decoding Series with SingleReadIterator

The application uses the method [DataDefCount](#) property to verify presence of data definitions. If a positive value is returned, it indicates presence of data definitions. The data definitions are decoded using the application implemented [DecodeDataDef\(\)](#). For the example code, refer to the example code of decoding Defined data in [Map](#).

Summary Data is contained in Series. It is typical metadata that describes Container's Entries. For example, Summary Data could specify the currency of each Entry's price or rules regarding the resorting of Entries. For the code example, refer to the example code of decoding summary data in [Map](#).

6.2.2.10.1 Decoding SeriesEntry

Data content is retrieved from the Series Entry using the `Data` property and decoded using the application-implemented `DecodeData()` function described in section 6.2.2, Data Decoding.

```
void DecodeSeriesEntry(SeriesEntry seriesEntry)
{
    DecodeData(seriesEntry.Data);
}
```

Example 81: Decoding SeriesEntry

6.2.2.11 Decoding Vector

The `Vector` may contain definitions like `Map` and `Series`. If the definitions exist, they need to be decoded prior to decoding the data contained in the Entries.

The Vector Entries are retrieved using the iterator (`VectorReadIterator` or `SingleReadIterator`). The iterator is initialized by calling `Start()` and specifying the `Vector` as the input parameter. A while loop is implemented to iterate through the Entries, extract and decode them. The `Off()` method, which returns a Boolean, determines whether the iterator has advanced past the end of the data. If the data is valid at the current position, it is extracted using `Value` property and decoded using `DecodeVectorEntry()` implemented by the application. The iterator is advanced to the next position using `Forth()`.

The following example illustrates the decoding of `Vector`.

```
void DecodeVector(Vector vector)
{
    VectorReadIterator vRit = new VectorReadIterator();

    vRit.Start(vector);

    if ((vector.IndicationMask &
        Vector.IndicationMaskFlag.SummaryData) != 0)
    {
        DecodeData(vector.SummaryData);
    }

    if (vector.DataDefCount > 0) // contains data
        definitions
    {
        DecodeDataDef(vector);
    }

    while (!vRit.Off())
    {
        DecodeVectorEntry(vRit.Value);
        vRit.Forth();
    }
}
```

Example 82: Decoding Vector with VectorReadIterator

```
void DecodeVector(Vector vector)
{
    SingleReadIterator sRit = new SingleReadIterator();

    sRit.Start(vector);

    if ((vector.IndicationMask &
        Vector.IndicationMaskFlag.SummaryData) != 0)
    {
        DecodeData(vector.SummaryData);
    }

    if (vector.DataDefCount > 0) // contains data
        definitions
    {
        DecodeDataDef(vector.DataDef);
    }

    while (!sRit.Off())
    {
        DecodeVectorEntry(sRit.Value as VectorEntry);
        sRit.Forth();
    }
}
```

Example 83: Decoding Vector with SingleReadIterator

The application uses the method [DataDefCount](#) property to verify presence of data definitions. If a positive value is returned, it indicates presence of data definitions. The data definitions are decoded using the application implemented [DecodeDatDef\(\)](#). For the example code, refer to the example code of decoding Defined data in [Map](#).

Summary Data is contained in [Vector](#). It is typically 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. For the example code, refer to the example code of decoding summary data in [Map](#).

6.2.2.11.1 Decoding VectorEntry

Data content is retrieved from the Vector Entry using the [Data](#) property and decoded using the application-implemented [DecodeData\(\)](#) function described in section 6.2.2,Data Decoding.

```
void DecodeVectorEntry(VectorEntry vectorEntry)
{
    DecodeData(vectorEntry.Data);
}
```

Example 84: Decoding VectorEntry

6.2.2.12 Decoding Permission Data

Permission data represents authorization information and maybe contained in a Filter Entry, Map Entry or Vector Entry. Presence of permission data is identified by two flags, the [IndicationMaskFlag.PermissionDataPerEntry](#) residing in the Container and [PermissionData](#) residing in the Entry. The permission data, if present, is retrieved using the [PermissionData](#) property and decoded in the same way as payload data. The section "Data Decoding" contains more details on data decoding.

Chapter 7 Message Package

7.1 Message Package Concepts

The **Message Package** defines the messages that flow between a service provider application and consumer application. A Message is an abstract container of header and raw data. It provides the extensible header information of OMM. All messages include an RDM Message Model type which identifies the specific Message Model for OMM.

The Message Package defines five types of abstract messages: the [ReqMsg](#), [RespMsg](#), [GenericMsg](#), [PostMsg](#), and [AckMsg](#). These message types are used to send and receive requests, responses, generic, post, and acknowledgement messages. These Message interfaces inherit from the abstract [Msg](#) interface defined in the Common Package. Using these interfaces, an application can access OMM header information from the Session Layer. Concrete descendants of Interest Specifications, Events and Commands contain Messages.

Along with an explanation of the five basic message types, the following table defines several Message Package concepts.

CONCEPT	DESCRIPTION
Ack Message	A descendant of the Msg class, the AckMsg flows from a Provider to a Consumer to indicate receipt of a specific message. The acknowledgment contains success or failure (negative acknowledgment or nak) information to the consumer. A consumer can request acknowledgment for a Post Message or a Close Message. The ack message is encapsulated in the OMM Item Event (OMMItemEvent). See sections 7.2.1.4, Encoding Ack Message and 7.2.2.6, Decoding Ack Message for its usage. An Ack Message's attributes can be retrieved using the AttribInfo interface.
AttribInfo	Attributes describing the Message. This is defined by the AttribInfo class.
Message	Interface that contains header information and (optionally) data interfaces. This is defined by the abstract Msg class. All major message types inherit from Msg . The Msg class is a descendant of the Data class.
Request Message	A descendant of the Msg class, it flows from the Consumer to the Provider to express interest in a particular information stream. The request message is encapsulated in the OMM Item Interest Specification (OMMItemIntSpec). For usage see sections 7.2.1.1, Encoding Request Message and 7.2.2.3, Decoding Request Message. Consumers send Request Messages and receive Response Messages. Providers receive the same Request Messages and send the same Response Messages.
Response Message	Response Messages are of type Refresh, Update, or Status. They flow from the Provider to the Consumer. The response message is encapsulated in the OMM Item Event (OMMItemEvent). For usage see sections 7.2.1.2, Encoding Response Message and 7.2.2.4, Decoding Response Message. Consumers send Request Messages and receive Response Messages. Providers receive the same Request Messages and send the same Response Messages.
Generic Message	A descendant of the Msg class that flows in both directions (between Providers and Consumers) as necessary on a per-domain model basis. Generic Message is application-defined messages that can contain any OMM primitive or container. They are also not required to follow the traditional Request/Response data flow. Any Domain can have this type of behaviour: Generic Message behavior is not limited to Market Data-based Domains (such as Market Price, etc). Consumers receive Generic Message encapsulated in an OMM Item Event (OMMItemEvent) and Providers receive Generic Message encapsulated in an OMM Solicited Item Event (OMMSolicitedItemEvent). For its usage see sections 7.2.1.5, Encoding Generic Message and 7.2.2.7, Decoding Generic Message.

CONCEPT	DESCRIPTION
Post Message	<p>A descendant of the Msg class, PostMsg flows from the Consumer to the Provider and carries information the consumer wants to publish. Consumers can push content via a Post Message into any cache within the RTDS. The posting capability, unlike unmanaged publishing or inserts, offers optional acknowledgments per posted message. There are two types of Posts:</p> <ul style="list-style-type: none"> 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 send a Post. 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. <p>The post message is encapsulated in the OMM Solicited Item Event (OMMSolicitedItemEvent). See sections 7.2.1.3, Encoding Post Message and 7.2.2.5, Decoding Post Message for its usage.</p> <p>A Post Message's attributes can be retrieved using the AttribInfo interface.</p>
Manifest	<p>The details describing the information received and sent from the Provider to the Consumer.</p> <p>It is contained in the Response Message and is defined by the Manifest class.</p>
View	<p>Allows a consumer application to specify interest in a specific subset of fields or elements. Any requested fields or elements (and possibly others) are contained in Response Message.</p> <p>See sections 7.2.1.1.2, Encoding Request Message for View and 7.2.1.2.2, Encoding Response Message for View for usage examples..</p>
Batch	<p>Allows a consumer application to specify interest in multiple items through a single Request Message. From the perspective of the RFA consumer interface, an initial stream request with n item names of interest results in n unique item streams in response: one stream for each individual requested item. From the perspective of the RFA provider interface, each unique item specified in a single batch Request Message appears as an individual item request.</p> <p>See section 7.2.1.1.3, Encoding Request Message for Batching Items for usage examples.</p>
Response type (Message Sub-type)	<p>Indicates the type of response and is contained in the RespMsg. The RespType could be a Refresh, Update or Status:</p> <ul style="list-style-type: none"> Refresh: contains all of the data for the item as requested by the consumer. Refresh messages are also sometimes called Images. Update: sent only when there is a change to the item Status: sent whenever there is a change in the status of a request <p>ReqMsgs do not have sub-types.</p>
Response status	Indicates Stream State or Data State. See section 7.1.14, Response Status (Stream States and Data States).
Refresh type	<p>If a refresh message is provided as a response to a Request Message it is referred to as a solicited refresh. If some kind of information change occurs — e.g., an error detected on a stream—an upstream provider can push out a refresh message to downstream consumers. This is referred to as an unsolicited refresh.</p> <p>Typically, solicited refresh messages are delivered only to the requesting consumer application while unsolicited refresh messages are delivered to all consumers receiving the relevant stream.</p>
Refresh fragmentation	The ability for an image to be split across multiple messages.
Multi-part refresh	In the case of refresh fragmentation, the collective set of refreshes forming an image. There is a RefreshComplete indicates in the final refresh message.
Payload Data	The data that is contained in the message.

Table 51: Message Package Concepts

7.1.1 Message Structure Concepts

The provider and consumer applications are communicated with messages. There are five message types in a message package namely Request Message, Response Message, Ack Message, Post Message, and Generic Message. These messages are descendent of the abstract [Msg](#) class that contains header information and possible data interfaces.

The structure of the message is shown in the picture below.

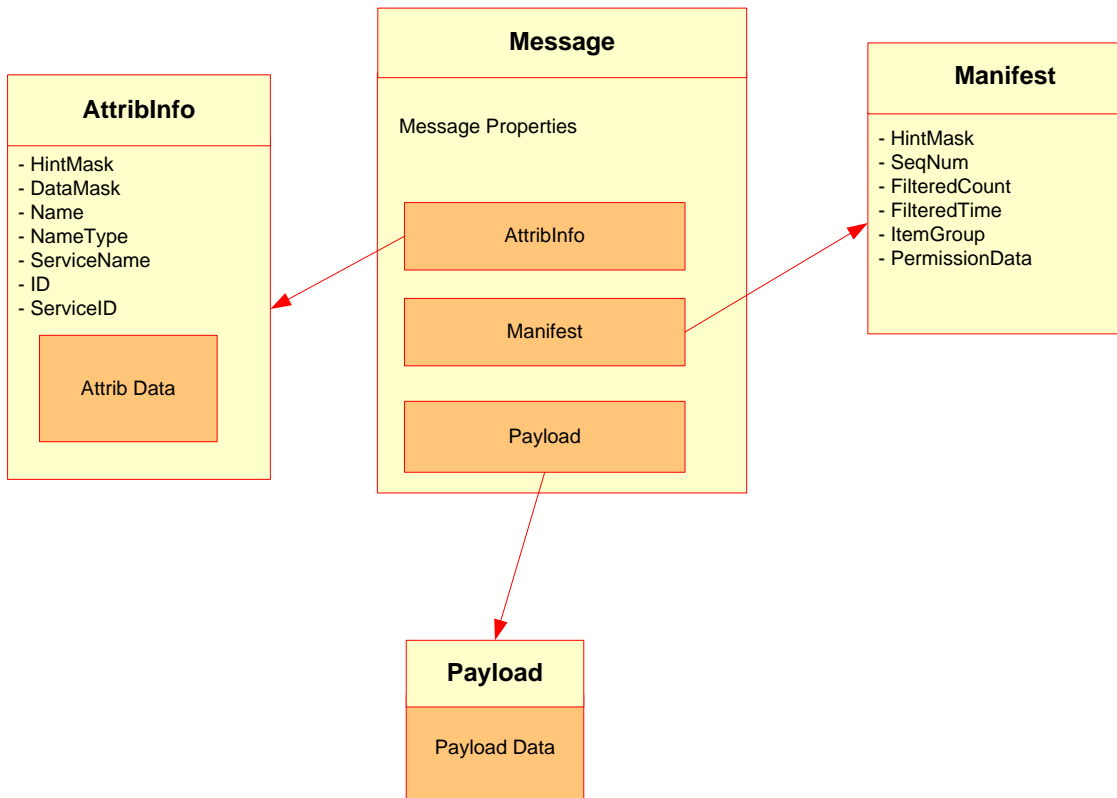


Figure 50: Structure of message

Normally, a message contains four main parts: Message properties, [AttribInfo](#), [Manifest](#), and [Payload](#), but some parts are optional for some message type. The message properties define the properties of each message based on message type. For more details, refer to section 7.1.1.4, Message Type.

7.1.1.1 [AttribInfo](#)

The [AttribInfo](#) defines attributes specified by both Consumer and Provider. In combination with this interface and attributes on the [ReqMsg](#), a Consumer uniquely identifies a request. In combination with this interface and attributes on the [RespMsg](#), a Provider indicates the attributes satisfied for a request from the Consumer. In combination with this interface and attributes on the [GenericMsg](#) a bidirectional Generic Message from a Consumer to a Provider or a Provider to a Consumer can be sent.

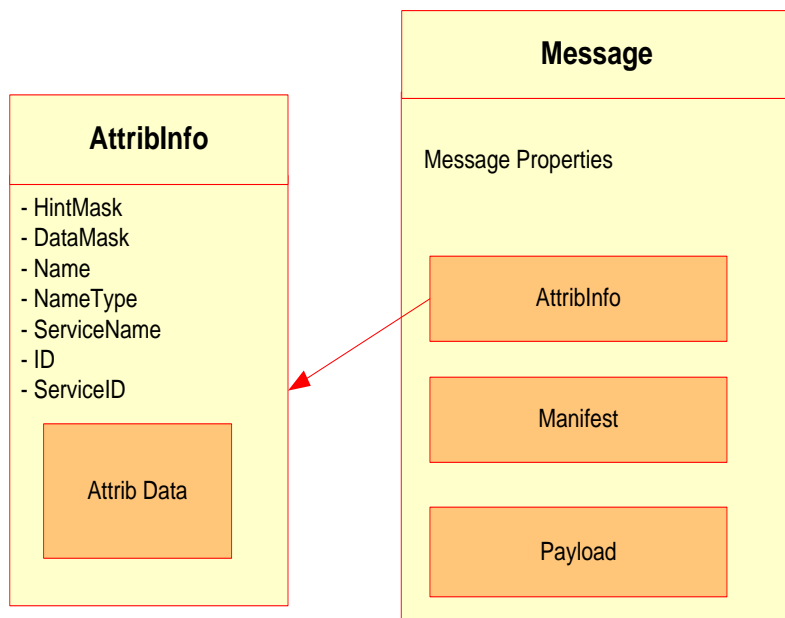


Figure 51: AttrInfo

The **AttrInfo** contains HintMask, DataMask, Name, NameType, ServiceName, ID, ServiceID, and Attr. See the table below for details.

PROPERTY NAME	DESCRIPTION														
HintMask	<p>A mask indicating the contents available on the AttrInfo. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DataMask</td><td>Indicates whether the AttrInfo contains a DataMask.</td></tr> <tr> <td>Name</td><td>Indicates whether the AttrInfo contains a Name.</td></tr> <tr> <td>NameType</td><td>Indicates whether the AttrInfo contains a Name Type.</td></tr> <tr> <td>ServiceName</td><td>Indicates whether the AttrInfo contains a Service Name.</td></tr> <tr> <td>ID</td><td>Indicates whether the AttrInfo contains an Identifier.</td></tr> <tr> <td>Attrib</td><td>Indicates whether the AttrInfo contains extensible attribute data.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	DataMask	Indicates whether the AttrInfo contains a DataMask.	Name	Indicates whether the AttrInfo contains a Name.	NameType	Indicates whether the AttrInfo contains a Name Type.	ServiceName	Indicates whether the AttrInfo contains a Service Name.	ID	Indicates whether the AttrInfo contains an Identifier.	Attrib	Indicates whether the AttrInfo contains extensible attribute data.
HINTMASK FLAG	DESCRIPTION														
DataMask	Indicates whether the AttrInfo contains a DataMask.														
Name	Indicates whether the AttrInfo contains a Name.														
NameType	Indicates whether the AttrInfo contains a Name Type.														
ServiceName	Indicates whether the AttrInfo contains a Service Name.														
ID	Indicates whether the AttrInfo contains an Identifier.														
Attrib	Indicates whether the AttrInfo contains extensible attribute data.														
DataMask	A mask that specifies selectable data to be received on response.														
Name	The name identifier. Examples of semantics are a symbol and a username. Depending on the Message Model type, a Provider can change the name of an open stream by specifying the StreamState of RespStatus to be Redirected. Redirected (RespStatus.StreamStateEnum.Redirected) is also known as a rename.														
NameType	<p>The name type. An example semantic is symbology with one type being a RIC. In order to set the name type, an application needs to ensure that the value set is within the range. If the name type is not set, then it is considered as the name type is equal to 1. For Login Message Model type, the ADS drops the name type in the response message if the name type is set to 1.</p> <p>The specific type and contents of the name should comply with the rules associated with the name type member. Valid Range: $0 \leq x < 256$. Reserved Range: $0 \leq x < 128$.</p>														
ServiceName	The name of the service providing the information.														
ID	A unique ID. Valid Range: $0 \leq x < 4G$. Reserved Range: $0 \leq x < 2G$														
ServiceID	The service ID for the service that provides the information.														

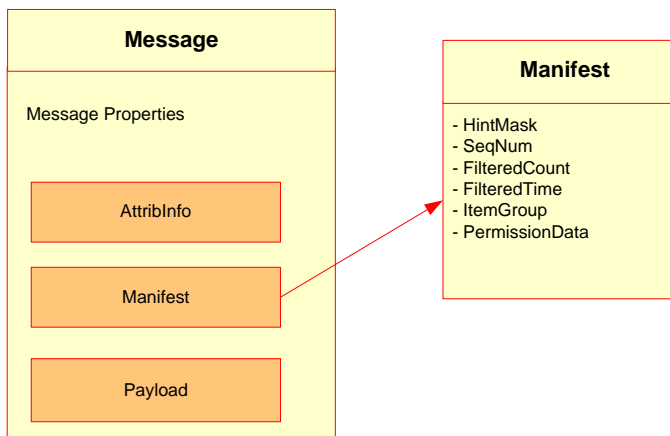
PROPERTY NAME	DESCRIPTION
Attrib	An extensible attribute data property.

Table 52: AttribInfo properties

NOTE: Applications can set the service ID on the [AttribInfo](#) of a message using the property [ServiceID](#). Applications should not set both the service ID and the service name on the [AttribInfo](#) of a [Msg](#) ([ReqMsg](#), [RespMsg](#), [AckMsg](#), [PostMsg](#) and [GenericMsg](#)). An Invalid API usage exception is thrown by RFA if application tries to set both. Provider applications can know the service ID of the service to which the item was requested by using the property [ServiceID](#).

7.1.1.2 Manifest

The [Manifest](#) is a Meta-data sent from a Provider to a Consumer. It contains only manifest properties information with no additional data.

**Figure 52: Manifest**

See the table below for details.

PROPERTY NAME	DESCRIPTION										
HintMask	<p>A mask indicating the contents available on the Manifest. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>Seq</td><td>Indicates whether the Manifest contains a sequence number.</td></tr> <tr> <td>Filtered</td><td>Indicates whether the Manifest contains a filtered count and filtered time.</td></tr> <tr> <td>ItemGroup</td><td>Indicates whether the Manifest contains an item group identifier. A Consumer should compare the previous value to detect a change.</td></tr> <tr> <td>PermissionData</td><td>Indicates whether the Manifest contains permission data.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	Seq	Indicates whether the Manifest contains a sequence number.	Filtered	Indicates whether the Manifest contains a filtered count and filtered time.	ItemGroup	Indicates whether the Manifest contains an item group identifier. A Consumer should compare the previous value to detect a change.	PermissionData	Indicates whether the Manifest contains permission data.
HINTMASK FLAG	DESCRIPTION										
Seq	Indicates whether the Manifest contains a sequence number.										
Filtered	Indicates whether the Manifest contains a filtered count and filtered time.										
ItemGroup	Indicates whether the Manifest contains an item group identifier. A Consumer should compare the previous value to detect a change.										
PermissionData	Indicates whether the Manifest contains permission data.										
SeqNum	The sequence number specified by a Provider, which is typically available on Refresh or Update. The SeqNum should typically be increasing to help with temporal ordering, but it may have gaps depending on the sequencing algorithm being used.										
FilteredCount	The data specifying how many updates were conflated. Valid Range: 0 <= x < 64K. Reserved Range: none.										
FilteredTime	The data specifying the length of time in milliseconds, which updates were conflated. Valid Range: 0 <= x < 4G. Reserved Range: none.										
ItemGroup	The group identifier of a message. Depending on the configuration, the group identifier provides a means to manage the data state of multiple items in a single message.										

PROPERTY NAME	DESCRIPTION
PermissionData	The permission data, which contains authorization information associated to the data. Permission data is typically available only on Refresh or Status. A few Message Model types may have permission data in updates. In this case, the permission data solely applies to this particular update.

Table 53: Manifest properties

7.1.1.3 Payload

The payload data is the information that satisfies the business purpose. This information will be associated with each message model type and contained in a data object.

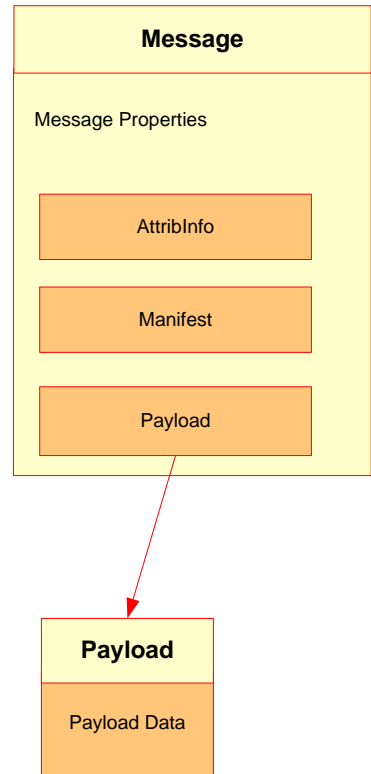


Figure 53: Payload

7.1.1.4 Message Type

7.1.1.4.1 Request Message

A Request Message is sent from a Consumer to a Provider. The request message is used by an OMM Consumer to express interest in a particular information stream. The application creates a Request Message and populates it with the necessary information. The message flows from the Consumer to the Provider and carries the request information. The request message is encapsulated in the OMM Item Interest Specification ([OMMItemIntSpec](#)). Refer to sections 7.2.1.1, Encoding Request Message and 7.2.2.3, Decoding Request Message for its usage.

The [ReqMsg](#) class is a descendant of the [Msg](#) class. Attributes of the request message can be specified using the [AttribInfo](#) property. The interaction behavior can be specified using flags provided in the request message.

Below is the structure of a Request Message.

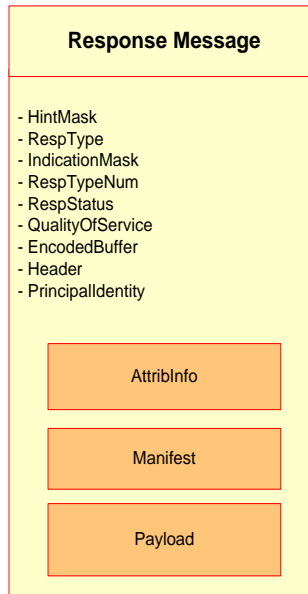


Figure 54: Request Message structure

The table below shows the entire Request Message properties.

PROPERTY NAME	DESCRIPTION												
HintMask	<p>A mask indicating the contents available on the Request Message. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>Priority</td><td>Indicates whether the message contains Priority Class and Priority Count.</td></tr> <tr> <td>QualityOfServiceReq</td><td>Indicates whether the message contains QualityOfServiceReq.</td></tr> <tr> <td>AttribInfo</td><td>Indicates whether the message contains AttribInfo.</td></tr> <tr> <td>Header</td><td>Indicates whether the message contains a header buffer.</td></tr> <tr> <td>Payload</td><td>Indicates whether the message contains payload data.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	Priority	Indicates whether the message contains Priority Class and Priority Count.	QualityOfServiceReq	Indicates whether the message contains QualityOfServiceReq.	AttribInfo	Indicates whether the message contains AttribInfo .	Header	Indicates whether the message contains a header buffer.	Payload	Indicates whether the message contains payload data.
HINTMASK FLAG	DESCRIPTION												
Priority	Indicates whether the message contains Priority Class and Priority Count.												
QualityOfServiceReq	Indicates whether the message contains QualityOfServiceReq.												
AttribInfo	Indicates whether the message contains AttribInfo .												
Header	Indicates whether the message contains a header buffer.												
Payload	Indicates whether the message contains payload data.												

PROPERTY NAME	DESCRIPTION										
IndicationMask	<p>A mask indicating the management of select attributes. Defaults to 0. All possible values of IndicationMaskFlag are as follows:</p> <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>AttribInfoInUpdates</td><td>Indicates whether the Consumer needs Attribute Information in its update messages.</td></tr> <tr> <td>FilteredInUpdates</td><td>When data conflation is taking place, filtered count or rate may be included in the RespMsg Manifest.</td></tr> <tr> <td>View</td><td>Indicates whether a Request Message's payload contains a view definition of Field Ids or Element Names.</td></tr> <tr> <td>Batch</td><td>Indicates whether a Request Message's payload contains a list of item names.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	AttribInfoInUpdates	Indicates whether the Consumer needs Attribute Information in its update messages.	FilteredInUpdates	When data conflation is taking place, filtered count or rate may be included in the RespMsg Manifest .	View	Indicates whether a Request Message's payload contains a view definition of Field Ids or Element Names.	Batch	Indicates whether a Request Message's payload contains a list of item names.
INDICATIONMASK FLAG	DESCRIPTION										
AttribInfoInUpdates	Indicates whether the Consumer needs Attribute Information in its update messages.										
FilteredInUpdates	When data conflation is taking place, filtered count or rate may be included in the RespMsg Manifest .										
View	Indicates whether a Request Message's payload contains a view definition of Field Ids or Element Names.										
Batch	Indicates whether a Request Message's payload contains a list of item names.										
InteractionType	<p>The InteractionType is a consumer-specified mask on the ReqMsg that defines whether the market information is just an initial image or whether there is also an interest after the refresh is complete (i.e., streaming). A Consumer needs to always set at least one of these flags to true, whether initially obtaining interest or changing interest. A Provider may receive a message containing these flags set to false, in the case of a Consumer relinquishing interest. Defaults to InitialImage and the InterestAfterRefresh flags are set to true, and thus is a stream. All possible values of InteractionTypeFlag are as follows:</p> <table> <tr> <th>INTERACTIONTYPE FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>InitialImage</td><td>Specifies an initial image on the response. In combination with updates being true, it indicates the type of interaction is a stream.</td></tr> <tr> <td>InterestAfterRefresh</td><td>Specifies to receive changes (in the form of updates, item status or group status) following the final refresh of a single part or multi-part refresh. In combination with InitialImage being true, it indicates the type of interaction is a stream. In the case that this flag is false, changes may still flow prior to receiving the final refresh of a multiple part refresh.</td></tr> <tr> <td>Pause</td><td>Specifies to pause updates.</td></tr> </table>	INTERACTIONTYPE FLAG	DESCRIPTION	InitialImage	Specifies an initial image on the response. In combination with updates being true, it indicates the type of interaction is a stream.	InterestAfterRefresh	Specifies to receive changes (in the form of updates, item status or group status) following the final refresh of a single part or multi-part refresh. In combination with InitialImage being true, it indicates the type of interaction is a stream. In the case that this flag is false, changes may still flow prior to receiving the final refresh of a multiple part refresh.	Pause	Specifies to pause updates.		
INTERACTIONTYPE FLAG	DESCRIPTION										
InitialImage	Specifies an initial image on the response. In combination with updates being true, it indicates the type of interaction is a stream.										
InterestAfterRefresh	Specifies to receive changes (in the form of updates, item status or group status) following the final refresh of a single part or multi-part refresh. In combination with InitialImage being true, it indicates the type of interaction is a stream. In the case that this flag is false, changes may still flow prior to receiving the final refresh of a multiple part refresh.										
Pause	Specifies to pause updates.										
PriorityClass ¹⁶	A value specifying the relative importance of this stream. The value allows the infrastructure to govern pre-emption. Consumer implementation ensures a Provider application only receives the highest priority class of any merged stream. The PriorityClass can be 1-10. Defaults to 1.										
PriorityCount ¹⁶	The number of downstream users that have interests at the related priority class. The value allows the infrastructure to govern pre-emption. Consumer implementation ensures a Provider application only receives the collective number of users at the highest priority class of any merged stream. The PriorityCount can be 1-65535. Defaults to 1.										
QualityOfServiceRequest	The requested Quality of Service. Defaults to the maximum value in range of requested QualityOfService in both dimensions of timeliness and rate.										
EncodedBuffer	The encoded buffer associated with a valid/complete Request Message. For setting EncodedBuffer property, the application needs to ensure that the Buffer has a valid encoded message.										
Header	A header buffer associated with the request.										
AttribInfo	An attribute information on the request										

¹⁶ A higher priority class value always takes precedence over any priority count value. For example, a stream with a priority class of 5 and priority count of 1 has a higher overall priority than a stream with a priority class of 3 and a priority count of 10,000.

PROPERTY NAME	DESCRIPTION
Payload	A payload data associated with the request.

Table 54: Request Message properties

7.1.1.4.2 Response Message

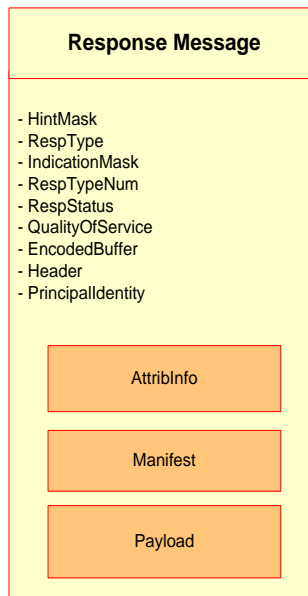
A Response Message is sent from a Provider to a Consumer. The Response Message contains the provider application's response information. In response to a Request Message from the consumer, the provider application creates a Response Message and populates it with the required information. The response message is encapsulated in the OMM Solicited Item Command ([OMMSolicitedItemCmd](#)).

Response Messages for the different Message Models may contain different properties and content. For more details, see the *RFA RDM Usage Guide .NET Edition*.

The [RespMsg](#) class is a descendant of the [Msg](#) class and flows from the Provider to the Consumer. The response message is encapsulated in the OMM Item Event ([OMMItemEvent](#)) and its usage as detailed in Sections 7.2.1.2 and 7.2.2.4.

The response message attributes can be retrieved using the [AttribInfo](#) property. The manifest information of the data received in the response message can be accessed using the [Manifest](#) property.

Below is the structure of a Response Message.

**Figure 55: Response Message structure**

The table below shows the entire Response Message properties.

PROPERTY NAME	DESCRIPTION																		
HintMask	<p>A mask indicating the contents available on the response. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>RespTypeNum</td><td>Indicates whether the message contains RespTypeNum.</td></tr> <tr> <td>RespStatus</td><td>Indicates whether the message contains RespStatus.</td></tr> <tr> <td>QualityOfService</td><td>Indicates whether the message contains QualityOfService.</td></tr> <tr> <td>AttribInfo</td><td>Indicates whether the message contains AttribInfo.</td></tr> <tr> <td>Manifest</td><td>Indicates whether the message contains Manifest.</td></tr> <tr> <td>Header</td><td>Indicates whether the message contains a header buffer.</td></tr> <tr> <td>Payload</td><td>Indicates whether the message contains payload data.</td></tr> <tr> <td>PrincipalIdentity</td><td>Indicates whether the message contains PrincipalIdentity.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	RespTypeNum	Indicates whether the message contains RespTypeNum .	RespStatus	Indicates whether the message contains RespStatus .	QualityOfService	Indicates whether the message contains QualityOfService .	AttribInfo	Indicates whether the message contains AttribInfo .	Manifest	Indicates whether the message contains Manifest .	Header	Indicates whether the message contains a header buffer.	Payload	Indicates whether the message contains payload data.	PrincipalIdentity	Indicates whether the message contains PrincipalIdentity .
HINTMASK FLAG	DESCRIPTION																		
RespTypeNum	Indicates whether the message contains RespTypeNum .																		
RespStatus	Indicates whether the message contains RespStatus .																		
QualityOfService	Indicates whether the message contains QualityOfService .																		
AttribInfo	Indicates whether the message contains AttribInfo .																		
Manifest	Indicates whether the message contains Manifest .																		
Header	Indicates whether the message contains a header buffer.																		
Payload	Indicates whether the message contains payload data.																		
PrincipalIdentity	Indicates whether the message contains PrincipalIdentity .																		
RespType	<p>The type of response. The default value is Refresh. All possible values of RespTypeEnum are as follows:</p> <table> <tr> <th>RESPTYPE ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>Refresh</td><td>Indicates the type of response is a Refresh. An application may receive multiple Refresh responses representing a single item. The IndicationMaskFlag.RefreshComplete indicates the final Refresh.</td></tr> <tr> <td>Status</td><td>Indicates the type of response is a Status. The RespStatus provides details of the Status.</td></tr> <tr> <td>Update</td><td>Indicates the type of response is an Update.</td></tr> </table>	RESPTYPE ENUM	DESCRIPTION	Refresh	Indicates the type of response is a Refresh. An application may receive multiple Refresh responses representing a single item. The IndicationMaskFlag.RefreshComplete indicates the final Refresh.	Status	Indicates the type of response is a Status. The RespStatus provides details of the Status.	Update	Indicates the type of response is an Update.										
RESPTYPE ENUM	DESCRIPTION																		
Refresh	Indicates the type of response is a Refresh. An application may receive multiple Refresh responses representing a single item. The IndicationMaskFlag.RefreshComplete indicates the final Refresh.																		
Status	Indicates the type of response is a Status. The RespStatus provides details of the Status.																		
Update	Indicates the type of response is an Update.																		
IndicationMask	<p>A mask indicating the management of select attributes. Defaults to 0. All possible values of IndicationMaskFlag are as follows:</p> <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>DoNotCache</td><td>Indicates that the message should not be cached.</td></tr> <tr> <td>DoNotFilter</td><td>Indicates that the message should not be conflated.</td></tr> <tr> <td>ClearCache</td><td>Indicates to clear the cache. This condition occurs if the data is known to be invalid.</td></tr> <tr> <td>RefreshComplete</td><td>Indicates whether a refresh is the final refresh. This flag is true in the case of the final refresh of a multi-part refresh or for a single part refresh.</td></tr> <tr> <td>DoNotRipple</td><td>Indicates that the ripple field in an update message should not be rippled.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	DoNotCache	Indicates that the message should not be cached.	DoNotFilter	Indicates that the message should not be conflated.	ClearCache	Indicates to clear the cache. This condition occurs if the data is known to be invalid.	RefreshComplete	Indicates whether a refresh is the final refresh. This flag is true in the case of the final refresh of a multi-part refresh or for a single part refresh.	DoNotRipple	Indicates that the ripple field in an update message should not be rippled.						
INDICATIONMASK FLAG	DESCRIPTION																		
DoNotCache	Indicates that the message should not be cached.																		
DoNotFilter	Indicates that the message should not be conflated.																		
ClearCache	Indicates to clear the cache. This condition occurs if the data is known to be invalid.																		
RefreshComplete	Indicates whether a refresh is the final refresh. This flag is true in the case of the final refresh of a multi-part refresh or for a single part refresh.																		
DoNotRipple	Indicates that the ripple field in an update message should not be rippled.																		
RespTypeNum	Indicate solicited or unsolicited image for Refresh Response Type; and trade, quote, correction or other for Update Response Type.																		
RespStatus	The status associated with the response. The RespStatus has the states DataState and StreamState. RespStatus is typically available only on Refresh or Status.																		
QualityOfService	The QoS specified by a Provider.																		

PROPERTY NAME	DESCRIPTION
AttribInfo	An attribute information associated with the response. AttribInfo is typically available only on Refresh or Status.
Manifest	The details describing the information received and sent from the Provider to the Consumer
Header	A header buffer associated with the response.
Payload	The payload data associated with the response. The payload data is typically available only on Refresh or Update.
EncodedBuffer	An Encoded buffer associated with a valid/complete Response Message. For setting the EncodedBuffer property, the application needs to ensure that the Buffer has a valid encoded message.
PrincipalIdentity	A PrincipalIdentity of the data publisher.

Table 55: Response Message properties

7.1.1.4.3 Post Message

A Post Message is a unidirectional message sent from Consumer to Provider. A Post Message contains the information that a consumer wants to publish. Using RFA's [PostMsg](#), an consumer can publish refresh, update, or status messages, as well as custom-defined data using RFA's [FieldList](#), [ElementList](#), [Map](#), or other container types. To do so, the consumer creates a Post Message and populates it with the necessary message and payload information.

A descendant of the [Msg](#) class, [PostMsg](#) flows from the Consumer to the Provider and carries the information the consumer wants to publish. The post message is encapsulated in the OMM Solicited Item Event ([OMMSolicitedItemEvent](#)). See its usage in sections 7.2.1.3, Encoding Post Message and 7.2.2.5, Decoding Post Message. You can retrieve the post message's attributes using the [AttribInfo](#) property.

Below is the structure of a Post Message.

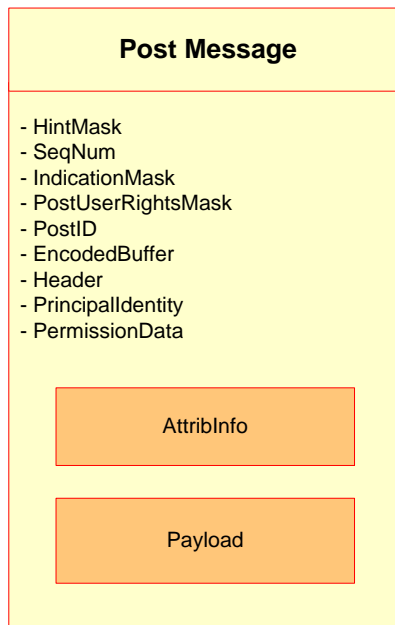


Figure 56: Post Message structure

The table below shows the entire Post Message properties.

PROPERTY NAME	DESCRIPTION																
HintMask	<p>A mask indicating the contents available on the Post Message. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>AttribInfo</td><td>Indicates presence of AttribInfo on this Post Message.</td></tr> <tr> <td>Header</td><td>Indicates presence of message header on this Post Message.</td></tr> <tr> <td>Payload</td><td>Indicates presence of payload on this Post Message.</td></tr> <tr> <td>PostId</td><td>Indicates presence of Post ID on this Post Message.</td></tr> <tr> <td>Seq</td><td>Indicates presence of sequence number on this Post Message.</td></tr> <tr> <td>PermissionData</td><td>Indicates presences of permission data on this Post Message.</td></tr> <tr> <td>PostUserRights</td><td>Indicates presences of user rights on this Post Message.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	AttribInfo	Indicates presence of AttribInfo on this Post Message.	Header	Indicates presence of message header on this Post Message.	Payload	Indicates presence of payload on this Post Message.	PostId	Indicates presence of Post ID on this Post Message.	Seq	Indicates presence of sequence number on this Post Message.	PermissionData	Indicates presences of permission data on this Post Message.	PostUserRights	Indicates presences of user rights on this Post Message.
HINTMASK FLAG	DESCRIPTION																
AttribInfo	Indicates presence of AttribInfo on this Post Message.																
Header	Indicates presence of message header on this Post Message.																
Payload	Indicates presence of payload on this Post Message.																
PostId	Indicates presence of Post ID on this Post Message.																
Seq	Indicates presence of sequence number on this Post Message.																
PermissionData	Indicates presences of permission data on this Post Message.																
PostUserRights	Indicates presences of user rights on this Post Message.																
IndicationMask	<p>A mask that conveys Consumer specified properties of the PostMsg. All possible values of IndicationMaskFlag are as follows:</p> <table> <tr> <th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>MessageInit</td><td>Indicates that this Post Message is the first part of a multi-part Post Message. If the MessageInit and MessageComplete Flags are set at the same time, this indicates that this is a single-part Post Message.</td></tr> <tr> <td>MessageComplete</td><td>Indicates that this is the last part of the multi-part Post Message.</td></tr> <tr> <td>WantAck</td><td>Indicates that application requests acknowledgement to this Post Message.</td></tr> </table>	INDICATIONMASK FLAG	DESCRIPTION	MessageInit	Indicates that this Post Message is the first part of a multi-part Post Message. If the MessageInit and MessageComplete Flags are set at the same time, this indicates that this is a single-part Post Message.	MessageComplete	Indicates that this is the last part of the multi-part Post Message.	WantAck	Indicates that application requests acknowledgement to this Post Message.								
INDICATIONMASK FLAG	DESCRIPTION																
MessageInit	Indicates that this Post Message is the first part of a multi-part Post Message. If the MessageInit and MessageComplete Flags are set at the same time, this indicates that this is a single-part Post Message.																
MessageComplete	Indicates that this is the last part of the multi-part Post Message.																
WantAck	Indicates that application requests acknowledgement to this Post Message.																
PostUserRightsMask	<p>A mask indicating the Post User Rights values. All possible values of PostUserRightsMaskFlag are as follows:</p> <table> <tr> <th>POSTUSERRIGHTSMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>Create</td><td>Indicates that the user is allowed to create records in cache with this post.</td></tr> <tr> <td>Delete</td><td>Indicates that the user is allowed to delete/remove records from cache with this post.</td></tr> <tr> <td>ModifyPermissionData</td><td>Indicates that the user is allowed to modify PermissionData for the records already in cache with this post.</td></tr> </table>	POSTUSERRIGHTSMASK FLAG	DESCRIPTION	Create	Indicates that the user is allowed to create records in cache with this post.	Delete	Indicates that the user is allowed to delete/remove records from cache with this post.	ModifyPermissionData	Indicates that the user is allowed to modify PermissionData for the records already in cache with this post.								
POSTUSERRIGHTSMASK FLAG	DESCRIPTION																
Create	Indicates that the user is allowed to create records in cache with this post.																
Delete	Indicates that the user is allowed to delete/remove records from cache with this post.																
ModifyPermissionData	Indicates that the user is allowed to modify PermissionData for the records already in cache with this post.																
AttribInfo	The AttribInfo associated with this Post Message.																
SeqNum	The sequence number associated with this Post Message. The PostMsg sequence number is for an application's general use. The value is typically ever increasing though it may have gaps. An application may use it to ensure temporal ordering. Additionally, an application may use it in conjunction with PostID to match Post Message with their respective acknowledgements. As it may have gaps, an application should not use it to detect gaps in the stream. Valid Range: $0 \leq x < 4G$. Reserved Range: none.																

PROPERTY NAME	DESCRIPTION
PostID	A Post ID associated with this Post Message. The PostMsg Post ID number uniquely identifies atomic Post Message. An application may send single- or multi-part Post Messages. The single-part Post Message are identified by a unique PostID number (in addition to IndictaionMask setting). The multi-part Post Messages are identified by the same PostID number and unique sequence number. Valid Range: $0 \leq x < 4G$. Reserved Range: none.
PrincipalIdentity	A PrincipalIdentity of the data publisher.
Header	A message's header associated with this Post Message.
Payload	A payload associated with this Post Message. PostMsg may house data or messages with data.
EncodedBuffer	An encoded buffer associated with this Post Message.
PermissionData	The permission data associated with this Post Message.

Table 56: Post Message properties

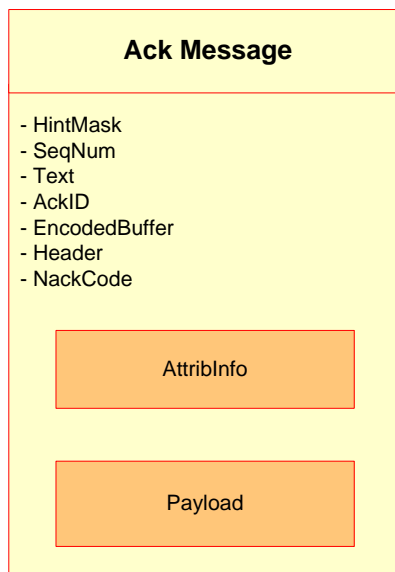
7.1.1.4.4 Ack Message

The Ack message is a unidirectional message sent from a Provider to a Consumer. The Ack message contains a positive or negative acknowledgement of receiving and processing of Post Message. Additionally, it contains the information that allows applications to uniquely match respective Post and Ack Messages. The application creates an Ack Message and populates it with the necessary information.

A descendant of the [Msg](#) class, the [AckMsg](#) flows from the Provider to the Consumer. The ack message is encapsulated in the OMM Item Event ([OMMItemEvent](#)). See its usage in sections 7.2.1.4, Encoding Ack Message and 7.2.2.6, Decoding Ack Message.

You can retrieve the ack message's attributes using the [AttribInfo](#) property.

Below is the structure of an Ack message.

**Figure 57: Ack Message structure**

The table below shows the entire Ack Message properties.

PROPERTY NAME	DESCRIPTION																		
HintMask	<p>A mask indicating the contents available on the Ack Message. All possible values of HintMaskFlag are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>Header</td><td>Indicates presence of message header on this Ack Message.</td></tr> <tr> <td>AttribInfo</td><td>Indicates presence of AttribInfo on this Ack Message.</td></tr> <tr> <td>Payload</td><td>Indicates presence of payload on this Ack Message.</td></tr> <tr> <td>Text</td><td>Indicates presence of text on this Ack Message.</td></tr> <tr> <td>NackCode</td><td>Indicates presence of negative acknowledgement code on this Ack Message. Default is false. If nack code is not present, this indicates that this message is a positive Ack Message.</td></tr> <tr> <td>Seq</td><td>Indicates presence of sequence number on this Ack Message.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	Header	Indicates presence of message header on this Ack Message.	AttribInfo	Indicates presence of AttribInfo on this Ack Message.	Payload	Indicates presence of payload on this Ack Message.	Text	Indicates presence of text on this Ack Message.	NackCode	Indicates presence of negative acknowledgement code on this Ack Message. Default is false. If nack code is not present, this indicates that this message is a positive Ack Message.	Seq	Indicates presence of sequence number on this Ack Message.				
HINTMASK FLAG	DESCRIPTION																		
Header	Indicates presence of message header on this Ack Message.																		
AttribInfo	Indicates presence of AttribInfo on this Ack Message.																		
Payload	Indicates presence of payload on this Ack Message.																		
Text	Indicates presence of text on this Ack Message.																		
NackCode	Indicates presence of negative acknowledgement code on this Ack Message. Default is false. If nack code is not present, this indicates that this message is a positive Ack Message.																		
Seq	Indicates presence of sequence number on this Ack Message.																		
AttribInfo	The Attribute Info associated with this Ack Message.																		
SeqNum	A sequence number associated with this ack message.																		
AckID	Ack ID associated with this Ack Message.																		
Text	Text associated with this Ack Message																		
NackCode	<p>A self describing code indicating a possible reason for negative acknowledgement. Lack of the NackCode indicates a positive acknowledgement. All possible values of NackCodeEnum are as follows:</p> <table> <tr> <th>NACKCODE ENUM</th><th>DESCRIPTION</th></tr> <tr> <td>AccessDenied</td><td>Indicating the negative acknowledgement from access denied.</td></tr> <tr> <td>DeniedBySrc</td><td>Indicating the negative acknowledgement from denied by source.</td></tr> <tr> <td>SourceDown</td><td>Indicating the negative acknowledgement from source down.</td></tr> <tr> <td>SourceUnknown</td><td>Indicating the negative acknowledgement from unknown source.</td></tr> <tr> <td>NoResources</td><td>Indicating the negative acknowledgement from no resources.</td></tr> <tr> <td>NoResponse</td><td>Indicating the negative acknowledgement from no response.</td></tr> <tr> <td>SymbolUnknown</td><td>Indicating the negative acknowledgement from unknown symbol.</td></tr> <tr> <td>NotOpen</td><td>Indicating the negative acknowledgement from not open.</td></tr> </table>	NACKCODE ENUM	DESCRIPTION	AccessDenied	Indicating the negative acknowledgement from access denied.	DeniedBySrc	Indicating the negative acknowledgement from denied by source.	SourceDown	Indicating the negative acknowledgement from source down.	SourceUnknown	Indicating the negative acknowledgement from unknown source.	NoResources	Indicating the negative acknowledgement from no resources.	NoResponse	Indicating the negative acknowledgement from no response.	SymbolUnknown	Indicating the negative acknowledgement from unknown symbol.	NotOpen	Indicating the negative acknowledgement from not open.
NACKCODE ENUM	DESCRIPTION																		
AccessDenied	Indicating the negative acknowledgement from access denied.																		
DeniedBySrc	Indicating the negative acknowledgement from denied by source.																		
SourceDown	Indicating the negative acknowledgement from source down.																		
SourceUnknown	Indicating the negative acknowledgement from unknown source.																		
NoResources	Indicating the negative acknowledgement from no resources.																		
NoResponse	Indicating the negative acknowledgement from no response.																		
SymbolUnknown	Indicating the negative acknowledgement from unknown symbol.																		
NotOpen	Indicating the negative acknowledgement from not open.																		
Header	A header associated with this Ack Message.																		
Payload	A payload data associated with this Ack Message.																		
EncodedBuffer	An Encoded buffer associated with a valid/complete Ack Message. For setting the EncodedBuffer property, the application needs to ensure that the Buffer has a valid encoded message.																		

Table 57: Ack Message properties

7.1.1.4.5 Generic Message

A Generic Message is a bi-directional message sent from a Provider to a Consumer and vice versa. A Generic Message contains an application's information that can be communicated over the network. The application creates a Generic Message and populates it with the necessary information.

The `GenericMsg` class is a descendant of the `Msg` class and flows in both directions (between Providers and Consumers) as is necessary on a per-domain model basis. The Generic Message is encapsulated in the OMM Item Event (`OMMItemEvent`) when it flows from the Provider to the Consumer and is encapsulated in the OMM Solicited Item Event (`OMMSolicitedItemEvent`) when it flows from the Consumer to the Provider. See its usage in sections 7.2.1.5, Encoding Generic Message and 7.2.2.7, Decoding Generic Message.

The generic message attributes can be retrieved using the `AttribInfo` property. `ServiceName` is not used on the generic message `AttribInfo` property.

Below is the structure of a Generic Message.

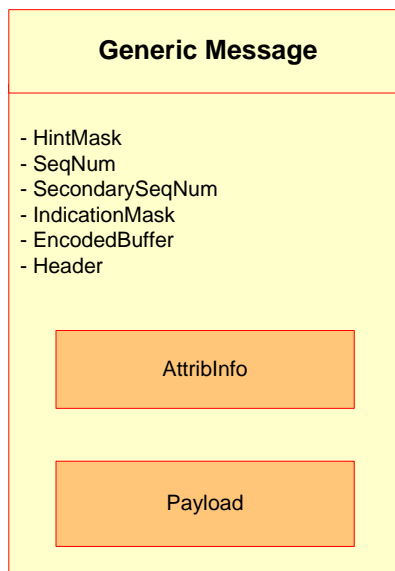


Figure 58: Generic Message structure

The table below shows the entire Generic message properties.

PROPERTY NAME	DESCRIPTION												
SeqNum	A Sequence Number on the generic message.												
SecondarySeqNum	A Secondary Sequence Number on the generic message.												
HintMask	<p>A mask indicating the contents available on the generic message. All possible values of <code>HintMaskFlag</code> are as follows:</p> <table> <tr> <th>HINTMASK FLAG</th><th>DESCRIPTION</th></tr> <tr> <td>Header</td><td>Indicates whether the message contains a Header buffer.</td></tr> <tr> <td>AttribInfo</td><td>Indicates whether the message contains <code>AttribInfo</code>.</td></tr> <tr> <td>Payload</td><td>Indicates whether the message contains <code>payload</code>.</td></tr> <tr> <td>Seq</td><td>Indicates whether the message contains a Sequence number.</td></tr> <tr> <td>SecondarySeq</td><td>Indicates whether the message contains a Secondary Sequence number.</td></tr> </table>	HINTMASK FLAG	DESCRIPTION	Header	Indicates whether the message contains a Header buffer.	AttribInfo	Indicates whether the message contains <code>AttribInfo</code> .	Payload	Indicates whether the message contains <code>payload</code> .	Seq	Indicates whether the message contains a Sequence number.	SecondarySeq	Indicates whether the message contains a Secondary Sequence number.
HINTMASK FLAG	DESCRIPTION												
Header	Indicates whether the message contains a Header buffer.												
AttribInfo	Indicates whether the message contains <code>AttribInfo</code> .												
Payload	Indicates whether the message contains <code>payload</code> .												
Seq	Indicates whether the message contains a Sequence number.												
SecondarySeq	Indicates whether the message contains a Secondary Sequence number.												

PROPERTY NAME	DESCRIPTION				
AttribInfo	An attribute information on the generic message.				
Payload	A payload data associated with the generic message.				
Header	A header buffer associated with the generic message				
EncodedBuffer	An encoded buffer associated with this GenericMsg .				
IndicationMask	<p>A mask indicating the management of select attributes. All possible values of IndicationMaskFlag are as follows:</p> <table><tr><th>INDICATIONMASK FLAG</th><th>DESCRIPTION</th></tr><tr><td>MessageComplete</td><td>Indicates whether this Generic Message is the final portion of the message.</td></tr></table>	INDICATIONMASK FLAG	DESCRIPTION	MessageComplete	Indicates whether this Generic Message is the final portion of the message.
INDICATIONMASK FLAG	DESCRIPTION				
MessageComplete	Indicates whether this Generic Message is the final portion of the message.				

Table 58: Generic Message properties

7.1.2 Symmetric Messaging Paradigm

In combination with the Session Layer Package, the Message Package supports a Symmetric Messaging Paradigm which enables Consumers and Providers to use the same Message Package interfaces. This not only avoids redundant methods on interfaces but also reduces the learning curve when coding between Consumer and Provider. Additionally use of the same message interfaces offer coherent interface usage for hybrid applications.

Consumers send [ReqMsg](#) and receive [RespMsg](#). Providers receive the same [ReqMsg](#) and send the same [RespMsg](#). Hybrid applications receive [ReqMsg](#) from a Consumer and forward the same [ReqMsgs](#) to another Provider. Similarly, a hybrid application receives [RespMsg](#) from a Provider and forwards the same [RespMsg](#) to the Consumer. A bi-directional [GenericMsg](#) from a Consumer to a Provider or from a Provider to a Consumer can be sent over an existing stream. Additionally, Consumers may send [PostMsg](#) and receive [AckMsg](#), while Providers receive [PostMsg](#) and send [AckMsg](#).

Typically a Consumer uses a [ReqMsg](#) to request an initial image and/or request interest after the initial image. A Consumer may also use a [ReqMsg](#) when relinquishing interest or changing the interest specification. Typically a Provider uses a [RespMsg](#) to provide the initial data, changes in data, and status.

To send [GenericMsg](#), the Consumer and the Provider should establish a stream through the request/response process. Once a stream is opened, bi-directional [GenericMsg](#) from a Consumer to a Provider (or vice versa) can be sent over the stream.

7.1.3 Common Message Inheritance

All RFA message types inherit from [Msg](#), which defines several message concepts that are common to all messages. They are explained in more detail in the table below:

MSG MEMBER	DESCRIPTION
MsgType	Identifies the specific type of a message (e.g., RespMsgEnum , ReqMsgEnum , etc.).
MsgModelType	Identifies the specific domain message model type (e.g., MMT_LOGIN , MMT_MARKET_PRICE , etc.). If value is less than 128, domain is a Refinitiv defined domain model. If value is 128 - 255, the domain is a user-defined domain model. Domain model definition is separate 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 RDM Usage Guide .NET Edition</i> .
HintMask	A mask that identifies the optional attributes contained on the interface. For more details, refer to Section 7.1.8.2.
IndicationMask	A mask that describes the management of information. For example, it may indicate whether a ReqMsg 's payload contains a view definition of Field IDs (a View). For more details, refer to Section 7.1.8.3.

Table 59: Common Message Properties

7.1.4 Message Payload and Attribute Data

Messages transport and contain data. The data is hierarchical and extensible. It may be sent from a provider to a consumer or from a consumer to a provider. Data may be payload or attribute data and is specific to the Message Model type. Payload data is typically information that satisfies some business purpose, such as Level I or Level II data. Attribute data is typically additional message attributes. For more details on data, refer to Chapter 6 "Data Package."

The message interface may also contain another message as its payload. [PostMsg](#) and [GenericMsg](#) are the only two message classes allowed to contain other messages.

7.1.5 Refresh Fragmentation

RespMsgs support the capability of refresh fragmentation. Refresh fragmentation is the ability for an image to be split across multiple, independently distributed messages. Each **RespMsg** is known as a refresh. The collective set of refreshes that form an image is called a multi-part refresh.

RFA provides application-level mechanisms for managing refresh fragmentation. These include:

- The identification of the last multi-part refresh
- The ability to easily segment OMM data structures across multi-part refreshes
- Provider-specified hints on the total number of OMM data structures across all multi-part refreshes¹⁷.

The **RespMsg** provides a flag to indicate the final refresh of a multi-part refresh. This flag is known as Refresh Complete. The Refresh Complete flag will also be set if the refresh is not fragmented (i.e., a single-part refresh).

7.1.5.1 Final Refresh Completeness

Updates to the multi-part refresh may arrive before the final refresh is received. However, the refresh is not considered complete until the final refresh of a multi-part refresh is received. The **ReqMsg** provides independent flags to specify a desire to receive an initial image and to receive interest after the final refresh. The **RespMsg** also contains a **RespStatus** that indicates the state of the stream.

Allowing changes prior to the final refresh ensures a consistent point in time for two types of item requests: those that desire interest after the final refresh (i.e., streaming request) and those that do not (i.e., snapshot request). Allowing changes prior to the final refresh also simplifies Providers so that they can flexibly deliver data.

7.1.5.1.1 Final Refresh and Entry Actions

Prior to the final part of a multiple refresh message, a consumer application may decode a container type with an entry-specific Action that it must to apply—for example, an add action. This is true whether the add action occurs in a refresh message or update message. After receiving an add action, a consumer application needs to apply any subsequent change (such as an update action or delete action). On occasion, multiple add actions can occur for the same item. In this case, consumer applications should apply the subsequent add action instead of the previous add action.

As add actions can occur in refresh messages or update messages prior to the final part of a multiple refresh message, provider and hybrid applications can concurrently service downstream applications without regard to whether add actions occur from an upstream provider application or from a local cache. Provider and hybrid applications:

- Forward add actions from an upstream provider application as an update message
- Forward add actions from a local cache as a Response Message

Because consumer applications can accommodate multiple add actions for the same item, the provider and hybrid applications do not need to synchronize update and Response Message processing across thread contexts.

For more details on Entry Actions, refer to section 6.1.7.4, Entry Actions.

¹⁷ The latter two mechanisms are provided through the Data Package (See section 6.1.11.1, Fragmentation for details)

7.1.6 Priority

Each item stream has an associated Priority that is set by the Consumer. A Consumer sends priority to a Provider to enable proper execution of the Provider's pre-emption algorithm. If a Consumer requests more items than the Provider supports, a Provider may either close the new request or close an existing stream to allow the new request.

Priority has two parts: Priority Class and Priority Count. The priority class indicates the general importance of the stream to the consumer. The priority count indicates the streams specific importance within the priority class. A [ReqMsg](#)'s Priority Class and Priority Count can be used to inform the Provider how many users at the highest Priority Class are using a particular item stream. Since the number of users may change over time, the item specification can be reissued¹⁸ with a new priority.

Priority Class takes precedence over Priority Count. For example, a stream with Priority Class 5 and Priority Count 1 takes precedence over a stream with Priority Class 3 and Priority Count 10,000.

The Provider will base its pre-emption decision primarily on the priority of the new request and existing streams, choosing the stream with the lowest priority for pre-emption. If multiple streams share the lowest priority, the item that has been open longest is generally chosen for pre-emption.

7.1.7 Interface Forwarding

Provider and Hybrid¹⁹ applications may need to forward interfaces or contained interfaces to other applications. Assuming the application need not change any attributes of the interface, the application may forward the interface simply by specifying this interface on another containing interface.

For example, a Provider application may choose to forward the contained interface from a [ReqMsg](#) to a [RespMsg](#) (e.g., the [AttribInfo](#) interface). In this example the Provider application may specify the [AttribInfo](#) that was received on a [ReqMsg](#) directly on a [RespMsg](#).

As another example, a Hybrid application may choose to forward a [ReqMsg](#) to another Provider or forward a [RespMsg](#) to an alternate Consumer. In this example the Hybrid application may specify a [ReqMsg](#) that was received from an Event directly on an Interest Specification that accepts a [ReqMsg](#). Similarly a Hybrid application may specify a [RespMsg](#) that was received from an Event directly on a Command that accepts a [RespMsg](#).

If the application wants to change the attributes of an interface it must copy the interface, or the containing interface, which provides the application with a unique instance of the interface whose attributes may now be changed.

7.1.8 Masks

Several of the interfaces in the Message Package (and also the Data Package) use Masks. A Mask is a set of flags that convey information to a Consumer or Provider. Masks differ in that a Consumer, Provider and/or implementation may specify the flags within a Mask.

Obtaining a mask (via a [get](#) property) provides a value containing all flags. Specifying a mask (via a [set](#) property) specifies all flags. Thus specifying a mask overwrites any formal values of the mask. Applications specifying a change to select flags need specify the logical operation to affect only those flags they wish to change.

¹⁸ Requests can also be reissued if the application wishes to change a characteristic of the original request. Reissue can be used for streaming request only.

¹⁹ A hybrid application acts as both a Consumer and Provider.

7.1.8.1 Interaction Type

The [InteractionType](#) is a consumer-specified mask on the [ReqMsg](#) that defines whether the market information is just an initial image or whether there is also interest after the refresh complete (i.e., streaming). The interaction type consists of three flags: InitialImage, InterestAfterRefresh, and Pause. At least one of the three flags must be set to true. The table below depicts the possible combinations for a Consumer and Provider, where the values in the Consumer columns determine what the values in the Provider columns will be.

FLAG	DESCRIPTION	CONSUMER		PROVIDER			
		Specify Initial Interest	Change Interest	Support optimized pause resume	Receive Initial Interest	Receive a Change in Interest	Receive a Relinquish in Interest
Initial Image	Specifies an initial image on the response.	True	True or False	N/A	True	True or False	False
Interest After Refresh	Specifies to receive changes following the final refresh.	True or False	True	N/A	True or False	True	False
Pause	Specifies to pause updates.	False	True	False	False	True	False
		True	True	True	True	True	False

Table 60: InteractionType

7.1.8.2 Hint Mask

The [HintMask](#) is a mask that identifies the optional attributes contained on an interface. Hint Masks reside on several interfaces throughout the Message Package (and also Data Package).

For example before checking the [AttribInfo](#) on a [RespMsg](#), an application can check whether the [AttribInfo](#) exists as shown below:

```
if ((respMsg.HintMask & RespMsg.HintMaskFlag.AttribInfo) != 0)
```

Example 85: Checking Hint Mask

An application should only attempt to obtain an attribute whose associated Hint Mask flag is true. If an application attempts to obtain an attribute whose associated flag is false, the result is an error condition and the data obtained is undefined.

Attributes residing on an interface that do not have an associated Hint Mask flag are not optional and thus are required attributes.

7.1.8.3 Indication Mask

The [IndicationMask](#) is a Consumer-specified or Provider-specified mask that describes the management of information. For example, the Indication Mask contains a flag, [AttribInfoInUpdates](#), for whether the Consumer needs Attribute Information in its update messages. Another example, [FilteredInUpdates](#), indicates whether the type of market information allows caching or time filtering. The Indication Mask may also indicate whether a [ReqMsg](#)'s payload contains a list of item names (a Batch), or whether it contains a view definition of Field IDs (a View), as shown below:

```
reqMsg.IndicationMask |= ReqMsg.IndicationMaskFlag.View;
```

Example 86: Setting IndicationMask

7.1.9 Batching Items in a Request Message

A consumer can specify interest in a list of items with a single Request Message using a Batch Request. This results in each item of interest being fulfilled by a normal item response on a unique stream. After the client receives a response, each item stream is completely independent of the original batch request. All RFA recovery mechanisms can perform on the stream as usual.

A batch request can be used for streaming or non-streaming item requests. All interest specifications specified in a batch Request Message must have the same request attributes. This feature is available for all existing RDM non-administrative²⁰ domains and any user defined domain message models as long as all the items specified in the batch request have the same Message Model Type.

The RFA consumer interface can determine whether a provider supports batch requests from the [SupportBatchRequests](#) element of the login [RespMsg](#)'s [AttribInfo.Attrib](#). If a provider does not support batching, RFA will internally send individual item requests to the provider instead of sending a batch request. For more details on the Login Response Message, refer to the *RFA RDM Usage Guide .NET Edition*.

An application should expect to get a batch handle for the initial batch request and a handle for each item that was requested. The handle that is associated with the batch request is different from a normal item handle. It cannot be used to reissue the batch request because by the time the application has received the response the stream associated with the batch handle is considered closed.

The first response for each item specified in a batch request always includes the item name and the item handle. Clients can use the item handle to identify subsequent Response Messages for the item or perform a reissue on the item stream.

A consumer can specify interest in a list of item handles with a single Request Message using the **Batch Reissue** feature. This feature allows clients to change pause and resume state, view, priority, or request a refresh on multiple item handles. An empty list of item handles passed for the Batch Reissue will apply attribute changes on all open items of non-administrative domain types.

The **Batch Close** feature allows clients to pass a list of item handles to close multiple items with a single close request message. An empty list of item handles passed for the Batch Close will close all open streams except login stream.

The Batching capability works together with RFA's request throttling. For more detail about RFA's throttling, refer to section 13.1.

²⁰ Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. Other domains are used for sending and receiving market information and are considered non-administrative.

7.1.10 Dynamic View

A Dynamic View allows a consumer application to specify a subset of data content that it is interested in. A providing application can choose to supply only this requested subset of content across all Response Messages. This filtering allows for reduced data flow across the connection. View use can be leveraged across all non-administrative domain model types, where specific usage and support should be indicated in the model definition. Although a specific View may be requested, it is possible for additional content to be provided and some content may also be unavailable and not provided.

The View feature is intended to increase the consumer performance in two ways:

- by reducing bandwidth usage through reducing the field list or element list size per Response Message
- by reducing decoding time in the client application through reducing the number of entries per field list or element list.

Views can be used for streaming item requests, non-streaming item requests, or batch requests. In a batch request, a single view specified is applied to all items contained in the batch.

The Consumer defines a View by setting the [IndicationMaskFlag.View](#) of the [ReqMsg](#) and encoding ViewType and ViewData element entries in an element list contained in the payload of the [ReqMsg](#). The [IndicationMaskFlag.view](#) indicates that a View request exists within the request message, and that a View definition is in the payload.

A client can dynamically change a view on an open item stream by reissuing a request with a new view. If there is no [IndicationMaskFlag.View](#) flag with reissued Request Message, the reissued request is for all fields (a full image). If there is a [IndicationMaskFlag.View](#) flag present on a reissued request msg, but no view definition in the 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.

If a failover/reconnect occurs, the previously requested view is recovered automatically by RFA or the RTDS through the regular request caching/recovery scheme. A different view from a reissue request can replace the existing view, and a different view from another initial request on same item will merge the existing view.²¹

The RFA consumer interface can determine whether a provider supports Dynamic Views from the [SupportViewRequests](#) element of the login [RespMsg](#)'s [AttribInfo.Attrib](#). If the provider does not support Views, RFA will internally remove the [IndicationMaskFlag.View](#) flag and send the request to the provider asking for a full image instead of sending the request for the specific view. For more details on the login [RespMsg](#), refer to the *RFA RDM Usage Guide .NET Edition*.

7.1.10.1 Views in RDM/DMM

Each RDM or DMM should describe how a specified View applies to a specific message model type, if at all. A View can be used for RDM or any user-defined domain message model that has pairs of FieldID/FieldValue (or ElementName/ElementValue) as long as the provider understands the View 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 a 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 consumer, but does not guarantee that the view will be fulfilled. A consumer can receive more or less fields than the view set for which it asked. It is the client's responsibility to maintain a view definition for use in filtering subsequent Response Messages if additional fields cannot be processed elsewhere in the consumer.

7.1.10.2 Enhanced Symbol List (RDM SymbolList)

The consumer applications can request data along with the names on a symbol list domain as well as determine supported features from the [SupportEnhancedSymbolList](#) element in the login [RespMsg](#)'s [AttribInfo.Attrib](#). Refer to the *RFA RDM Usage Guide .NET Edition* for more details on the login [RespMsg](#).

²¹ If a consumer requests multiple requests with different views on same item, during an item recovery the last refresh received by the consumer may contain different views in different situations depending on when the requests were sent out and when the refreshes were received.

When data is requested along with names on the symbol list, RFA opens the items listed in the symbol list response for the consumer application. While opening items from the symbol list on the consumer application's behalf, RFA sends these item requests as batch requests if the back-end server supports the **Batch Request** feature. Currently the consumer application can only declare that it wants data but cannot specify the domain for the names in the symbol list. Therefore, RFA will open individual items of the symbol list as MarketPrice domain. Handles to individual items will be provided to the application on the first MarketPrice refresh of the item by RFA.

For example, if the consumer application has requested for data on the symbol list request (0#SIAC) that contains items TRI.N, CSCO, and GE, RFA will then open these three items as MarketPrice requests to the upstream provider. The application gets the handles to these three items on the first response to each item.

When the consumer application opens an item IBM and then makes a symbol list request asking for data that also contains IBM, RFA will then provide a separate handle to the application for IBM that was opened from the symbol list. The consumer application will have two handles for IBM in this case. If updates are received on the symbol list stream, RFA will open only those items that were never opened before. RFA ignores item names from "Delete" or "Update" action of the symbol list stream update.

Re-issue done by the application on a symbol list handle will be applicable only on the symbol list stream and will not be applied to individual items opened by RFA from the symbol list. Suppose the application does a `ReissueClient()` on a symbol list handle for a priority change. This applies only to the symbol list stream, while the priority change will not affect the individual items opened by RFA from the symbol list. Similarly, if the application does a `ReissueClient()` on a symbol list handle for a pause/resume, then only the symbol list stream will be paused/resumed and individual item streams opened from the symbol list by RFA will not be paused/resumed.

When RFA is recovering after a connection goes down and comes back up, individual items of the symbol list and the symbol list are recovered as usual. Similarly, when a service goes down and comes back up, individual items of the symbol list and the symbol list are recovered as usual. If the application requested data while opening a symbol list to a service group, RFA will then open individual items to the same concrete service to which the symbol list request was made.

7.1.11 Item Groups

Item Groups can be used to efficiently update the state of many item streams through the use of a single group message instead of many individual item messages. For example, by using Item Groups, a single `RespMsg` can be used to report that an entire item group has become stale instead of providing a Status `RespMsg` for each affected item.

7.1.11.1 Provider Processing

Each open data stream belongs to an item group. The provider is responsible for assigning data streams to an item group. The item group assignment is set by the Provider in the initial Refresh Response Message. A data stream's item group id is present in the Manifest of the Refresh Response Message. If the provider does not specify a manifest and/or does not assign an item group to an item stream, then the item stream belongs to the default item group. The item group can be modified by the Provider with a Status Response Message or a subsequent Refresh Response Message. Item groups are defined on a per-service basis and may be used across multiple domains provided by a Service. However, item groups are never applied to streams in the Login and Directory domains.

Providers can change the status of an item group using a directory service update message. Using this single message, a provider can change the status of all items in the item group. This technique can save considerable network bandwidth versus sending status messages for multiple items. See the Directory section in the *RFA RDM Usage Guide .NET Edition* for more information.

7.1.11.2 Consumer Processing

By default, the RFA consumer interface tracks item group information sent by the provider for all open streams in watchlist entries. When a directory service update message is received, RFA will use its watchlist to determine which items are impacted by the status update and send a status update message for each of those items to the client application.

If the client would prefer to do this processing, the feature can be disabled in RFA using the configuration parameter `groupStatusFanoutEnabled`.

7.1.11.3 Item GroupId Scenario

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 serviceId is also considered the values should be unique. A Consumer application should track the serviceId/groupId 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 its needs, but should keep in mind that each item group must be 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.

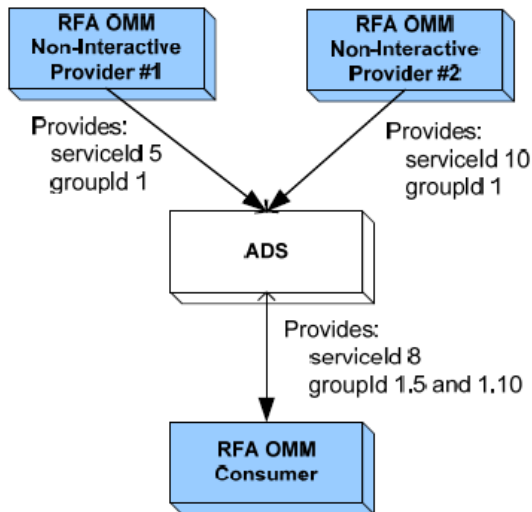


Figure 59: Item Group Example

The above 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 the data into a single service, and distributes the 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 (ADS) has modified serviceId 5, groupId 1 into a groupId of 1.5 and serviceId 10, groupId 1 into a groupId 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 groupId 1.5 are suspect. This would have no impact to any items belonging to groupId 1.10.

Consumer clients that choose to do their own item group processing should treat the data contained in the itemGroup buffer of the directory service update as opaque data that may vary in length. The actual data contained in the itemGroup data buffer is a collection of one or more unsigned 2-byte integer values. Providers that combine multiple data sources must ensure that the item groups in the resulting Service are unique. This can be accomplished by appending a 2-byte value to each itemGroup that is forwarded to ensure uniqueness.

7.1.12 Private Streams

Private Streams allows OMM applications to establish private streams, which are similar to standard streams, except that data exchanged on private streams flows exclusively between two and only two endpoints: the OMM Consumer and OMM Provider.²²

²² Standard streams, in contrast, may be shared between or fanned out to many OMM Consumers.

Data flow on the private stream may be cached or aggregated as defined by the specific domain message model. Private streams and standard streams may be established and maintained within the same application. RFA will not attempt to recover a private stream connection connection that goes down.

Private Streams may be used as a mechanism to exchange any data between two exclusive endpoints, for example, data related to transactions. Such data would be transported by a Generic Message on a private stream.

7.1.13 Visible Publisher Identifier (VPI)

Visible publisher identification in OMM can be found in the `Reuters.RFA.Common.PublisherPrincipalIdentity` class. This class provides both publisherID and publisher address. The VPI in OMM can be obtained from the `Reuters.RFA.Message.RespMsg` and `Reuters.RFA.Message.PostMsg` classes, and both messages contain the `Reuters.RFA.Common.PublisherPrincipalIdentity` class.

The interface for obtaining the VPI values are provided by RFA on the `Reuters.RFA.Message.RespMsg` and `Reuters.RFA.Message.PostMsg` classes. OMM Consumer applications get the VPI from the `Reuters.RFA.Message.RespMsg` class, while OMM Provider applications get the VPI from the `Reuters.RFA.Message.PostMsg` class.

RFA provides the setting of VPI only on the `Reuters.RFA.Message.RespMsg` class. The **OMM publisher** may optionally choose to set the VPI on the response message, or choose not to set the VPI on the response message. If the upstream publisher from which the consumer is consuming is an intermediary device that gets data from its upstream source, then the intermediary device will route 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 it publishes. While processing a post message received from the posting device, the OMM publisher application gets the VPI from the `Reuters.RFA.Message.PostMsg` class and may set the VPI onto the response message received from the post, when reflecting the posted content to the downstream device.

RFA does not provide the interface to the client application that allows setting of the VPI on the `Reuters.RFA.Message.PostMsg` class. RFA will internally populate the VPI on the post message submitted by an OMM Consumer application and send the post out to the network.

In addition to accessing VPI via the RFA-provided `PostMsg` and `RespMsg`, OMM applications can also obtain it via Field Identifiers defined from the publisher component.

NOTE: The VPI can also be obtained via FIDs defined from the publisher component. For details, refer to the publishing component's documentation.

7.1.14 Response Status (Stream States and Data States)

The Response Status determines item status through two values: the stream state and the data state. The stream state indicates the state of the stream (e.g., **Open**, **Closed**) while the data state indicates the validity of the data (e.g., **OK**, **Suspect**). The Response Status is applicable to both the Consumer, which receives the Response Status, and the Provider, which sends it. The Session Layer may also inform applications of Response Status for reasons such as a change in data health or an item being closed.

The following diagram depicts the various item status states specifying stream state and data state. (For simplicity other factors that affect item state, such as the Refresh Complete flag, have been omitted.) The diagram depicts the transitions between item status state, such as when the application specifies interest and when it receives an OMM Item Event.

Each state may contain multiple values for any particular state (e.g., **OK | Suspect**). This syntax implies the state may be either **OK** or **Suspect**.

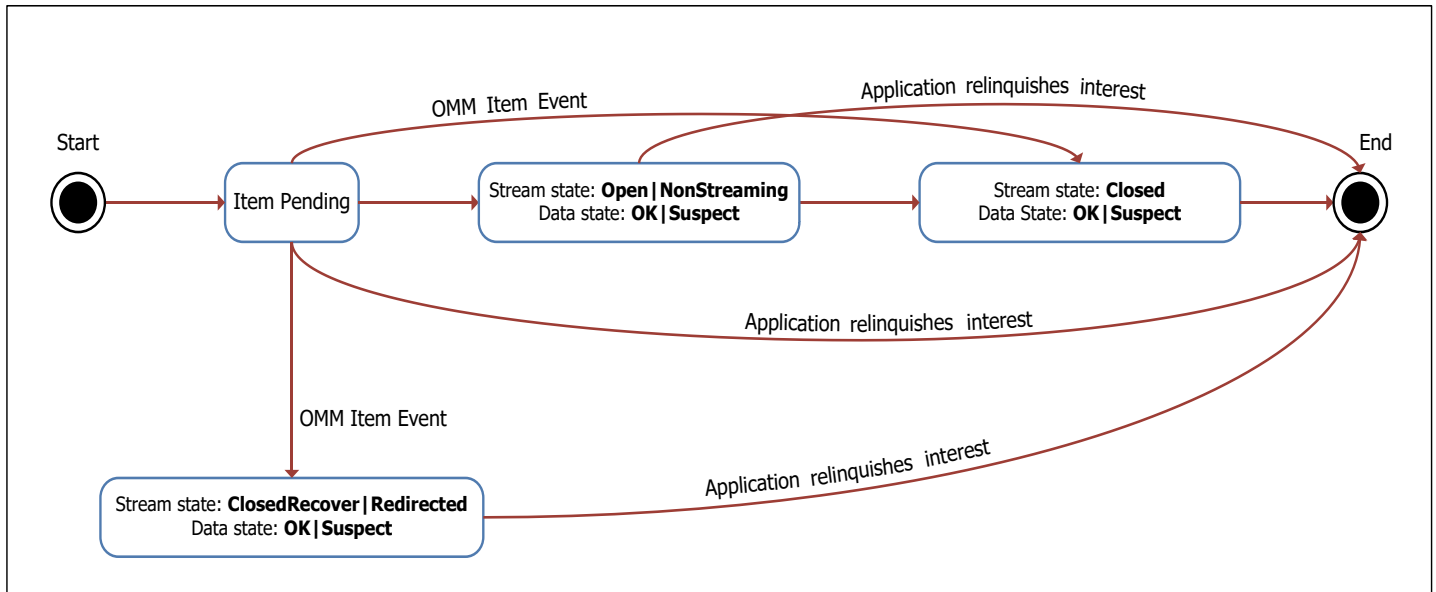


Figure 60: OMM Item State Diagram

The following table describes the combinations of stream and data states from a Consumer application point of view.

STREAMSTATE ²³	DATASTATE ²⁴	DESCRIPTION
NonStreaming	OK Suspect	After receiving a final refresh message or status message, the stream is closed and no updates are delivered.
Open	OK	All data associated with the stream is healthy and current. Also received after the connection to the provider has been accepted.
Open	Suspect	The health of some or all of the data on a stream is out of date or cannot be confirmed as current (e.g., the service is down). This is also received immediately after initiating a login and is sent by RFA, not the upstream provider. If an application does not allow suspect data, a stream might change from Open to Closed or ClosedRecover as a result.

²³ A StreamState of **Unspecified** is typically an initialization value and is not intended to be encoded or decoded. Attempts to encode it will result in a StreamState of **open**. It may still be received, however, when connecting to older components. If it is received, it should be interpreted as **open**.

²⁴ A StreamState of **Unspecified** is typically an initialization value and is not intended to be encoded or decoded. Attempts to encode it will result in a StreamState of **open**. It may still be received, however, when connecting to older components. If it is received, it should be interpreted as **open**.

STREAMSTATE ²³	DATASTATE ²⁴	DESCRIPTION
Closed	OK Suspect	Data is not available on this service and connection and is not likely to become available.
ClosedRecover	OK Suspect	The current stream is closed; however, data can be recovered on this service and connection at a later point in time. ²⁵
Redirected	OK Suspect	The current stream is closed. The application can issue a new request for the data using the new message attributes from the redirect message.

Table 61: Stream and Data State Descriptions: Consumer

The following table describes the combinations of stream and data states from a Provider application point of view.

STREAMSTATE	DATASTATE	DESCRIPTION
NonStreaming	OK Suspect	After sending the final refresh message or status message, the stream is closed.
Open	OK	The stream data is healthy and current. Also sent after a consumer connection has been accepted.
Open	Suspect	The health of some or all of the data on a stream is out of date may not be current (e.g., the service is down).
Closed	OK Suspect	Data is not available on this service and connection and is not likely to become available.
ClosedRecover	OK Suspect	The current stream is closed; however, a consumer application may attempt to recover this service and connection at a later point in time. ²⁶
Redirected	OK Suspect	The current stream is closed. The application can issue a new request for the data using the new message attributes from the redirect message.

Table 62: Stream and Data State Descriptions: Provider

STATUS CODE DEFINITION	DESCRIPTION
AlreadyOpen	Indicates that the stream item is already open.
DacsDown	Indicates that the connection to DACS is down. Users are not allowed to connect.
DacsMaxLoginsReached	Indicates that the maximum number of DACS logins was reached.
DacsUserAccessToAppDenied	Indicates that the user is not allowed to use the application.
Error	Indicates that an internal error was received from the sender. Specific information should be available in the text.
ExceededMaxMountsPerUser	Indicates that the maximum number of mounts per user was exceeded.
FailoverCompleted	Indicates that the recovery from a failover condition is finished.
FailoverStarted	Indicates that a component is recovering from a failover condition.
FullViewProvided	Indicates that the full view (e.g., all available fields) is provided, regardless of whether a specific view was requested.
GapDetected	Indicates a gap was detected between messages. A gap might be detected via an external reliability mechanism, or when using the seqNum element on a message.

²⁵ Single Open behavior can modify this state. A consumer connecting to a provider that supports [Singleopen](#) will never receive a response message with a stream state of [ClosedRecover](#).

²⁶ Single Open behavior can modify this state. A provider that supports [Singleopen](#) handles recovery for the client and never sends a response message with a stream state of [ClosedRecover](#). If the provider does attempt to send a response of [ClosedRecover](#), RFA will return an error.

STATUS CODE DEFINITION	DESCRIPTION
GapFill	Reserved for future use.
InvalidArgument	Indicates that a parameter on the request is invalid or unrecognized. Specific information should be contained in the text element.
InvalidView	Indicates that the requested view is invalid, possibly due to bad formatting. Additional information should be available in the text.
JustInTimeFilteringStarted	Indicates that Just-In-Time conflation started on the stream.
NoBatchViewSupportInReq	Indicates that the request does not support batch or view.
NoResources	Indicates that resources are not available to accommodate the stream.
None	No additional state code information is required or present.
NonUpdatingItem	Indicates that a streaming request was made for non-updating data.
NotAuthorized	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 QoS.
NotFound	Indicates that requested information was not found, though it might become available later or by changing some of the request parameters.
NotOpen	Indicates that the stream was not open. Additional information should be available in the text.
Preempted	Indicates that the stream was preempted, possibly by a caching device. Typically indicates the user exceeded an item limit, whether specific to the user or to a component in the system. Further information should be available in the text.
SourceUnknown	Indicates that the requested service is not known, though the service might become available later.
TickByTickResumed	Indicates that Just-In-Time conflation on the stream is finished.
Timeout	Indicates that a timeout occurred somewhere in the system while processing requested data.
TooManyItems	Indicates that a request cannot be processed because too many streams are already open.
UnableToRequestAsBatch	Indicates that a batch request cannot be used for this request. The user can instead split the batched items into individual requests.
UnsupportedViewType	Indicates that the domain on which a request is made does not support the requested viewType.
UsageError	Indicates invalid usage of code within the system. Specific information should be available in the text.
UserUnknownToPermSys	Indicates that the user is unknown to the permissioning system, which could be DACS, AAA, or Refinitiv Real-Time Edge Device.

Table 63: Status Code Definitions

7.2 Message Package Usage

RFA uses OMM to communicate between the sender and receiver so that the communication message will not depend on RFA interfaces. This chapter describes about the encoding and decoding processes on a data and message package.

The picture below illustrates a general message flow between sender and receiver.

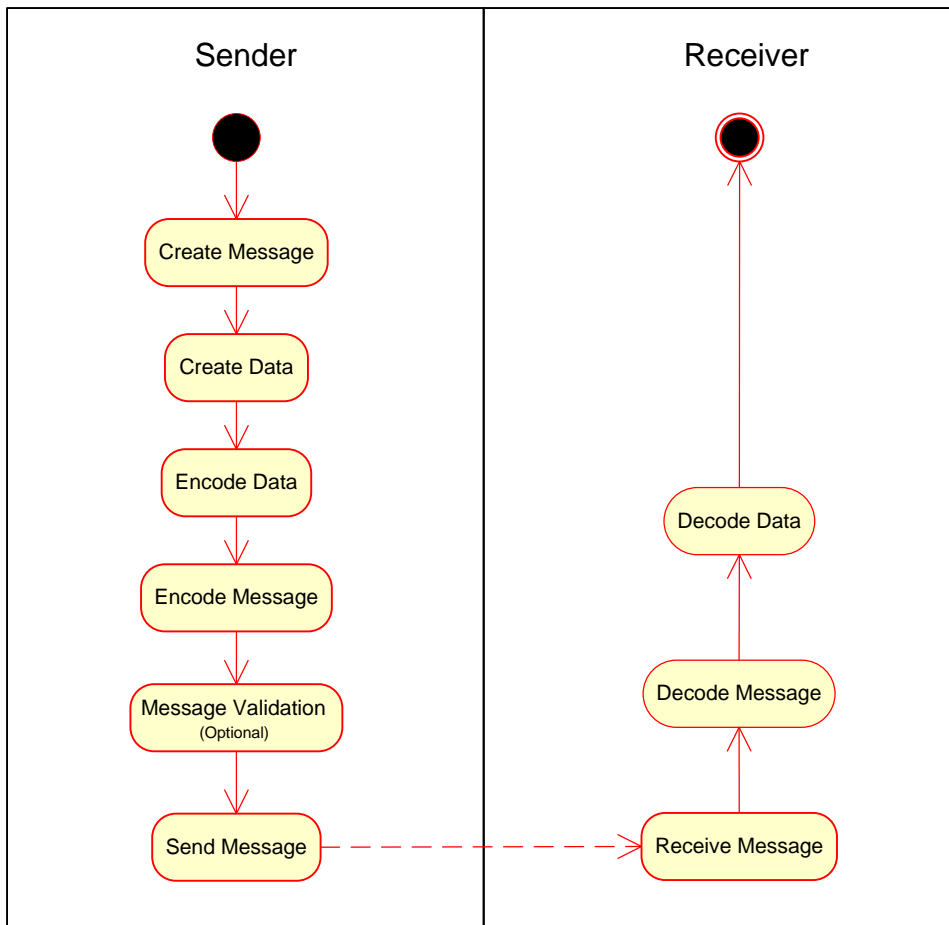


Figure 61: Message Flow between Sender and Receiver

7.2.1 Message Encoding

This section describes how to encode a message and its payload data for the sender side. Below is the encoding process.

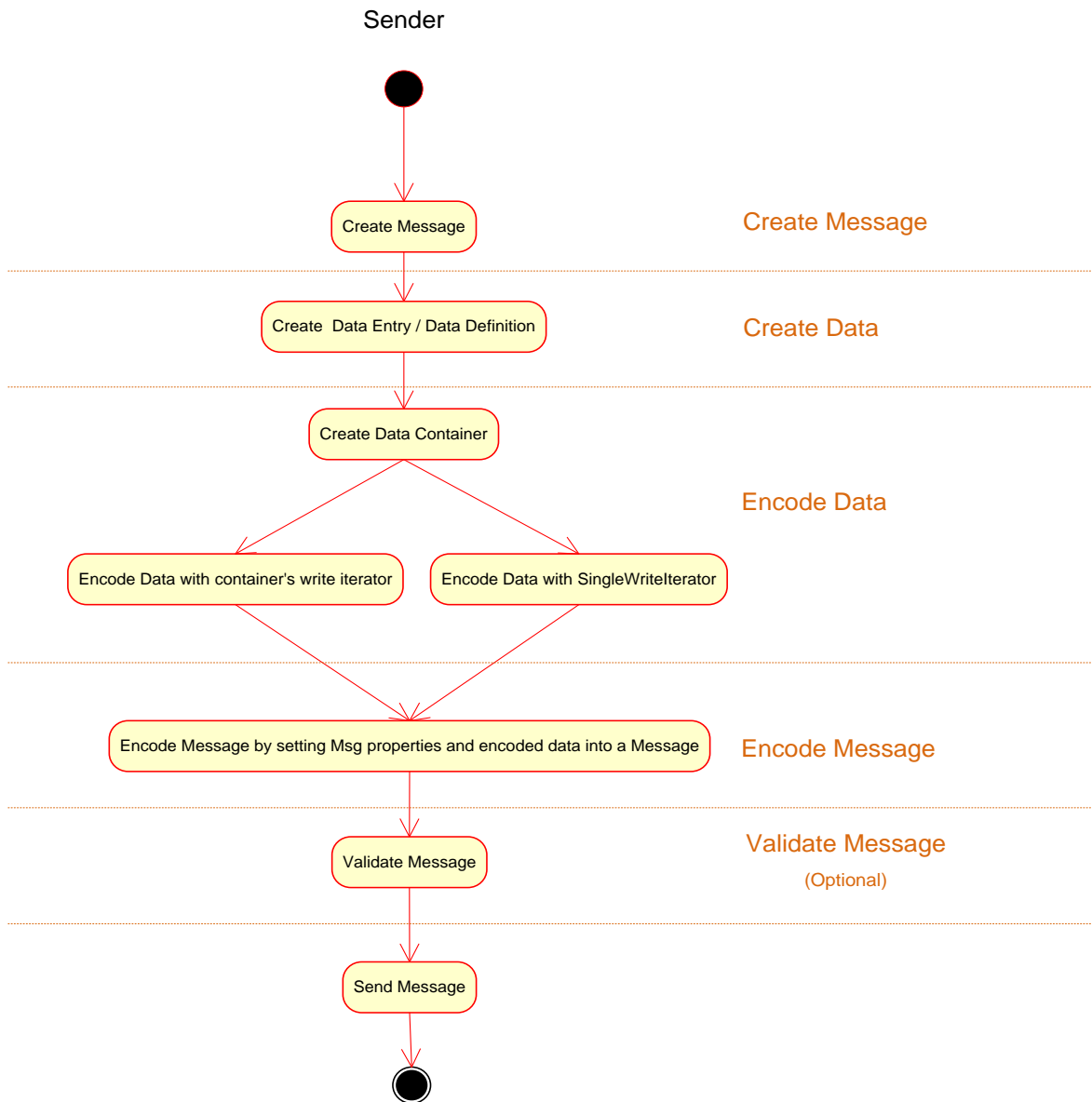


Figure 62: Encoding process

For encoding a message, the sender has to create an appropriate message type and payload data for this message. The payload data can be categorized into two types namely data container or data leaf. The data container can be encoded from data entries by using either specific container-specific write iterator or SingleWriteliterator. Moreover, the data leaf can be encoded by setting appropriate values. Finally, the message is encoded by setting the payload and message properties.

For more information about encoding data, refer to section 6.2.1.

7.2.1.1 Encoding Request Message

7.2.1.1.1 Encoding Request Message

The request message can be encoded through the following common procedures:

- Set the `MsgModelType` of the request message.
- Set the `InteractionType` of the request message. For streaming request, the `InteractionType` must be set with `InteractionTypeFlag.InitialImage` and `InteractionTypeFlag.InterestAfterRefresh`.
- Set `NameType` and `Name` of the `AttribInfo`. Both properties indicate that `AttribInfo` represents request information with the user name.
- Encode the application Id and the position into the `AttribInfo`.
- Set the `AttribInfo` to the request message.

The following example shows how to encode the login request message. For more example, refer to the StarterConsumer example.

```
ReqMsg reqMsg = new ReqMsg();
reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
    ReqMsg.InteractionTypeFlag.InterestAfterRefresh;
AttribInfo attribInfo = new AttribInfo();
attribInfo.NameType = RDM.Login.USER_ID_TYPES.USER_NAME;
attribInfo.Name = cfgVariables.UserName;
ElementList elementList = new ElementList();
ElementEntry element = new ElementEntry();
DataBuffer elementData = new DataBuffer();
ElementListWriteIterator elwiter = new ElementListWriteIterator();

elwiter.Start(elementList);
element.Name = RDM.Login.ENAME_APP_ID;
elementData.SetFromString(cfgVariables.AppId, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);

element.Name = RDM.Login.ENAME_POSITION;
elementData.SetFromString(cfgVariables.Position, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);

elwiter.Complete();
attribInfo.Attrib = elementList;
reqMsg.AttribInfo = attribInfo;
```

Example 87: Encoding Request Message

7.2.1.1.2 Encoding Request Message for View

Users can define a View by setting the **View** to [IndicationMaskFlag](#) and encoding the **ViewType** and **ViewData** element entries in an element list contained in the payload of Request Message. The **View** indicates whether there is a View requirement with the request message and a View definition in its payload.

To change the view, the same process can be followed when performing a reissue. If a reissue is being done for a different purpose (e.g., pausing a stream, priority change) and the user wants to keep the current view, the [IndicationMaskFlag.View](#) flag should be set with no ViewData and ViewType. To remove a view, the user can perform a reissue with no [IndicationMaskFlag.View](#).

The request message for View can be encoded through the following common procedures:

- Set the [MsgModelType](#) of the request message.
- Set the [InteractionType](#) of the request message. For streaming request, the [InteractionType](#) must be set with [InteractionTypeFlag.InitialImage](#) and [InteractionTypeFlag.InterestAfterRefresh](#).
- Set [NameType](#), [Name](#) and [ServiceName](#) of the [AttribInfo](#). [NameType](#) and [Name](#) properties indicate that [AttribInfo](#) represent request information with the instrument name ric.
- Set the [AttribInfo](#) to the request message.
- Set the [IndicationMaskFlag.View](#) to indicationMask of the request message.
- Encode ViewType and ViewData element entries into the payload.
- Set payload data to the request message for view. The data type of the payload data depends on the Message Model type. For more information, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must encode and set to the request message by using the [payload](#) property. For more information about the encoding step, refer to section 6.2.1, Data Encoding.

The following example shows how to encode the item request message for View. For more example, refer to the `StarterConsumer_BatchView` example.

```
ReqMsg reqMsg = new ReqMsg();
AttribInfo attribInfo = new AttribInfo();

reqMsg.MsgModelType = msgModelType;
reqMsg.InteractionType = interType;

attribInfo.NameType = RDM.RDM.INSTRUMENT_NAME_TYPES.INSTRUMENT_NAME_RIC;
attribInfo.Name = itemName;
attribInfo.ServiceName = serviceName;

ElementList attribElementList = new ElementList();
if (cfgvariables.SendAttribInfo)
{
    //encode the List name into the AttribInfo
    ElementEntry element = new ElementEntry();
    DataBuffer elementData = new DataBuffer();
    ElementListWriteIterator elwiter = new ElementListWriteIterator();
    RFA_String listName = new RFA_String();

    elwiter.Start(attribElementList);
    listName.Set("Request List");
    element.Name = listName;
    RFA_String Name = new RFA_String("Item Name");
    elementData.SetFromString(Name, DataBuffer.DataBufferEnum.StringAscii);

    element.Data = elementData;
```

```

    elwiter.Bind(element);
    elwiter.Complete();
    attribInfo.Attrib = attribElementList;
}
reqMsg.AttribInfo = attribInfo;

//This section sets a view for a single item
//A view is defined as specification of a subset of Field(Entry)s or Element(Entry)s of a particular
    open item.
//The response message contains only those fields or entries defined by the View in item requests.
//A View is defined by the user by setting the ViewType and corresponding ViewData element
//entries in an element list in the payload of Request Message
int viewFieldCount = 0;
if (cfgVariables.ItemNameViewFIDs != null)
{
    viewFieldCount = cfgVariables.ItemNameViewFIDs.Count;
}

ElementList elementList = new ElementList();
if (viewFieldCount > 0)
{
    reqMsg.IndicationMask = (byte)(reqMsg.IndicationMask | ReqMsg.IndicationMaskFlag.View);

    ElementEntry element = new ElementEntry();
    ElementListWriteIterator elwiter = new ElementListWriteIterator();
    DataBuffer dataBuffer = new DataBuffer();
    elwiter.Start(elementList);

    //encode a viewType
    element.Name = RDM.RDM.REQUEST_MSG_PAYLOAD_ELEMENT_NAME.ENAME_VIEW_TYPE;
    dataBuffer.SetUInt(RDM.RDM.VIEW_TYPES.VT_FIELD_ID_LIST, DataBuffer.DataBufferEnum.UInt);
    element.Data = dataBuffer;
    elwiter.Bind(element);

    ArrayWriteIterator arrWit = new ArrayWriteIterator();
    Reuters.RFA.Data.Array elementData = new Reuters.RFA.Data.Array();
    ArrayEntry arrayEntry = new ArrayEntry();
    element.Clear();
    arrWit.Start(elementData);

    //encode the ViewData which is comprised of field IDs specified by the user in the
        viewFieldIdList parameter
    for (int i = 0; i < viewFieldCount; i++)
    {
        dataBuffer.Clear();
        arrayEntry.Clear();
        dataBuffer.SetFromString(((cfgVariables.ItemNameViewFIDs)[i]), DataBuffer.DataBufferEnum.Int);
        arrayEntry.Data = dataBuffer;
        arrWit.Bind(arrayEntry);
    }
    arrWit.Complete();

    element.Name = RDM.RDM.REQUEST_MSG_PAYLOAD_ELEMENT_NAME.ENAME_VIEW_DATA;
    element.Data = elementData;
    elwiter.Bind(element);
    elwiter.Complete();

    reqMsg.Payload = elementList;
}

```

Example 88: Encoding Request Message for View

7.2.1.1.3 Encoding Request Message for Batching Items

The consumer application can specify interest in a list of items by using a single batch Request Message. To do this, the user sets the **Batch** request flag and encodes an **ItemList** element entry in an element list contained in the payload of the request message. The **Batch** request flag indicates that the request message contains a batch of items and that the payload includes an **ItemList**.

The request message for Batch can be encoded through the following common procedures:

- Set the `MsgModelType` of the request message.
- Set the `InteractionType` of the request message. For streaming request, the `InteractionType` must be set with `InteractionTypeFlag.InitialImage` and `InteractionTypeFlag.InterestAfterRefresh`.
- Set `ServiceName` of the `AttribInfo`.
- Encode list name into the `AttribInfo`.
- Set the `AttribInfo` to the request message.
- Set the `IndicationMask` of the request message to `IndicationMaskFlag.Batch`.
- Encode **ItemList** into the payload.
- Set payload data to the request message for Batch. The data type of the payload data depends on the Message Model type. For more information, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must encode and set to the batch request message by using the `Payload` property. For more information about the encoding step, refer to Section 6.2.1.

The following example shows how to encode the request message for Batch. For more example, refer to the `StarterConsumer_BatchView` example.

```
ReqMsg reqMsg = new ReqMsg();
reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
    ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

AttribInfo attribInfo = new AttribInfo();
attribInfo.ServiceName = cfgVariables.ServiceName;

ElementList attribElementList = new ElementList();
if (cfgVariables.SendAttribInfo)
{
    //encode the List name into the AttribInfo
    ElementEntry element1 = new ElementEntry();
    DataBuffer elementData1 = new DataBuffer();
    ElementListWriteIterator elwiter1 = new ElementListWriteIterator();
    elwiter1.Start(attribElementList);

    RFA_String listName = new RFA_String();
    listName.Set("Request List");
    element1.Name = listName;

    RFA_String stringListValue = new RFA_String(listValue);
    elementData1.SetFromString(stringListValue, DataBuffer.DataBufferEnum.StringAscii);
    element1.Data = elementData1;
    elwiter1.Bind(element1);
    elwiter1.Complete();
    attribInfo.Attrib = attribElementList;
}
```

```

}
reqMsg.AttribInfo = attribInfo;

//set Batch Request Flag
reqMsg.IndicationMask = ReqMsg.IndicationMaskFlag.Batch;

//set itemList in payload.
ElementList elementList = new ElementList();
ElementEntry element = new ElementEntry();
ElementListWriteIterator elwiter = new ElementListWriteIterator();

elwiter.Start(elementList);

ArrayWriteIterator arrWit = new ArrayWriteIterator();
Reuters.RFA.Data.Array elementData = new Reuters.RFA.Data.Array();
//encode a ItemList for batch
arrWit.Start(elementData);

DataBuffer dataBuffer = new DataBuffer();
ArrayEntry arrayEntry = new ArrayEntry();
for (int i = 0; i < count; i++)
{
    dataBuffer.Clear();
    arrayEntry.Clear();
    dataBuffer.SetFromString(strList[i], DataBuffer.DataBufferEnum.StringAscii);
    arrayEntry.Data = dataBuffer;
    arrWit.Bind(arrayEntry);
}
arrWit.Complete();

element.Name = RDM.RDM.REQUEST_MSG_PAYLOAD_ELEMENT_NAME.ENAME_BATCH_ITEM_LIST;
element.Data = elementData;
elwiter.Bind(element);

//This section sets a view for the batch (if a view was specified).
//A view is defined as specification of a subset of Field(Entry)s or Element(Entry)s of a particular open
    item.
//The response message contains only those fields or entries defined by the view in item requests.
//A view is defined by the user by setting the ViewType and corresponding ViewData element
//entries in an element list in the payload of Request Message.
int viewFieldCount = 0;
if (viewFieldIdList != null)
{
    viewFieldCount = viewFieldIdList.Count;
}

if (viewFieldCount > 0)
{
    //set view Request Flag
    reqMsg.IndicationMask = (byte)(reqMsg.IndicationMask | ReqMsg.IndicationMaskFlag.View);
    element.Clear();
    element.Name = RDM.RDM.REQUEST_MSG_PAYLOAD_ELEMENT_NAME.ENAME_VIEW_TYPE;
    dataBuffer.Clear();
    dataBuffer.SetUInt(RDM.RDM.VIEW_TYPES.VT_FIELD_ID_LIST, DataBuffer.DataBufferEnum.UInt);
    element.Data = dataBuffer;
    elwiter.Bind(element);

    //encode the ViewData
    element.Clear();
    elementData.Clear();

```

```

arrWit.Start(elementData);

for (int i = 0; i < viewFieldCount; i++)
{
    dataBuffer.Clear();
    arrayEntry.Clear();
    dataBuffer.SetFromString(viewFieldIdList[i], DataBuffer.DataBufferEnum.Int);
    arrayEntry.Data = dataBuffer;
    arrWit.Bind(arrayEntry);
}

arrWit.Complete();

element.Name = RDM.RDM.REQUEST\_MSG\_PAYLOAD\_ELEMENT\_NAME.ENAME_VIEW_DATA;
element.Data = elementData;
elwiter.Bind(element);
}

elwiter.Complete();
reqMsg.Payload = elementList;

```

Example 89: Encoding Request Message for Batch

7.2.1.1.4 Encoding Request Message for SymbolList to Request Data for Individual Items

The OMMConsumer can request individual items as well as names from the SymbolList domain. To do this, the user encodes the [DataStreams](#) element entry in an element list and set it to the [SymbolListBehaviors](#) element entry in the element list contained in the payload of the request message.

The request message for SymbolList can be encoded through the following common procedures:

- Set the [MsgModelType](#) of the request message.
- Set the [InteractionType](#) of the request message. For streaming request, the [InteractionType](#) must be set with [InteractionTypeFlag.InitialImage](#) and [InteractionTypeFlag.InterestAfterRefresh](#).
- Set [ServiceName](#) of the [AttribInfo](#).
- Set [Name](#) of the [AttribInfo](#).
- Set the [AttribInfo](#) to the request message.
- Encode [DataStreams](#) into an element list. Set the element list to the [SymbolListBehaviors](#) element entry.
- Encode [SymbolListBehaviors](#) into the payload.

Set payload data to the request message. The data type of the payload data depends on the Message Model type. For more details, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must be encoded and set to the SymbolList request message by using the [Payload](#) property. For more details about the encoding steps, refer to Section 6.2.1.

The following example shows how to encode the request message for SymbolList. For more example, refer to the StarterConsumer_SymbolList example.

```
ReqMsg reqMsg = new ReqMsg();
AttribInfo attribInfo = new AttribInfo();

reqMsg.MsgModelType = RFA.RDM.RDM.MESSAGE_MODEL_TYPES.MMT_SYMBOL_LIST;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

attribInfo.NameType = RDM.RDM.INSTRUMENT_NAME_TYPES.INSTRUMENT_NAME_RIC;
attribInfo.Name = itemName;
attribInfo.ServiceName = serviceName;

reqMsg.AttribInfo = attribInfo;
// Note : Encode Symbol List Requestable behaviours.
// Data for the individual items of the symbol list can be requested as snapshot or streaming.
ElementList enhSymListReq = new ElementList();
ElementEntry element = new ElementEntry();
ElementListWriteIterator elwiter = new ElementListWriterIterator();
DataBuffer dataBuffer = new DataBuffer();
ElementList reqBehaviors = new ElementList();
ElementListWriteIterator elReqwiter = new ElementListWriteIterator();
ElementEntry elReqBehavior = new ElementEntry();
If (cfgVariables.isDataStreamsConfigured)
{ // Encode request payload with DataStreams:
    elReqwiter.Start(reqBehaviors);
    elReqBehavior.Name = RDM.RDM.REQUEST_MSG_PAYLOAD_ELEMENT_NAME.ENAME_DATA_STREAMS;
    dataBuffer.UInt = cfgVariables.dataStreams; // Requesting data snapshot or streaming
    elReqBehavior.Data = dataBuffer;
    elReqwiter.Bind(elReqBehavior);
    elReqwiter.Complete();

    // Encode request symbol list behaviors:
    dataBuffer.Clear();
    elwiter.Start(enhSymListReq);
    element.Name = RDM.RDM.REQUEST_MSG_PAYLOAD_ELEMENT_NAME.ENAME_SYMBOL_LIST_BEHAVIORS;
    element.Data = reqBehaviors;
    elwiter.Bind(element);
    elwiter.Complete();

    reqMsg.Payload = enhSymListReq;
}
```

Example 90: Encoding Request Message with DataStreams for SymbolList Enhancement

7.2.1.2 Encoding Response Message

7.2.1.2.1 Encoding Response Message

The provider application creates a Response Message and populates it with the required information. The response message is encapsulated in the OMM Solicited Item Command ([OMMSolicitedItemCmd](#)).

The response message can be encoded through the following common procedures:

- Set the [MsgModelType](#) of the response message. The Message Model is the same as the Message Model type of the request message received by the provider application.
- Set response type of the response message. The response type can be [Refresh](#), [Update](#) or [Status](#). The Response Status describes an item's interaction state ([Open](#), [Closed](#) ...) and data quality ([Ok](#), [Suspect](#) ...). Response Status is available only on Refresh message and Status message.
- Set the [IndicationMask](#). The [IndicationMask](#) describes the management of information. The [IndicationMaskFlag.DoNotCache](#) indicates that the response message should not be cached. The [IndicationMaskFlag.DoNotFilter](#) indicates that the Response Message should not be conflated. If the created Response Message is the last fragment of refresh, the [IndicationMaskFlag.RefreshComplete](#) has to be set.
- Set the [RespTypeEnum](#). The [RespTypeEnum](#) describes the type specifier.
- Set [AttribInfo](#) to the response message.
- Set [Manifest](#) to the response message (optional). The [Manifest](#) interface allows the application to set meta-data including the Sequence Number created by the provider; Filter Info on updates, Item Group Identifier and Permission Data.
- Set [QualityofService](#) to the response message. The [QualityofService](#) attribute describes the quality of service supported by the service.
- Set payload data to the response message. The data type of the payload data depends on the Message Model type. For more information, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must encode and set to the response message by using the [Payload](#) property. For more information about the encoding step, refer to Section 6.2.1.

The following example shows how to encode the response message. For more example, refer to the StarterProvider_Interactive example.

```
void EncodeMarketPriceMsg(RespMsg respMsg, RespMsg.RespTypeEnum respType, AttribInfo attribInfo, RespStatus
respstatus, QualityOfService qos, bool isSetAttribute, bool isSolicited, byte msgModelType)
{
    // set MsgModelType
    respMsg.MsgModelType = msgModelType;

    // Set response Type
    respMsg.RespType = respType;

    if (respType == RespMsg.RespTypeEnum.Refresh)
    {
        // Set respStatus
        respMsg.RespStatus = respStatus;
        // set IndicationMask
        respMsg.IndicationMask = (byte)(respMsg.IndicationMask |
            RespMsg.IndicationMaskFlag.RefreshComplete);
        // TODO: This probably should be configurable for other indicationMask settings

        // set RespTypeEnum
        if (issolicited)
        {
            respMsg.RespTypeEnum = RDM.RDM.INSTRUMENT_REFRESH_RESPONSE_TYPES.REFRESH_SOLICITED;
        }
    }
}
```



```

        //for solicited refresh, value is 0
    }
    else
    {
        respMsg.RespTypeEnum = RDM.RDM.INSTRUMENT_REFRESH_RESPONSE_TYPES.REFRESH_UNSOLICITED;
        //for unsolicited refresh, value is 1
    }
}
else if (respType == RespMsg.RespTypeEnum.Update)
{
    // set indication mask for update
    respMsg.IndicationMask = RDM.RDM.INSTRUMENT_UPDATE_RESPONSE_TYPES.INSTRUMENT_UPDATE_UNSPECIFIED;
    // set RespTypeEnum
    respMsg.RespTypeEnum = RDM.RDM.INSTRUMENT_UPDATE_RESPONSE_TYPES.INSTRUMENT_UPDATE_UNSPECIFIED;
    //for unspecified update response, value is 0
}
else
{
    // this is a status message
    // Set respStatus
    respMsg.RespStatus = respStatus;
    // set IndicationMask
    respMsg.IndicationMask = respMsg.IndicationMask;
    // TODO: This probably should be configureable for other indicationMask settings
}

// Set attribute
if (isSetAttribute)
{
    respMsg.AttribInfo = attribInfo;
}

// Set QoS
if (QoS != null)
{
    respMsg.QualityOfService = QoS;
}
}

```

Example 91: Encoding Response Message

7.2.1.2.2 Encoding Response Message for View

The provider application must decide how to respond to a Request Message that contains a view specification. First, the provider needs to decode 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, ideally it sends only the fields or elements specified in the **ViewData** in any 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 or fields in addition to those specified in the **ViewData**.

The response message for view can be encoded through the common procedures in Section 7.2.1.2. For more example, refer to the StarterProvider_BatchView example.

7.2.1.3 Encoding Post Message

The consumer application creates a Post Message and populates it with the necessary message and payload information. For Publisher Principal Identity (i.e. VPI), RFA internally sets to the Post Message before routing it to the network. The post message can be encoded through the following common procedures:

- Set the [MsgModelType](#) of the post message.

- Set the `IndicationMask`. The `IndicationMask` describes the management of information. The `IndicationMaskFlag.MessageInit` indicates that this Post Message is the first part of multi-part Post Message. The `IndicationMaskFlag.MessageComplete` indicates that this Post Message is the last part of multi-part Post Message. If the `IndicationMaskFlag.MessageInit` and `IndicationMaskFlag.MessageComplete` are set at the same time, this indicates that this is a single-part Post Message.
- Set `AttribInfo` to the post message.
- Set `SeqNum` to the post message. (Optional)
- Set `PostId` to the post message (optional). `PostId` uniquely identifies atomic Post Message. If the `PostId` is set, `wantAck` in the Indication Mask Flag must be set. This `wantAck` indicates that application request acknowledgement to this Post Message.
- Set `PermissionData` to the post message (optional).
- Set `PostUserRightsMask` to the post message (optional). The `PostUserRightsMask` indicates the Post User Rights values. The `PostUserRightsMaskFlag.Create` indicates that the user is allowed to create records in cache with this post. The `PostUserRightsMaskFlag.Delete` indicates that the user is allowed to delete/remove records from cache with this post. The `PostUserRightsMaskFlag.ModifyPermissionData` indicates that the user is allowed to modify the `PermissionData` for records already in cache with this post.
- Set payload data to the post message. The data type of the payload data depends on the Message Model type. For more information, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must be encoded and set to the post message by using the `Payload` property. For more information about the encoding step, refer to Section 6.2.1.

NOTE: Sending a Post Message is supported only on items whose stream state is `Open` and data state is `OK`.

The following example shows how to encode the post message. For more example, refer to the `StarterConsumer_Post` example.

```
void SubmitPostMsgWithData(long handle, AttribInfo attribInfo, int postID, int seqNum, byte domain)
{
    PostMsg postMsg = new PostMsg();

    // this application encodes a single part post msg only on any specified domain
    postMsg.MsgModelType = domain;
    postMsg.IndicationMask = PostMsg.IndicationMaskFlag.MessageInit |
        PostMsg.IndicationMaskFlag.MessageComplete;

    // since attrib info is optional...
    if (attribInfo != null)
        postMsg.AttribInfo = attribInfo;

    // since sequence number is optional, if value passed in negative
    // it is not set on post msg
    if (seqNum >= 0)
        postMsg.SeqNum = (uint)seqNum;

    // since post id is optional, if value passed in negative
    // it is not set on post msg; this also means that ack is not requested
    if (postID >= 0)
    {
        postMsg.IndicationMask = (byte)(postMsg.IndicationMask | PostMsg.IndicationMaskFlag.WantAck);
        postMsg.PostID = (uint)postID;
    }

    // Permission data may be included in the PostMsg for the user to be permissioned to post
    // information, using the PostMsg.PermissionData property.
}
```

```

int serviceId = 1;
int peOperator = AuthorizationLock.OperatorEnum.OR;
List<uint> peList = new List<uint>();
peList.Add(6562);
Reuters.RFA.Common.Buffer buffer = new Reuters.RFA.Common.Buffer();

AuthorizationLock authLock = new AuthorizationLock(serviceID, peOperator, peList);
AuthorizationLockData lockData = new AuthorizationLockData();
AuthorizationLockStatus retStatus = new AuthorizationLockStatus();
AuthorizationLockData.LockResultEnum result = authLock.GetLock(lockData, retStatus);

if (result != AuthorizationLockData.LockResultEnum.LOCK_SUCCESS)
{
    authLock.Dispose();
}

byte[] c_LockData = lockData.LockData;
int c_LockDataSize = lockData.Size;
buffer.SetFrom(c_LockData, c_LockDataSize, c_LockDataSize);
postMsg.PermissionData = buffer;

// Set the optional Post User Rights: sets the Post User Rights Mask to create records in a
// cache with this post or modifies permission data for records already in a cache with this post.
postMsg.PostUserRightsMask = (PostMsg.PostUserRightsMaskFlag.Create |
    PostMsg.PostUserRightsMaskFlag.ModifyPermissionData);

// encode data payload to FilterList
FieldList fl = new FieldList();
// ...

// set payload to Post Message.
postMsg.Payload = fl;
}

```

Example 92: Encoding Post Message

7.2.1.4 Encoding Ack Message

The provider application creates the acknowledgement message and populates it with the necessary information. The acknowledgement message can be encoded through the following common procedures:

- Set [AckId](#) to the ack message.
- Set [MsgModelType](#) of the ack message.
- Set [SeqNum](#), [NackCode](#) and [Text](#). These properties are optional.
- Call [SetAssociatedMetaInfo\(\)](#) method to set releated meta info to the ack message with handle.
- Set payload data to the ack message (optional). The data type of the payload data depends on the Message Model type. For more information, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must be encoded and be set into the acknowledgement message using the [Payload](#) property. For more information about the encoding step, refer to Section 6.2.1.

The following example shows how to encode the post message. For more example, refer to the StarterProvider_Post example.

```
void SubmitAckMsg(RequestToken requestToken, long handle, bool postIdSet, UInt32 ackID, bool sequenceNumSet,
UInt32 sequenceNum, bool nackCodeSet, byte nackCode, RFA_String text, byte msgModelType)
{
    AckMsg ackMsg = new AckMsg();

    ackMsg.AckID = ackID;

    // client app is required to set msg Model type of the posted message.
    ackMsg.MsgModelType = msgModelType;

    if (sequenceNumSet)
    {
        ackMsg.SeqNum = sequenceNum;
    }

    if (nackCodeSet)
    {
        ackMsg.NackCode = nackCode;
    }
    if (!text.Empty())
    {
        ackMsg.Text = text;
    }

    ackMsg.SetAssociatedMetaInfo(handle);
    // ...
}
```

Example 93: Creating the Ack Message

7.2.1.5 Encoding Generic Message

A Generic Message is a bi-directional message sent from a Provider to a Consumer or from a Consumer to a Provider. The application creates the generic message and populates it with the necessary information. The generic message can be encoded through the following common procedures:

- Set the `MsgModelType` of the generic message.
- Set `NameType` and `Name` of the `AttribInfo`. Both properties indicate that `AttribInfo` represent request information with the user name.
- Set `AttribInfo` to the generic message.
- Set the `IndicationMask`. The `IndicationMask` describes the management of information. The `MessageComplete` indicated that this Generic Message is the final portion of the message.
- Set payload data to the generic message. The data type of the payload data depends on the Message Model type. For more information about the data type in the payload data, refer to the *RFA RDM Usage Guide .NET Edition*. Data in the payload data must encode and set to the generic message by using `Payload` property. For more information about the encoding step, refer to Section 6.2.1.

The following example shows how to encode the generic message. For more example, refer to the `StarterConsumer_GenericMsg` or `StarterProvider_GenericMsg` example.

```
GenericMsg genericMsg = new GenericMsg();
genericMsg.MsgModelType = cfgVariables.GenericMsgModelType;

AttribInfo attribInfo = new AttribInfo();
attribInfo.NameType = RDM.RDM.INSTRUMENT_NAME_TYPES.INSTRUMENT_NAME_RIC;

attribInfo.Name = new RFA_String("SPEAK");

genericMsg.AttribInfo = attribInfo;

// Encode payload

genericMsg.Payload = payload;
genericMsg.IndicationMask = GenericMsg.IndicationMaskFlag.MessageComplete;
```

Example 94: Creating Generic Message for consumer

7.2.1.6 Message Validation

There are five types of messages: [ReqMsg](#), [RespMsg](#), [GenericMsg](#), [PostMsg](#), and [AckMsg](#). RFA provides an interface to verify that constructed messages of these types conform to the Refinitiv Domain Models as specified in the *RFA RDM Usage Guide .NET Edition*.

The [validateMsg\(\)](#) interface accepts two parameters, both of which are optional.

- The first parameter ([warningText](#)) is an [RFA_String](#) that can be populated with the information about the message.
- The second parameter ([isReissue](#)) is a flag to indicate whether the request message is a reissue.

[validateMsg\(\)](#) returns a status enum based on the validation result: [MsgValidationError](#), [MsgValidationOk](#), or [MsgValidationWarning](#).

Consult the *RFA RDM Usage Guide .NET Edition* to see the values for RFA messages whether they are **Required**, **Not Used**, or **Recommended**. The message validator follows the following rules when validating messages:

- If a Required component is missing from the message, then an [InvalidUsageException](#) will be thrown.
- If a Not Used component is included in the message, then a warning text parameter will be populated.
- If a Recommended component is missing from the message, then a warning text parameter will be populated.

```
// Create a Login message (as shown in other example code)
ReqMsg reqMsg = new ReqMsg();
AttribInfo attribInfo = new AttribInfo();
ElementList elementList = new ElementList();
reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
                        ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

// Encode the username, app ID and position here (as shown in other example code)
attribInfo.Attrib = elementList;
reqMsg.AttribInfo = attribInfo;

// validate the login message
RFA_String warning = new RFA_String();
try
{
    Byte status = reqMsg.ValidateMsg( warning, false );

    // The message passed validation. It's okay to send the login request.
    ommConsumer.RegisterClient( eventQueue, ommItemIntSpec, this, null );
}
catch (InvalidUsageException e)
{
    Console.WriteLine("Invalid Message: " + e.Status.StatusText.ToString());
}
```

Example 95: Validating a Login message

7.2.2 Message Decoding

This section describes how to decode a message and its payload data for receiver. Below is the decoding process.

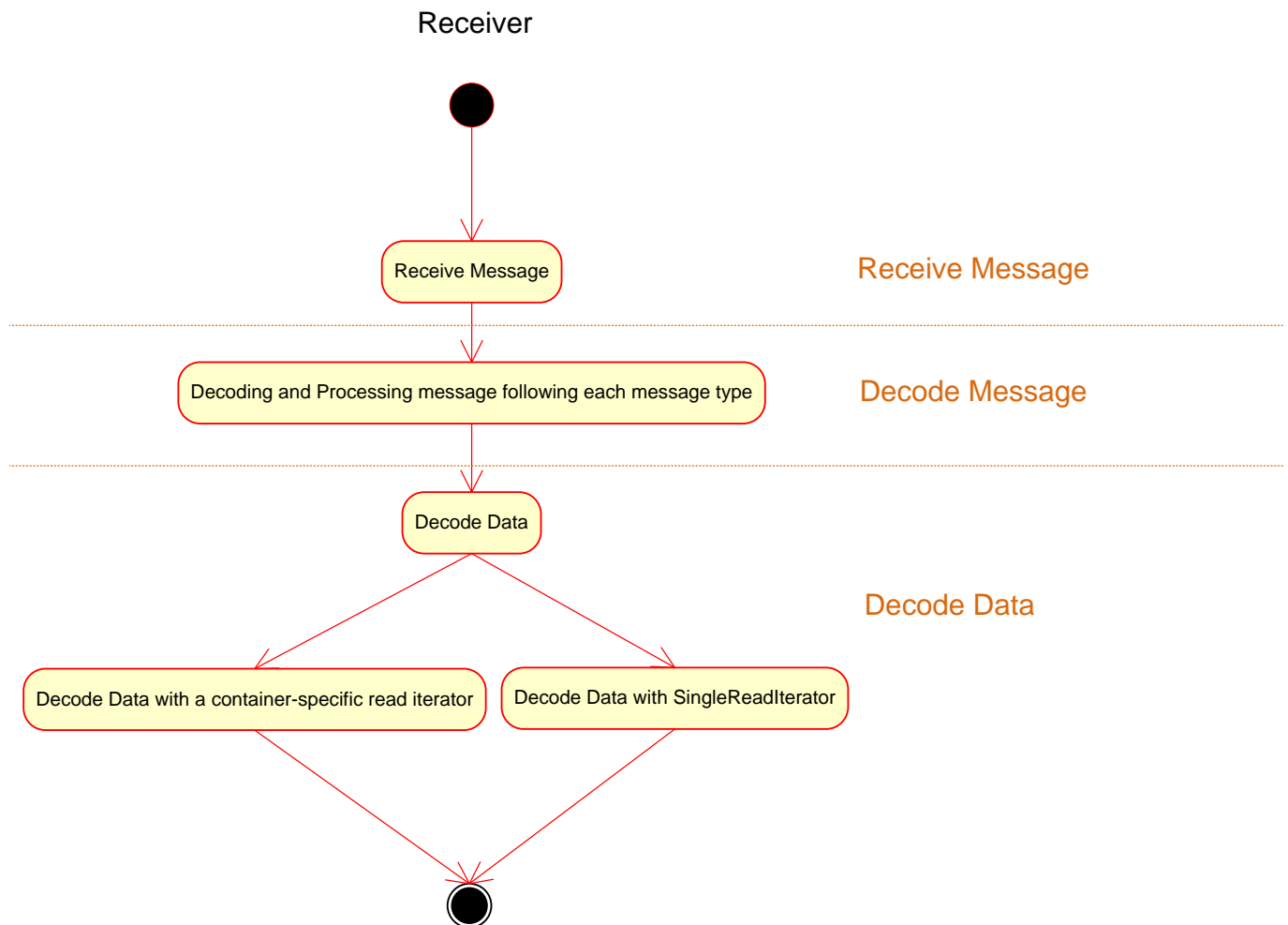


Figure 63: Decoding process

For decoding a message, the receiver has to decode it according to the message type. If the message contains payload data, then the receiver has to check whether the type of payload data is data container or data leaf. In case of data container, a container's read iterator or [SingleReadIterator](#) are used to decode it. Therefore, a consumer application gets appropriate values contained in decoded leaves.

For more information about decoding data, refer to section 6.2.2, Data Decoding.

7.2.2.1 Usage of Masks

The data or message received contains various masks to indicate the availability of various categories of information. By looking at the masks, the application knows the available information and processes them accordingly.

In the example below, the hint mask is retrieved from the `FieldList` using the `HintMask` property and checked for the presence of information using `FieldList.HintMaskFlag.Info`. If the hint mask indicates the presence of information, the details are output using `GetInfoDictID()` and `GetInfoFieldListNum()`.

```
if ( ( input.HintMask & FieldList.HintMaskFlag.Info ) != 0)
{
    Console.WriteLine("Dict ID : " + input.GetInfoDictID() + "; FieldListNum : "+
input.GetInfoFieldListNum());
}
```

Example 96: Usage of Masks

7.2.2.2 Decoding Message Type

An application uses the `Msg.MsgType` interface to determine what type of message was received and which process should be applied. This processing applies to the messages received directly from events, e.g., `OMMItemEvent` or `OMMSolicitedItemEvent`, as well as to the messages received as payloads of other messages. The application then casts the message received into a concrete message class. The code snippet below illustrates this behavior.

Further decoding of received messages (e.g., getting attributes, payload, manifest) follows the patterns described in this manual in the relevant sections.

```
void DecodeMsg(Msg msg)
{
    switch (msg.MsgType)
    {
        case Message.MsgTypeEnum.ReqMsg:
            DecodeReqMsg(msg as ReqMsg);
            break;
        case Message.MsgTypeEnum.RespMsg:
            DecodeRespMsg(msg as RespMsg);
            break;
        case Message.MsgTypeEnum.GenericMsg:
            DecodeGenericMsg(msg as GenericMsg);
            break;
        case Message.MsgTypeEnum.PostMsg:
            DecodePostMsg(msg as PostMsg);
            break;
        case Message.MsgTypeEnum.AckMsg:
            DecodeAckMsg(msg as AckMsg);
            break;
        default:
            outer.Out("DecodeData() - Invalid message type.\r\n");
            break;
    }
}
```

Example 97: Decoding Message Type

7.2.2.3 Decoding Request Message

After checking the message type using the `Msg.MsgType` property, the application retrieves the Request Message from the Event and processes the message based upon the Message Model type. The `Msg` property is used to retrieve an

encapsulated Message from the Event. If the message type is `MsgTypeEnum.ReqMsg`, then the application needs to downcast the retrieved message to `ReqMsg`.

The request may be an open request or a close request. The provider application determines the request type by the `InteractionType` property in the request message.

For a close request, the provider application needs to close the item previously opened. The request token of the previously opened item can be retrieved from the event using the `RequestToken` property.

For an open request, the provider application processes the request message based on the Message Model Type. The Message Model type is obtained using the `MsgModelType` property.

The following example shows how to decode the market price request message. See the `StarterProvider_Interactive` example for more details.

```
void ProcessReqMsg(OMMSolicitedItemEvent evnt)
{
    ReqMsg msg = (ReqMsg)(evnt.Msg);
    if (((msg.InteractionType & ReqMsg.InteractionTypeFlag.InitialImage) == 0) && ((msg.InteractionType &
ReqMsg.InteractionTypeFlag.InterestAfterRefresh) == 0))
    {
        ProcessCloseReq(evnt);
        return;
    }

    switch (msg.MsgModelType)
    {
        case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE:
        {
            ProcessMarketPriceReq(evnt);
        }
        break;
        //...
    }
}

void ProcessMarketPriceReq(OMMSolicitedItemEvent evnt)
{
    ReqMsg reqMsg = (ReqMsg)(evnt.Msg);

    if ((reqMsg.InteractionType == (ReqMsg.InteractionTypeFlag.InitialImage |
ReqMsg.InteractionTypeFlag.InterestAfterRefresh)))
    {
        // ...
    }
    else if ((reqMsg.InteractionType & ReqMsg.InteractionTypeFlag.InitialImage) != 0)
    {
        // ...
    }
    else
    {
        // this request was paused / resumed
    }
}
```

Example 98: Decoding Request Message

Provider application normally responds to a request by sending Response Message, whereas hybrid application forwards the request to other providers. The resulting response message is sent to the originating consumer. For Batch and View example, refer to the StarterProvider_BatchView example.

7.2.2.4 Decoding Response Message

After checking the message type using the `Msg.MsgType` property, the application retrieves the Response Message from the Event and processes the message based upon the Message Model type. The `Msg` property is used to retrieve an encapsulated Message from the Event. If the message type is `MsgTypeEnum.RespMsg`, then the application needs to downcast the retrieved message to `RespMsg`.

The consumer application processes the response message based on the Message Model Type. The Message Model type is obtained using the `MsgModelType` property. `Manifest`, `AttribInfo`, and `PrincipalIdentity` may be available in the Response Message. You can retrieve availability of this information using HintMask and check it using the flags `RespMsg.HintMaskFlag.Manifest`, `RespMsg.HintMaskFlag.AttribInfo` and `RespMsg.HintMaskFlag.PrincipalIdentity` respectively. For decoding processs for payload, refer to section 6.2.2,Data Decoding.

The following example shows how to decode the market price response message and extract Publisher Principal Identity (i.e. VPI) from `Reuters.RFA.Common.PrincipalIdentity`. The visible publisher identification can be found in the `Reuters.RFA.Common.PrincipalIdentity` class. This class provides both the publisher ID and publisher address. The following is the snipet code from the StarterConsumer_Post example.

```
void ProcessRespMsg(OMMItemEvent evnt, RespMsg respMsg)
{
    switch (respMsg.MsgModelType)
    {
        case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE:
            ProcessMarketPriceResponse(evnt, respMsg);
            break;
        //...
    }
}

void ProcessMarketPriceResponse(OMMItemEvent evnt, RespMsg respMsg)
{
    //...

    // Get Visible Publisher Identity that sent the post message
    if ((respMsg.HintMask & RespMsg.HintMaskFlag.PrincipalIdentity) != 0) &&
        (respMsg.PrincipalIdentity.IdentityType ==
        CommonPrincipalIdentityTypeEnum.PublisherPrincipalIdentity))
    {
        PublisherPrincipalIdentity ppi = (PublisherPrincipalIdentity)respMsg.PrincipalIdentity;
        UInt32 userAddress = ppi.UserAddress;
        UInt32 userID = ppi.UserID;
        text += " from publisher (user ID: ";
        text.Append(userID);
        text.Append(") at user address: ";
        text.Append(userAddress);
        text.Append("\r\n");
    }

    if ((respMsg.HintMask & RespMsg.HintMaskFlag.AttribInfo) != 0)
    {
        AttribInfo attribInfo = respMsg.AttribInfo;
        byte hint = attribInfo.HintMask;
        if ((hint & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            text.Append("\r\n    serviceName : ");
        }
    }
}
```

```

        text.Append(attribInfo.ServiceName);
    }
    if ((hint & AttribInfo.HintMaskFlag.Name) != 0)
    {
        text.Append("\r\n    symbolName : ");
        text.Append(attribInfo.Name);
    }
}

if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
{
    RespStatus status = respMsg.RespStatus;
    text.Append(OMMStrings.RespStatusToString(status));
}
AppUtil.Log(AppUtil.LEVEL.TRACE, text.ToString());

if ((respMsg.HintMask & RespMsg.HintMaskFlag.Payload) != 0)
{
    decoder.DecodeData(respMsg.Payload);
}

```

Example 99: Decoding Response Message

7.2.2.4.1 Decoding Batch Item Response Message

Individual item responses are sent back to the client after the provider receives a batch request.

The first response (refresh or status) for an item specified in a batch item name list always includes the item name and the item handle. The client can use the item handle to identify subsequent Response Messages for the item or perform any kind of reissue on the newly created stream.

For recalling encoding process, a batch request can be associated with a unique closure when calling [RegisterClient\(\)](#) by passing in the unique closure pointer. Individual item handles for items from the batch request contain the closure pointer. The application can retrieve the closure from an item handle to identify which batch request the item came from.

```

private enum ReqType
{
    SingleReq = 0,
    BatchReqForBatchItemList1,
    BatchReqForBatchItemList2
};

private sealed class ItemInfo
{
    public ReqType ReqType;
    public long ItemHandle;
    public RFA_String ItemName;
};

List<ItemInfo> itemList = new List<ItemInfo>();

void ProcessMarketPriceResponse(OMMItemEvent evnt, RespMsg respMsg)
{
    RFA_String temp = new RFA_String("<- Received MMT_MARKET_PRICE ");
    temp.Append(OMMStrings.MsgRespTypeToString(respMsg.RespType));
    if (evnt.Closure != null)
    {
        temp.Append(" ");
        temp.Append(evnt.Closure as RFA_String);
    }
}

```

```

    }

    long itemHandle = evnt.Handle;
    ItemInfo theItem = null;
    if ((respMsg.RespType == RespMsg.RespTypeEnum.Refresh || respMsg.RespType ==
        RespMsg.RespTypeEnum.Status)
        && ((respMsg.HintMask & RespMsg.HintMaskFlag.AttribInfo) != 0))
    {
        theItem = FindItemFromMap(respMsg.AttribInfo.Name, itemHandle);
        AttribInfo attribInfo = respMsg.AttribInfo;
        byte hint = attribInfo.HintMask;
        if ((hint & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            temp.Append("\r\n    serviceName : ");
            temp.Append(attribInfo.ServiceName);
        }
        if ((hint & AttribInfo.HintMaskFlag.Name) != 0)
        {
            temp.Append("\r\n    symbolName : ");
            temp.Append(attribInfo.Name);
        }
    }
    else
    {
        theItem = FindItemFromMap(null, itemHandle);
    }
    if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
    {
        RespStatus status = respMsg.RespStatus;
        temp.Append(OMMStrings.RespStatusToString(status));
    }
    if (theItem == null)
    {
        AppUtil.Log(AppUtil.LEVEL.ERR, string.Format("{0} received invalid item data from server",
temp.ToString()));
        return;
    }
    else
    {
        if (theItem.ItemName.Empty())
        {
            AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("{0}\r\n    Batch handle: {1}\r\n
itemFrom      : {2}",
temp.ToString(), theItem.ItemHandle, ReqTypeToString(theItem.ReqType)));
        }
        else
        {
            AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("{0}\r\n    itemName      : {1}\r\n
itemHandle    : {2}\r\n    itemFrom      : {3}",
temp.ToString(), theItem.ItemName.ToString(), theItem.ItemHandle,
ReqTypeToString(theItem.ReqType)));
        }
    }
    if ((respMsg.HintMask & RespMsg.HintMaskFlag.Payload) != 0)
    {
        decoder.DecodeData(respMsg.Payload);
    }
}

// The client matches the item name with the item handle, then stores the item handle

```

```

// in a container for future use.
ItemInfo FindItemFromMap(RFA_String itemName, long itemHandle)
{
    int count = itemList.Count;

    //look for the item by handle
    for (int i = 0; i < count; i++)
    {
        if (itemList[i].ItemHandle == itemHandle)
        {
            return itemList[i];
        }
    }

    //If its the first response for an item in a batch we need look it up by name and insert the handle
    //also, we might have duplicate items. So we might have two entries in the table with the same name
    if (itemName != (RFA_String)null)
    {
        ItemInfo item = null;
        for (int i = 0; i < count; i++)
        {
            item = itemList[i];
            if ((item.ItemHandle == 0) && item.ItemName == itemName)
            //If the item name matches and there is no handle yet
            {
                itemList[i].ItemHandle = itemHandle;
                return itemList[i];
            }
        }
    }
    return null;
}

```

Example 100: Processing Batch Item Response

7.2.2.5 Decoding Post Message

After checking the message type using the `Msg.MsgType` property, the application retrieves the Post Message from the Event and processes the message based upon the Message Model type. The property `Msg` is used to retrieve an encapsulated Message from the Event. If the message type is `MsgTypeEnum.PostMsg`, then the application needs to downcast the retrieved message to `PostMsg`.

The provider application determines the content available by the `HintMask` property in the post message. For decoding processs for payload, refer to section 6.2.2,Data Decoding.

The following example shows how to decode the post message. For more example, refer to the `StarterProvider_Post` example.

```
void ProcessPostMsg(OMMSolicitedItemEvent evnt)
{
    PostMsg postMsg = evnt.Msg as PostMsg;
    RequestToken reqToken = evnt.RequestToken;
    long CSH = evnt.Handle;
    byte hintMask = postMsg.HintMask;
    byte indicationMask = postMsg.IndicationMask;

    RFA_String text = new RFA_String("<- Received PostMsg ");
    // Get Visible Publisher Identity that sent the post message:
    PrincipalIdentity ppi = postMsg.PrincipalIdentity;
    if ( ppi.IdentityType == CommonPrincipalIdentityTypeEnum.PublisherPrincipalIdentity)
    {
        PublisherPrincipalIdentity ppId = (PublisherPrincipalIdentity)(ppi);
        UInt32 userAddress = ppId.UserAddress;
        UInt32 userID = ppId.UserID;
        text += " with publisher [user ID: ";
        text.Append(userID);
        text.Append("] at [user address: " );
        text.Append( userAddress );
        text.Append( "]" );
    }
    text += "from Client Session Handle:";

    text.Append((ulong)CSH);
    AppUtil.Log(AppUtil.LEVEL.TRACE, text.ToString());

    ClientWatchList cw1 = providerWatchList.GetClientWatchList(CSH);
    ClientWatchList.TokenInfo TS = cw1.GetTokenInfo(reqToken);

    // Populate attribInfo of the response message according to the following rule:
    // 1) If this is off-stream post message (matches login request token),
    //     then get attribInfo from the post message; it MUST be there!
    // 2) If this is on-stream post message (matches item request token),
    //     then get it from the TokenInfo which contains attribInfo received
    //     with this items request; attribInfo may or may not be on the post message
    //     since it is not needed (it may be optimized out by consumer app)
    AttribInfo ai;

    if (TS.isItemRequest)
    {
        ai = TS.attribInfo;
    }
    else
    {

```

```

        ai = postMsg.AttribInfo;
    }

    uint postID = 0;
    bool isPostIdSet = false;
    if ((hintMask & PostMsg.HintMaskFlag.PostId) != 0)
    {
        isPostIdSet = true;
        postID = postMsg.PostID;
    }

    UInt32 sequenceNum = 0;
    bool isSequenceNumSet = false;
    if ((hintMask & PostMsg.HintMaskFlag.Seq) != 0)
    {
        isSequenceNumSet = true;
        sequenceNum = postMsg.SeqNum;
    }

    bool isWantAck = ((indicationMask & PostMsg.IndicationMaskFlag.WantAck) != 0) ? true : false;

    if (isWantAck)
    {
        RFA_String text1 = new RFA_String();
        if (cfgVariables.IsIgnoreNackCode)
        {
            text1.Append("Positive AckMsg");
        }
        else
        {
            text1 = cfgVariables.NegativeAckCode;
        }

        SubmitAckMsg(reqToken, CSH, isPostIdSet, postID, isSequenceNumSet, sequenceNum,
            !cfgVariables.IsIgnoreNackCode, nackCodeValue, text1, postMsg.MsgModelType, ai);
    }

    //Re-send this post to all clients that subscribed to this item
    //But only if the postMsg's payload is a response message

    if ((hintMask & PostMsg.HintMaskFlag.Payload) != 0)
    {
        RFA.Common.Data payload = postMsg.Payload;
        if (DataEnum.Msg == payload.DataType)
        {
            byte msgMMT = postMsg.MsgModelType;
            Msg msg = payload as Msg;
            byte payloadMMT = msg.MsgModelType;
            if (MsgTypeEnum.RespMsg == msg.MsgType && msgMMT == payloadMMT)
            {
                RespMsg respMsg = msg as RespMsg;
                RespMsg ncRespMsg = respMsg as RespMsg;

                List<RequestToken> requestTokens = new List<RequestToken>();
                List<long> clientSessions = new List<long>();

                providerWatchList.FindTokens(ai.Name, msgMMT, requestTokens, clientSessions);

                SubmitPostedData(ncRespMsg, ppi, ai.ServiceName, requestTokens, clientSessions);
            }
        }
    }

```

```

    }

    decoder.DecodeData(payload);
}
}

```

Example 101: Decoding Post Message

7.2.2.6 Decoding Ack Message

After checking the message type using the `Msg.MsgType` property, the application retrieves the Ack Message from the Event and processes the message based upon the Message Model type. The `Msg` property is used to retrieve an encapsulated Message from the Event. If the message type is `MsgTypeEnum.AckMsg`, then the application needs to downcast the retrieved message to `AckMsg`.

The consumer application determines the content available by the `HintMask` property in the ack message. For decoding processs for payload, refer to section 6.2.2, Data Decoding.

The following example shows how to decode the ack message. For more example, refer to the StarterProvider_Post example.

```

void ProcessAckMsg(OMMItemEvent evnt, AckMsg ackMsg)
{
    //...

    if ((ackMsg.HintMask & AckMsg.HintMaskFlag.AttribInfo) != 0)
    {
        temp.Append("\r\n    AttribInfo received:");

        AttribInfo atInfo = ackMsg.AttribInfo;

        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.Name) != 0)
        {
            temp.Append("\r\n    name          : ");
            temp.Append(atInfo.Name);
            temp.Append("\r\n    nameType      : ");
            temp.Append((Int32)atInfo.NameType);
        }
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            temp.Append("\r\n    serviceName   : ");
            temp.Append(atInfo.ServiceName);
        }
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.ID) != 0)
        {
            temp.Append("\r\n    ID            : ");
            temp.Append(atInfo.ID);
        }
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.Attrib) != 0)
        {
            temp.Append("\r\n    Attribute Data Type: ");
            temp.Append((UInt32)atInfo.Attrib.DataType);
        }
    }
    else
    {
        temp.Append("\r\n    AttribInfo : NOT specified");
    }
}

```



```

temp.Append("\r\n    AckId      : ");
temp.Append(ackMsg.AckID);
if ((ackMsg.HintMask & AckMsg.HintMaskFlag.Seq) != 0)
{
temp.Append("\r\n    SequenceNo : ");
temp.Append(ackMsg.SeqNum);
}
else
{
temp.Append("\r\n    SequenceNo : not received");
}
if ((ackMsg.HintMask & AckMsg.HintMaskFlag.Text) != 0)
{
temp.Append("\r\n    Received TextFlag");
temp.Append("\r\n    Status text : ");
temp.Append(ackMsg.Text);
}
if ((ackMsg.HintMask & AckMsg.HintMaskFlag.NackCode) != 0)
{
temp.Append("\r\n    Received NackCodeFlag");
temp.Append("\r\n    Nack code      : ");
temp.Append((UInt32)ackMsg.NackCode);
}
else
{
temp.Append("\r\n    NackCodeFlag not received (Positive Ack)");
}
if ((ackMsg.HintMask & AckMsg.HintMaskFlag.Payload) != 0)
{
// Decode payload
}
else
{
temp.Append("\r\n");
}
}

```

Example 102: Decoding Ack Message

7.2.2.7 Decoding Generic Message

After checking the message type using the `Msg.MsgType` property, the application retrieves the Generic Message from the Event and processes the message based upon the Message Model type. The property `Msg` is used to retrieve an encapsulated Message from the Event. If the message type is `MsgTypeEnum.GenericMsg`, then the application needs to downcast the retrieved message to `GenericMsg`.

7.2.2.7.1 Decoding Generic Message in a Consumer

The consumer application determines the content available by the `HintMask` property in the generic message. For decoding processs for payload, refer to section 6.2.2,Data Decoding.

The following example shows how to decode the generic message. For more example, refer to the `StarterConsumer_GenericMsg` example.

```
void ProcessGenericMsg(OMMItemEvent evnt, GenericMsg genericMsg)
{
    //...

    if ((genericMsg.HintMask & GenericMsg.HintMaskFlag.AttribInfo) != 0)
    {
        temp.Append(" - with AttribInfo");
        AttribInfo attribInfo = genericMsg.AttribInfo;
        byte hint = attribInfo.HintMask;
        if ((hint & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            temp.Append("\r\nservice name    : ");
            temp.Append(attribInfo.ServiceName);
        }
        if ((hint & AttribInfo.HintMaskFlag.Name) != 0)
        {
            temp.Append("\r\nsymbol name      : ");
            temp.Append(attribInfo.Name);
        }
    }

    if ((genericMsg.HintMask & GenericMsg.HintMaskFlag.Payload) != 0)
    // Decode payload
}
```

Example 103: Decoding Generic Message in a Consumer

7.2.2.7.2 Decoding Generic Message in a Provider

The provider application determines the content available by the [HintMask](#) property in the generic message. For decoding processs for payload, refer to section 6.2.2,Data Decoding.

The following example shows how to decode the generic message. For more example, refer to the StarterProvider_GenericMsg example.

```
void ProcessGenericMsg(OMMSolicitedItemEvent evnt)
{
    GenericMsg genMsg = evnt.Msg as GenericMsg;
    RequestToken reqToken = evnt.RequestToken;
    long CSH = reqToken.Handle;

    OMMSolicitedItemCmd solItemCmd = new OMMSolicitedItemCmd();
    if (genericMsg == null)
    {
        genericMsg = new GenericMsg();
    }

    genericMsg.Clear();
    if ((genMsg.HintMask & GenericMsg.HintMaskFlag.AttribInfo) != 0)
    {
        RFA_String name = new RFA_String("\r\n    name    : ");
        RFA_String service = new RFA_String("\r\n    service : ");
        AttribInfo atInfo = genMsg.AttribInfo;
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.Name) != 0)
        {
            name.Append(atInfo.Name);
        }
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            service.Append(atInfo.ServiceName);
        }
        AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("<- Received OMMSolicitedItemEvent with Generic Message {0}{1}{2}",
            OMMStrings.SolicitedItemEventToString(evnt).ToString(), name.ToString(), service.ToString()));
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.Attrib) != 0)
        {
            decoder.DecodeData(genMsg.AttribInfo.Attrib);
        }
        genericMsg.AttribInfo = (genMsg.AttribInfo);
    }
    else
    {
        AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("<- Received OMMSolicitedItemEvent with Generic Message {0}", OMMStrings.SolicitedItemEventToString(evnt).ToString()));
    }
    if ((genMsg.HintMask & GenericMsg.HintMaskFlag.Payload) != 0)
    {
        decoder.DecodeData(genericMsg.Payload);
    }
    else
    {
        AppUtil.Log(AppUtil.LEVEL.WARN, "No payload in received Generic Msg");
    }

    ElementList payLoad = new ElementList();
    payLoad.SetAssociatedMetaInfo(reqToken.Handle);
    ElementListWriteIterator iter = new ElementListWriteIterator();
```

```

ElementEntry entry = new ElementEntry();
DataBuffer buffer = new DataBuffer();
RFA_String ename = new RFA_String();
RFA_String val = new RFA_String();

iter.Start(payload);
ename.Set("Data");
val.Set("Provider's <GENERIC> Message");
buffer.SetFromString(val, DataBuffer.DataBufferEnum.StringAscii);
entry.Name = ename;
entry.Data = buffer;
iter.Bind(entry);
iter.Complete();

genericMsg.Payload = payload;
genericMsg.MsgModelType = (byte)(cfgVariables.GenericMsgModelType);
genericMsg.IndicationMask = GenericMsg.IndicationMaskFlag.MessageComplete;

solItemCmd.Msg = genericMsg;
solItemCmd.RequestToken = reqToken;
Submit(solItemCmd, null, new RFA_String("GENERIC_MESSAGE"));
}

```

Example 104: Decoding Generic Message in a Provider

Chapter 8 Configuration Package

8.1 Configuration Package Concepts

The **Configuration Package** provides access and management of a hierarchical, in-memory configuration storage. Applications may populate this configuration storage programmatically or from various persistent repositories such as the Windows Registry or flat-file.

Alternative RFA packages such as SessionLayer or Logger Package use the Configuration Package to retrieve configuration information. An application must initialize and populate the particular Config Database instance identified by the Context name (must be “RFA”) before using the Session Layer or Logger Package as they retrieve configuration information from this Config Database instance.

Applications may initialize and populate other Config Database instances for their own use. Applications identify these Config Database instances by an alternate name (i.e., not “RFA”). Similarly, applications may retrieve configuration information from these Config Database instances.

8.1.1 Hierarchy Representation, Config Trees, Config Values, and Namespaces

The Configuration Package introduces several concepts related to populating and querying configuration information. The following table defines these concepts. Subsequent sections describe interfaces (or methods) related to these concepts.

CONCEPT	DESCRIPTION
Config Database	In-memory hierarchical configuration storage accessible by alternative RFA packages and applications. Alternative RFA packages always use the particular Config Database instance identified by the Context name (must be “RFA”).
Config Node	A specific location within the hierarchy of a Config Database. A Config Node may be any one of specific concrete type (e.g., Config String, Config Long).
Config Repository	A Persistent hierarchical storage such as Windows Registry or flat file.
Config Tree	A specific concrete type of Config Node that typically contains a set of child Config Nodes.
Root Config Tree	The uppermost Config Tree in a Config Database.
Namespace	A Second uppermost Config Trees used to logically group sets of configuration hierarchies in a Config Database.
Softlink	A specific type of Config Node used to point to another Config Node for configuration sharing.
Staging Config Database	Temporary in-memory hierarchical storage used to populate a Config Database

Table 64: General Configuration Package Concepts

Consumer and Provider applications usually require some sort of configuration from persistent storage, such as relational databases, flat files, web-based resources, Windows Registry, legacy systems, etc. The Config Package is a platform independent, thread-safe solution that provides the applications, and RFA itself, with an easy-to-use, uniform access to a variety of **Configuration Repositories**.

The Config Package hides differences between various Configuration Repositories and exposes configuration information via in-memory configuration databases, or **Config Databases**. A Config Database can be used by Packages within RFA and can also be used by the application. A Config Database is identified by its name. Usually, the RFA Packages share the same instance of the Config Database with the well-known name documented in the *RFA Configuration Guide .NET Edition*.

The Config Package allows for loading configuration information from the supported Configuration Repositories into a Config Database. This process is also known as populating the database. Populating is performed by the application and can be done incrementally. Applications can also populate a Config Database programmatically, which simplifies integration with application-specific persistent configuration storage. (Saving a Config Database to persistent storage is not supported.)

Population is done via a **Staging Configuration Database** and can be done incrementally. This process involves loading the Staging Configuration Database and merging it into the Configuration Database. Loading of a Staging Configuration Database is supported in two ways:

- From the Windows Registry or a flat file
- At runtime by an application

The following diagram illustrates the relationship between the Staging Configuration Database, the Configuration Database, and an RFA package, the Session Layer, that uses the Configuration Database.

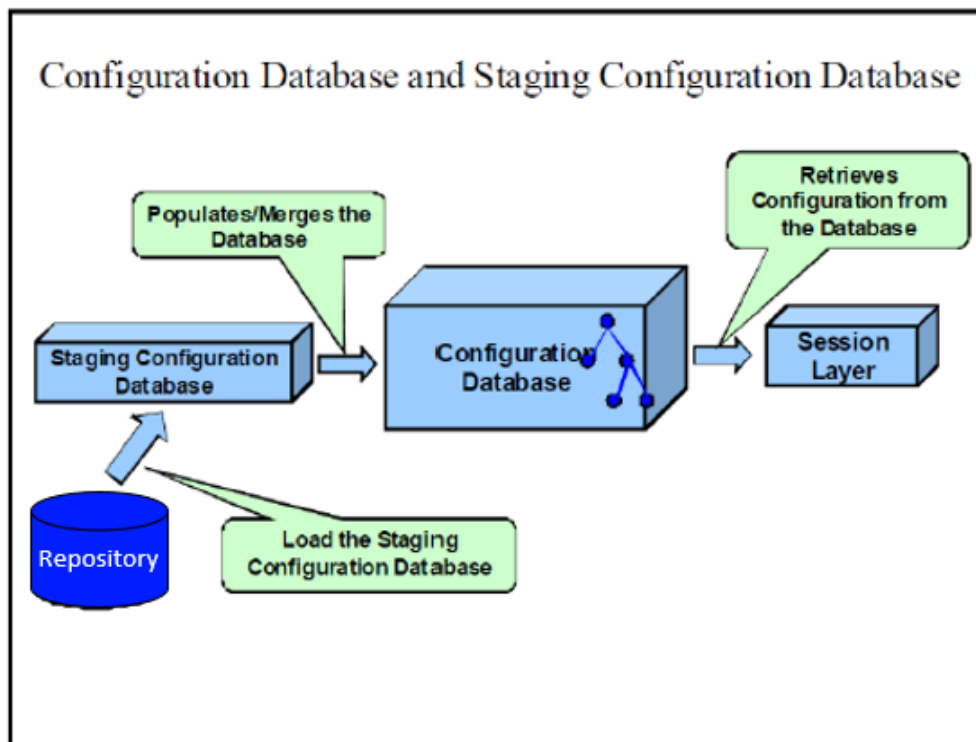


Figure 64: Configuration Database and Staging Configuration Database

Staging Configuration Databases, regardless of the mechanism used to load them, can be merged into a single Configuration Database. In the case of duplicate leaf nodes, the first loaded value will be maintained (i.e. values are not overwritten). This is true for both the Configuration Database and the Staging Configuration Database.

A Config Database stores hierarchical configuration in a configuration tree, or **Config Tree**, which, in turn, is a collection of Config Nodes. A Config Tree is itself a special kind of Config Node. The main Config Tree, which represents the entire Config Database, is called the **Root Config Tree**.

Config Nodes that belong to a Config Tree are called **Child Config Nodes**, or **Children**. The Config Tree in this case is the parent for these Nodes. If a Child Config Node is a Config Tree, it's called a **Config Subtree**.

Each Config Node is identified by its name property. Names are case insensitive and can contain white spaces and other control characters.

Besides Config Trees, the other kind of Config Nodes are **Config Values**, which represent terminal, or leaf, nodes in the configuration hierarchy. Config Values contain actual configuration data and are associated with a data type. Config

Values are represented as name/data pairs. For a list of data types supported by the Config Package, see Table 65: Data Types Supported by the Config Package.

A Config Node can be located in a Config Database using its **Fully Qualified Path**, which includes, in reverse order, name of the Config Node, name of its parent, name of its parent's parent, etc. up to the Root Config Tree; the names are separated by a "\" character. Similarly, a **Relative Path** can be used to locate a Config Node in a Config Subtree.

Softlinks allow configuration sharing within a Configuration Database. They are a special kind of Config Node that point to another Config Node called the **Target Config Node**. Softlinks are conceptually similar to Windows Explorer shortcuts or symbolic links on Unix file systems.

To allow for component-style application development and prevent name conflicts the Config Package supports logical grouping of configuration into separate **Namespaces**. A Namespace is an arbitrarily chosen name. Each Namespace corresponds to a Config Subtree of the Root Config Tree. This subtree is called a **Namespace Tree** and has the same name as the Namespace. Namespaces are always specified programmatically. Namespace names can contain any character, including '\'. The actual contents of the respective Namespace Tree can be populated from a persistent storage or programmatically.

NOTE: The recommended way of naming namespaces is on a per application or even a per application component basis. This reduces the chances of name conflicts and enables configuration separation.

The figure below demonstrates the above terms and concepts using an example Config Database that contains three Namespaces: `default`, `quoteAnalyzer`, and `chartDisplay`. Each Namespace Tree contains a hierarchy of Config Trees and Config Values of various types. Examples of a Fully Qualified Path and a Relative Path to a Config Node are also shown in this diagram.

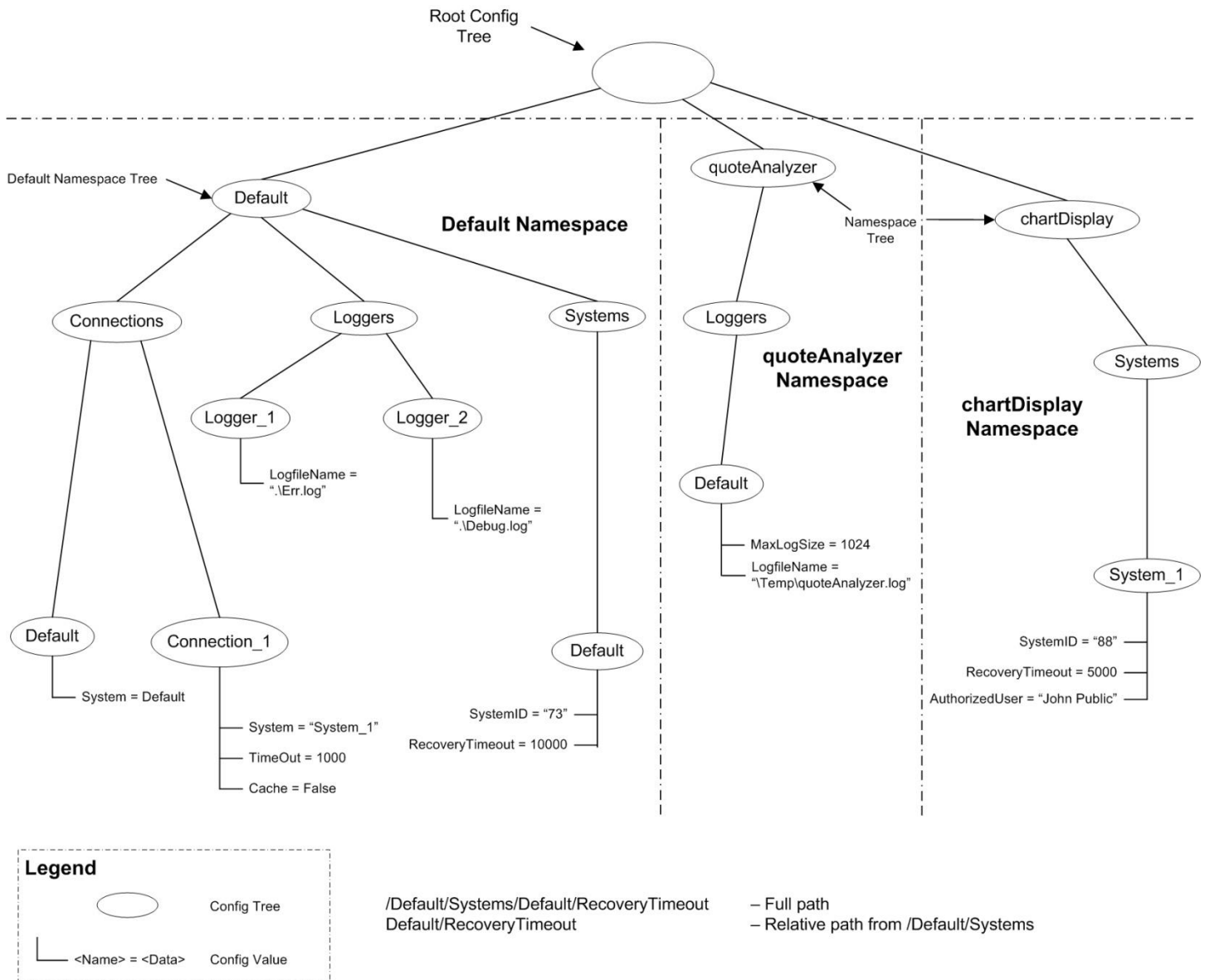


Figure 65: Terms and Concepts of the Config Package by Example

For more information about all of the configuration parameters available to use in RFA, see the *RFA Configuration Guide .NET Edition*.

8.1.2 RFA Namespaces

Namespaces are used by RFA-compliant packages to prevent name conflicts and to support component-style application development. They are represented in the Config Database as a **ConfigTree** that is a child of the Root Config Tree.

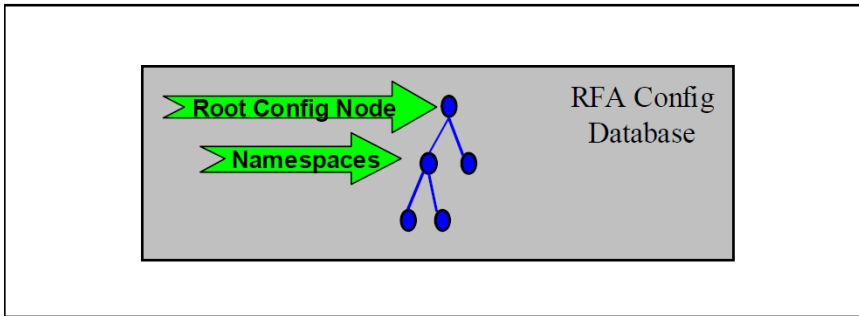


Figure 66: RFA Configuration Database and Namespace

8.1.2.1 RFA Component Names and the Default Namespace

RFA-compliant software usually consists of one or more components that implement package-specific interfaces.

Component Names are used to identify specific component instances and provide for configuration. A name is specified using one of the following formats:

- namespace::compName
- ::compName
- compName

Examples include "Default::Component1", "namespace::Component1", "::Component1", and "Component1."

For named Components exposed through the interface, if a namespace is not provided, RFA packages will automatically assume the "Default" namespace (e.g., "Component1" becomes "Default::Component1").

One exception exists for named RFA interfaces. The Logger interface does not use the namespace concept. The name given to the logger is a simple flat name (similar to the name given to the Configuration Database).

8.1.2.2 RFA Component Configuration

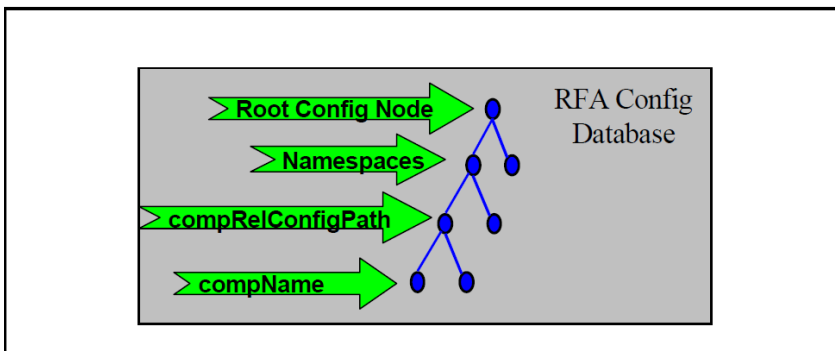


Figure 67: RFA Configuration Database Layout

The RFA components that require configuration are called **Configurable Components**. A component configures itself at startup (or when instantiated) by retrieving all the necessary configuration information from the RFA Config Database. The configuration tree of the component is located in the [ConfigDatabase](#) using the Component Name requested along with the **Component Relative Config Path** under `namespace\compRelConfigPath\compName`. The `namespace` and `compName` come from the Component Name, while `compRelConfigPath` defines the sub-path between the namespace and instance and is defined by the component itself. Most Component Relative Config Paths are defined as the plural version of the Class Name—e.g., “Components”, “Sessions”, “Event Queues”, etc.

8.1.3 Configuration Sharing and Component Instance Sharing

RFA Software Packages provide two types of components: Shareable and Non-Shareable. When an application calls an RFA API to get an instance of a Shareable Component, the API can actually return an already-existing and configured instance. For Non-Shareable Components, a new instance must be created and configured every time an application calls to get a Component Instance.

Both Shareable and Non-Shareable Components can benefit from configuration sharing by getting their configuration from the same Config Tree. The rules of Component Instance sharing are specific to each Package and covered in more detail in their respective sections in the *RFA Configuration Guide .NET Edition*.

RFA extends the concept of Softlinks with **Implicit Soft Links**. A Softlink is implied when the componentName, specified when getting a Component Instance from the API, does not have a corresponding **Config Node** (a Config Tree or a **Soft Link**) in the Config Database. When this occurs, an Implicit Soft Link is assumed from the missing Config Node to the Default Config Node in the specified Namespace Tree.

8.1.4 Types of Configuration Values

The table below summarizes the supported data types and the corresponding Config Value types.

CONFIG VALUE TYPE	DESCRIPTION	VALUE RANGE
Config Bool	Boolean value	{True, False}
Config Long	Long integer value	[-2147483648, 2147483647]
Config String	ANSI string	N/A
Config StringList	List of ANSI strings	N/A
Config StringW	Unicode string	N/A
Config StringListW	List of Unicode strings	N/A

Table 65: Data Types Supported by the Config Package

8.2 Configuration Package Usage

An application must initialize and populate the Config Database instance identified by the Context name (must be “RFA”). Optionally applications may retrieve configuration information from this Config Database instance or other Config Database instances. Applications may perform operations such as retrieving specific Config Nodes, iterating over Config Nodes or iterating across a Config Softlink. Regarding the concept of Configuration module. The following sections will describe the type of the persistent repositories in detail and show the example code of populating information from config database.

8.2.1 Populating the Config Database from the Windows Registry

The Config Package provides full support for the Windows Registry. It presents a consistent object-oriented model with configuration sharing capabilities and relieves the user from necessity to code against the Win32 Registry API.

When populating a Config Database from the Windows Registry, a registry key to load configuration from must be specified programmatically. It must begin with the **HKEY***. For example:

- **HKEY_LOCAL_MACHINE\Software\Reuters\RFA**
- **HKEY_CURRENT_USER\Software\COMPANY_A\MyAppName\ConfigLocation**

The Windows Registry values map to the Config Value types as described in the table below.

CONFIG VALUE TYPE	WINDOWS REGISTRY DATA TYPE	COMMENTS
Config Bool	REG_SZ	Only true and false are valid values (case insensitive)
Config Long	REG_DWORD	
Config String	REG_SZ	
Config StringList	N/A	Not currently supported
Config StringW	REG_SZ	
Config StringListW	N/A	Not currently supported

Table 66: Mapping between Config Values and Windows Registry Data Types

The following example is a listing of the Windows Registry-based RFA configuration.

NOTE: For configurations and their value types, refer to the *RFA Configuration Guide .NET Edition*.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\AcmeInc]
[HKEY_LOCAL_MACHINE\SOFTWARE\AcmeInc\Default]
[HKEY_LOCAL_MACHINE\SOFTWARE\AcmeInc\Default\Connections]
[HKEY_LOCAL_MACHINE\SOFTWARE\AcmeInc\Default\Connections\Connection_1]
"System"="System_1"
"SystemID"="88"
"RecoveryTimeout"=dword:000007b6
"Cache"="False"
"AuthorizedUser"="Daniel McCarty"
[HKEY_LOCAL_MACHINE\SOFTWARE\AcmeInc\Default\Connections\Default]
"System"="Default"
```

Example 105: Registry file for the Sample Configuration

8.2.1.1 Softlinks and Hardlinks

If a registry key contains a registry value of the `REG_SZ` type named `softlink`, the data in this registry value will be interpreted as a Softlink. A Config Tree will not be generated for this registry key; instead, a Softlink will be created with the same name as the registry key and its **Target Namespace** and **Target Node Name** properties will be set as specified by `targetNamespace` and `targetNodeName` respectively.

NOTE: The implementation of the Config Package resolves Softlinks during the Config Database population process and reports an error if the Target Config Node is missing.

The Config Package also supports Hardlinks, which are similar to Softlinks with the exception that they point into a location in the Configuration Repository, not in the Config Database.

If a registry key contains a registry value of the `REG_SZ` type named `hardlink`, the value will be interpreted as a Hardlink. In this case, the respective Config Tree will be created and populated with the contents of the registry key specified by the data of the registry value. This data must be another registry key location—e.g., `[HKEY_LOCAL_MACHINE\SOFTWARE\AcmeInc\Quote Analyzer\Dynamic]`.

NOTE: The Config Package verifies the contents of the registry key and, if it contains a Softlink or Hardlink, ensures that this is the only named value in this key. If this is not the case, an error is reported to the application.

8.2.2 Populating the Config Database from a File

The Config Package provides full support to load configuration data from a flat file. Typically, the file (or files) can be kept locally, but in general the file(s) can be kept on any host that is accessible via standard file system API. If the file is located at a central server, no distribution is necessary; otherwise, the file must be copied to each host. The flat file location must be specified—e.g.,

- **C:\myConfigFiles\filename.cfg**
- **/usr/configFiles/filename.cfg**

The flat file format is a Name, Value pair. The config type will not be part of the settings except the cases of softlink and hardlink, and the type will be determined by the config value.

When using a flat file, the following guidelines should be followed:

- Any line that starts with '#' is treated as a comment line
- Configuration parameters take the format of parameter = value
- The parameter name is the full configuration path name; double quotes are optional. However, double quotes are required when special characters are used as part of the name. Special characters include '=', '#', ' '(space), ':', '\', and '"'.
 - Specifications for values:
 - Boolean value must be **true** or **false** (case insensitive)
 - Long value will be the combinations of numeric chars
 - String value must be enclosed with double quotation marks
 - Softlink value uses **softlink:** as keyword followed by a string value
 - Hardlink value uses **hardlink:** as keyword followed by a string value
 - There is no space between **hardlink** (**softlink**) and ':', but spaces are allowed between the ':' and the string value.

Some examples of the flat file format are listed below.

```
\Sessions\Session1\Timeout = 10000
\Sessions\Session1\LogEnabled = true
\Sessions\Session1\LogInfo = "From Session1"
\Sessions\Session1\QueueSize = 30
\Sessions\Session2 = softlink:"Session1"
"\Sessions\Session3\Queue Size" = 60
\Sessions\Session1\Timeout = 20000
"\Sessions\Session3\" = "Equal"
\Sessions\Session4 = hardlink:"rfahardlink.txt"
\Sessions\Session5 = softlink:"Session2"
```

Example 106: Configuration Flat File Format

8.2.3 Populate Config Database

The following diagram shows the steps needed to initialize and populate configuration information into a Config Database. An application must initialize and populate the Config Database instance identified by the Context in order to use the SessionLayer Package or Logger Package. Subsequent sections describe these activities.

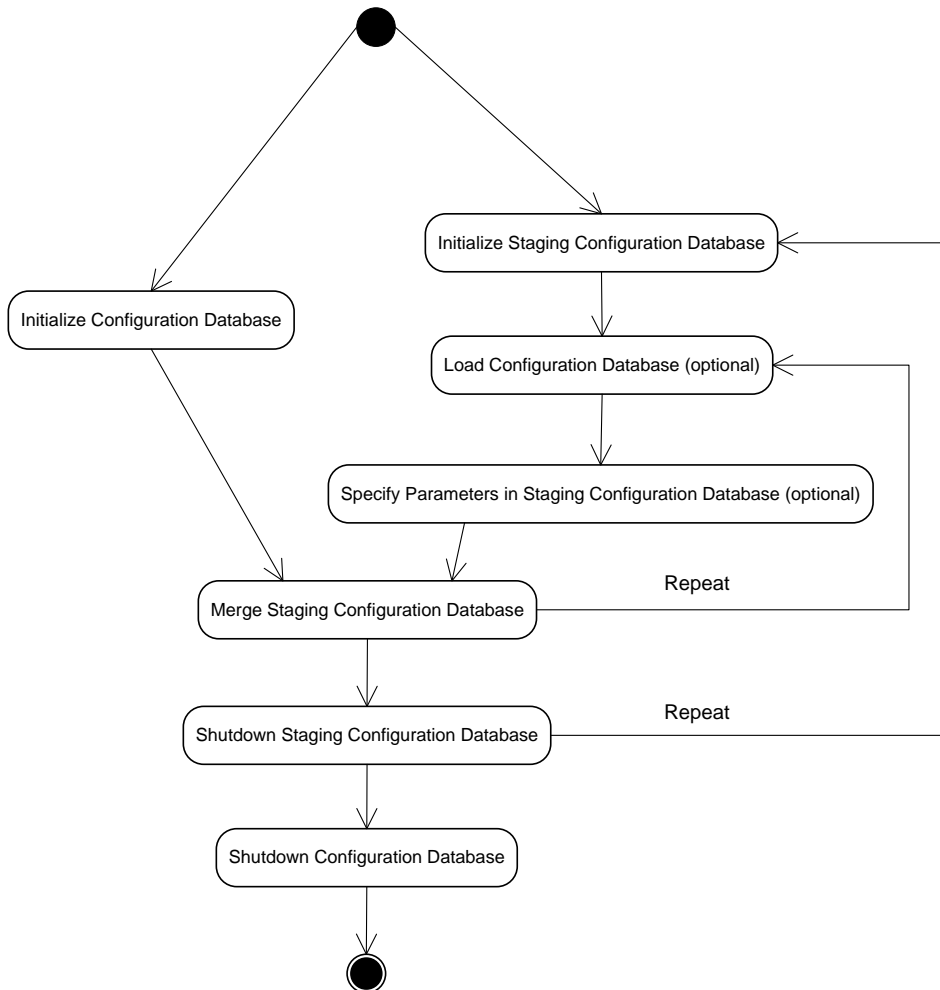


Figure 68: Config Database: Populate Configuration

8.2.3.1 Initializing and Populating the RFA Config Database

To populate a Config Database an application must perform the activities *Initialize Staging Configuration Database*, *Initialize Configuration Database* as shown above.

The application can populate the configuration information two ways:

- By loading the Staging Config Database: inputs configuration information from a Config Repository.
- By specifying in the Staging Config Database: inputs configuration information programmatically.

As shown in the figure above, the application may repeat either or both of these activities on the same Staging Config Database instance. RFA always appends newly added configuration information and will not overwrite an existing configuration parameter's value with a new value.

To populate the Config Database the application should merge the Staging Config Database. As shown in the figure above, an application may repeat merging several Staging Config Databases instances into a single Config Database. When the application no longer needs a *Staging Configuration Database*, it should shut down the Staging Configuration Database.

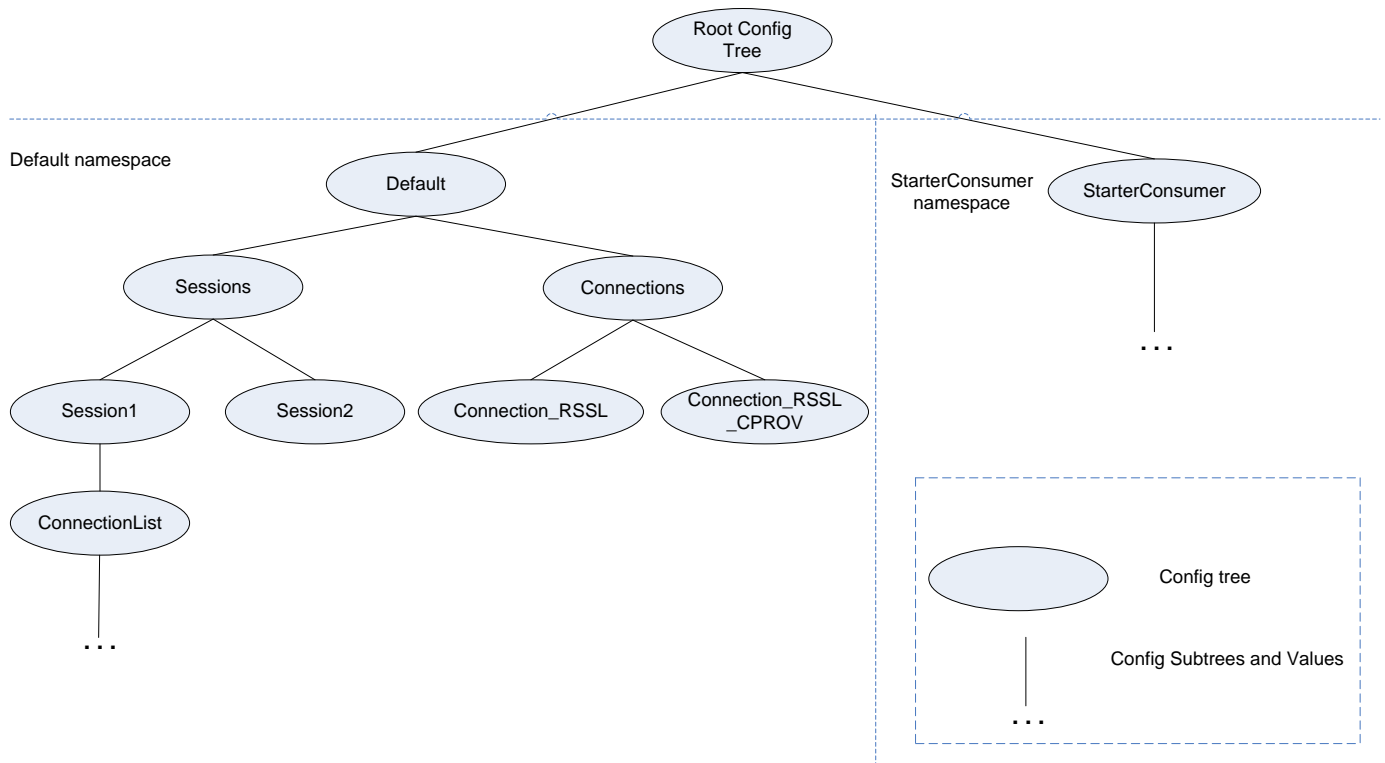


Figure 69: Config Database: Configuration Repository Example

The Figure 69 shows tree structure of configuration database of default configuration and configuration of StarterConsumer example application. The following example will use this configuration hierarchical to show how to implement the application to retrieve information from this structure:

```
// --- Initialize RFA Config Database ---
// Instantiate and load RFA Staging Config Database
try
{
    Reuters.RFA.Config.StagingConfigDatabase stagingConfigDatabase = StagingConfigDatabase.Create();
    if (stagingConfigDatabase == null)
    {
        System.Console.WriteLine("Cannot create StagingConfigDatabase.");
        Environment.Exit(-1);
    }
    bool loadRetVal = stagingConfigDatabase.Load(ConfigRepositoryTypeEnum.windowsRegistry, new
RFA_String("HKEY_LOCAL_MACHINE\\SOFTWARE\\Reuters\\RFA\\Default"));
    if (!loadRetVal)
    {
        System.Console.WriteLine("Cannot load configuration depository by StagingConfigDatabase.");
        Environment.Exit(-1);
    }

    // Acquire Application ConfigDatabase
    Reuters.RFA.Config.ConfigDatabase configDatabase = ConfigDatabase.Acquire(new RFA_String("RFA"));
    if (configDatabase == null)
    {
        System.Console.WriteLine("Cannot create ConfigDatabase.");
        Environment.Exit(-1);
    }

    //Merge configuration from the StagingConfigDatabase into ConfigDatabase with "Default" namespace
    configDatabase.Merge(stagingConfigDatabase);

    // Shutdown Staging Config Database. It is no longer needed.
    stagingConfigDatabase.Destroy();
}
catch (Reuters.RFA.Common.InvalidUsageException e)
{
    System.Console.WriteLine(e.Status.StatusText.ToString());
}
```

Example 107: Initialize RFA Config Database

In the above example, the application first creates a Staging Config Database by calling its `Create()` method. The `Create()` method returns a handle to a unique instance of a Staging Config Database (`stagingConfigDatabase`). The application then populates the Staging Config Database by calling the `Load()` method specifying the repository type (`windowsRegistry`) and location within the Config Repository. Alternatively, the application may have specified the type `flatFile`, which instructs the Staging Config Database to load from a file. The application may optionally append configuration parameters by using the various `SetXXX()` methods —e.g., `SetBool()`, `SetLong()`, `SetString()`, etc.— available in the Staging Config Database interface.

When using various `Setxxx()` methods to load or append configuration parameters, RFA will compare the known configuration parameter types with any input configuration parameter types. If there is a mismatch, then an exception is thrown. It is the application's responsibility to catch the exception as shown in the preceding example. For example, if an application attempts to load the valid known configuration parameter name `rsslPort` (whose type is `string`) as a type `long`, then an exception is thrown.

Next the application initializes the Config Database by calling the static `Acquire()` method. This method requires one parameter, which is the Config Database name. RFA packages always use the Config Database name defined by the Context. This name is "RFA." The static `Acquire()` method returns a handle to a Config Database (`configDatabase`).

RFA may share the instance returned by the `Acquire()` method among other application components. This is known as **Instance sharing**. RFA bases a unique instance of a Config Database on its name. Benefits of instance sharing include alleviating the need for the application to pass around a Config Database handle amongst multiple application components and ability to minimize resource consumption.

Next, the application merges the contents of the Staging Config Database into the Config Database. To do so, the application calls `Merge()` on the Config Database handle specifying the Staging Config Database. Finally the application calls `Destroy()` on the Staging Config Database because it is no longer needed.

8.2.3.2 Shutting down Config Database

Before shutting down a Config Database, the application typically performs operations on other RFA packages. Optionally, the application may perform operations on the Config Database to query RFA configuration information as described in section 8.2.4. When the application no longer uses any RFA packages (e.g., application is about to exit or call `Context.Uninitialize()`), it should perform the **Shutdown Configuration Database** activity as in Figure 68. The application should shut down the Config Database via the `Release()` method as follows:

```
// --- Shutdown Config Database
configDatabase.Release();
```

Example 108: Shutdown Configuration

Warning! After calling the `Release()` method, the application MUST NOT use the `ConfigDatabase` handle. The `Release()` makes this handle invalid. The application MUST make sure that it releases each acquired `ConfigDatabase` before it calls `Context.Uninitialize()`.

8.2.3.3 Populating an Application Specific Config Database

Populating a specific application Config Database is almost no different than that of populating the Config Database identified by the Context. The only difference is that when calling the static `Acquire()` method, it specifies a name different than the name defined by the Context. The name specified by the Context is "RFA."

8.2.4 Query Configuration Information

Once having populated a Config Database, an application may query configuration information by retrieving Config Nodes, iterating over Config Trees or iterating across Config Softlinks.

8.2.4.1 Retrieving RFA Configuration Nodes

Before querying configuration information from a Config Database, an application should obtain a handle to the Root Config Tree. As indicated above, the Root Config Tree is the uppermost Config Tree in a Config Database. An example is as follows:

```
// --- Retrieve Root Config Tree ---
ConfigTree rootConfigTree = configDatabase.ConfigTree;
```

Example 109: Retrieve Root Config Tree

Once having retrieved the Root Config Tree handle (`rootConfigTree`), an application may retrieve specific Config Nodes. The next example depicts an application using the Root Config Tree handle to obtain an arbitrary Config Tree handle and subsequently a Config Node. In this case, the application knows the Config Node is of type Config String.

```
// --- Retrieve Config Node with Known Type ---
// Retrieve a specific session Config Node
ConfigTree configTree = (ConfigTree)(rootConfigTree.GetNode(new RFA_String("Default\\Sessions\\Session1\\")));

// Retrieve a specific session parameter as a known Config String type
System.Console.WriteLine(configTree.GetChildAsString(new RFA_String("ConnectionList")));
```

Example 110: Retrieve Config Node with Known Type

In this example, the application uses the Root Config Tree handle (`rootConfigTree`) to obtain a handle to the **Session1** Config Node. To obtain this handle, the application uses the `GetNode()` method and specifies the path “Default\\Sessions\\Session1\\.” Knowing this Config Node is a Config Tree; the application casts the handle to a Config Tree handle.

Next the application uses the Config Tree handle to retrieve the value of a specific configuration parameter from the **Session1** Config Node. This configuration parameter is a Config Node entitled “**ConnectionList**.” Knowing this Config Node is a Config String, the application retrieves the parameter using the `GetChildAsString()` method. This method accepts one argument, which is the parameter name (i.e., “**ConnectionList**”).

The next example depicts an application retrieving the same configuration parameter, but does not assume the application knows the specific parameter type is a Config String.

```
// --- Retrieve Config Node with Unknown Type ---
// Retrieve a specific session parameter as a Config Node
ConfigNode configNode = rootConfigTree.GetNode(new RFA_String("Default\\Sessions\\Session1\\ConnectionList\\"));

// Switch on node type and output value
switch (configNode.Type)
{
    case ConfigDatabaseNodeTypeEnum.stringValueNode:
    {
        ConfigString configParam = (ConfigString)(configNode);
        System.Console.WriteLine(configParam.Value.ToString());
        break;
    }
    case ConfigDatabaseNodeTypeEnum.longValueNode:
    {
        ConfigLong configParam = (ConfigLong)(configNode);
        System.Console.WriteLine(configParam.Value);
        break;
    }
    default:
    {
        System.Console.WriteLine("Unexpected Type");
        break;
    }
}
```

Example 111: Retrieve Config Node with Unknown Type

In this example, the application uses the handle to the Root Config Tree (`rootConfigTree`) to obtain a handle to a specific Config Node, which is a configuration parameter. The application uses the `GetNode()` method to retrieve the Config Node handle. The application specifies the path “Default\\Sessions\\Session1\\ConnectionList\\” to obtain this Config Node handle.

Next the application uses the Config Node handle and switches on the type via the `Type` property. Each case statement processes a different type by outputting the parameter’s value. For brevity, this example only shows two different Config Node types. An application may use the same technique to identify different types such as Config Bool, Config Tree and Config Softlink.

8.2.4.2 Iterating RFA Configuration Nodes

An application may iterate over Config Trees as shown in the following example:

```
// --- Iterate over Config Tree ---
{
    // Retrieve a node handle to Sessions and cast to a ConfigTree handle
    ConfigTree tree = (ConfigTree)(rootConfigTree.GetNode(new RFA_String("Default\\Sessions\\")));

    // Create an Iterator from the Sessions node.
    ConfigNodeIterator it = tree.CreateIterator();
    if (it == null)
    {
        System.Console.WriteLine("Cannot create ConfigNodeIterator.");
        Environment.Exit(-1);
    }
}
```

```
// Iterate over the Sessions node and extract full name of each Session.
ConfigNode node = null;
for (it.Start(); !(it.Off()); it.Forth())
{
    node = it.Value;
    System.Console.WriteLine(string.Format("FullName: {0}", node.FullName.ToString()));
}

// Destroy iterator. It is no longer needed.
it.Destroy();
}
```

Example 112: Iterate Over Config Tree

In this example, the application uses the handle to the Root Config Tree ([rootConfigTree](#)) to obtain a handle to a specific Config Node, which is a Config Tree. The application uses the [GetNode\(\)](#) method to retrieve the Config Node handle. The application specifies the path “**Default\\Sessions**” to obtain this Config Node handle. Knowing this Config Node is a Config Tree; the application casts it to a Config Tree handle.

Next the application creates an iterator on this Config Tree via the [CreateIterator\(\)](#) method. The application then iterates over all children of the Config Tree node outputting the Full Name via the [FullName](#) property. The Full Name is merely the entire path of any particular Config Node. Finally the application destroys the iterator via the [Destroy\(\)](#) method.

8.2.4.3 Iterating RFA Configuration Nodes by using Foreach Loop

An application can use [foreach](#) loop to iterate over Config Trees as shown in the following example:

```
// --- Iterate over Config Tree ---
{
    // Retrieve a node handle to Sessions and cast to a ConfigTree handle
    ConfigTree tree = (ConfigTree)(rootConfigTree.GetNode(new RFA_String("Default\\Sessions\\")));

    // Iterate over the Sessions node and extract full name of each Session.
    foreach (ConfigNode node in tree)
    {
        System.Console.WriteLine(string.Format("FullName: {0}", node.FullName.ToString()));
    }
}
```

Example 113: Iterate Over Config Tree by using foreach loop

Moreover, [ConfigTree](#) provides the [GetEnumerator\(\)](#) method to get an enumerator for iterating through it by using the methods of [IEnumerator](#).

NOTE: Using [foreach](#) loop can degrade application performance as an enumerator is instantiated at the beginning and released at the end of the loop.

8.3 Configuration Package: Putting it All Together

This section combines code snippets described throughout section 8.2 to realize complete examples.

8.3.1 Populating and Querying a Config Database

This example populates and queries the Config Database instance used by alternative RFA packages. The application loads a Staging Config Database from the Windows Registry and merges it into the Config Database. Next, the application retrieves a known Config Node type, retrieves an unknown Config Node type and finally performs iteration over a Config Tree.

```
using System;
using Reuters.RFA.Config;
using Reuters.RFA.Common;

namespace StarterConfiguration
{
    class Program
    {
        static void Main(string[] args)
        {
            // --- Initialize RFA Config Database ---
            // Instantiate and load RFA Staging Config Database
            try
            {
                Reuters.RFA.Config.StagingConfigDatabase stagingConfigDatabase =
StagingConfigDatabase.Create();
                if (stagingConfigDatabase == null)
                {
                    System.Console.WriteLine("Cannot create StagingConfigDatabase.");
                    Environment.Exit(-1);
                }

                bool loadRetVal = stagingConfigDatabase.Load(ConfigRepositoryTypeEnum.windowsRegistry, new
RFA_String("HKEY_LOCAL_MACHINE\\SOFTWARE\\Reuters\\RFA\\Default"));
                if (!loadRetVal)
                {
                    System.Console.WriteLine("Cannot load configuration depository by StagingConfigDatabase.");
                    Environment.Exit(-1);
                }

                // Acquire Application ConfigDatabase
                Reuters.RFA.Config.ConfigDatabase configDatabase = ConfigDatabase.Acquire(new
RFA_String("RFA"));
                if (configDatabase == null)
                {
                    System.Console.WriteLine("Cannot create ConfigDatabase.");
                    Environment.Exit(-1);
                }

                //Merge configuration from the StagingConfigDatabase into ConfigDatabase with "Default"
                namespace
                configDatabase.Merge(stagingConfigDatabase);

                // Shutdown Staging Config Database. It is no longer needed.
                stagingConfigDatabase.Destroy();
            }
        }
    }
}
```

```

// --- Retrieve Root Config Tree ---
ConfigTree rootConfigTree = configDatabase.ConfigTree;

// --- Retrieve Config Node with Known Type ---
// Retrieve a specific session Config Node
ConfigTree configTree = (ConfigTree)(rootConfigTree.GetNode(new
RFA_String("Default\\Sessions\\Session1\\")));

// Retrieve a specific session parameter as a known Config String type
System.Console.WriteLine(configTree.GetChildAsString(new RFA_String("ConnectionList")));

// --- Retrieve Config Node with Unknown Type ---
// Retrieve a specific session parameter as a Config Node
ConfigNode configNode = rootConfigTree.GetNode(new
RFA_String("Default\\Sessions\\Session1\\ConnectionList\\"));

// Switch on node type and output value
switch (configNode.Type)
{
    case ConfigDatabaseNodeTypeEnum.stringValueNode:
    {
        ConfigString configParam = (ConfigString)(configNode);
        System.Console.WriteLine(configParam.Value.ToString());
        break;
    }
    case ConfigDatabaseNodeTypeEnum.longValueNode:
    {
        ConfigLong configParam = (ConfigLong)(configNode);
        System.Console.WriteLine(configParam.Value);
        break;
    }
    default:
    {
        System.Console.WriteLine("Unexpected Type");
        break;
    }
}

// --- Iterate over Config Tree ---
{
    // Retrieve a node handle to Sessions and cast to a ConfigTree handle
    ConfigTree tree = (ConfigTree)(rootConfigTree.GetNode(new
RFA_String("Default\\Sessions\\")));

    // Create an Iterator from the Sessions node.
    ConfigNodeIterator it = tree.CreateIterator();
    if (it == null)
    {
        System.Console.WriteLine("Cannot create ConfigNodeIterator.");
        Environment.Exit(-1);
    }

    // Iterate over the Sessions node and extract full name of each Session.
    ConfigNode node = null;
    for (it.Start(); !(it.Off()); it.Forth())
    {
        node = it.Value;
        System.Console.WriteLine(string.Format("FullName: {0}", node.FullName.ToString()));
    }
}

```

```

        // Destory iterator. It is no longer needed.
        it.Destroy();
    }

    // --- Shutdown Config Database
    configDatabase.Release();
}
catch (Reuters.RFA.Common.InvalidUsageException e)
{
    System.Console.WriteLine(e.Status.StatusText.ToString());
}
}
}
}
}

```

Example 114: Populating and Querying a Config Database

8.3.2 Merging Multiple Namespaces from a Windows Registry Configuration

Following the Figure 69, this example shows how to merge the multiple namespaces to the Config Database. Following the StarterConsumer example, there are two configuration sets used. First is set of configuration for RFA API used to create the communication channel and the second is the set of configuration for application level.

Two Staging Config Databases have to be created for loading those two configuration from different locations on the Windows Registry repository with using different namespace. First, creating Staging Config Database to load the configuration parameters from "HKEY_LOCAL_MACHINE\SOFTWARE\Reuters\RFA\Default" and merge to the Config Database using "Default" namespace. Next, creating the second Staging Config Database to load the configuration parameters from "HKEY_LOCAL_MACHINE\SOFTWARE\Reuters\RFA\StarterConsumer" and merge to the Config Database using "StarterConsumer" namespace. Finally, create the iterator to retrieve information from "StarterConsumer" node on the console.

```

using System;
using Reuters.RFA.Config;
using Reuters.RFA.Common;

namespace StarterConfiguration
{
    class Program
    {
        static void Main(string[] args)
        {
            // --- Initialize RFA Config Database ---
            // Instantiate and load RFA Staging Config Database
            try
            {
                Reuters.RFA.Config.StagingConfigDatabase stagingConfigDatabase =
                    StagingConfigDatabase.Create();
                if (stagingConfigDatabase == null)
                {
                    System.Console.WriteLine("Cannot create StagingConfigDatabase.");
                    Environment.Exit(-1);
                }

                bool loadRetVal = stagingConfigDatabase.Load(ConfigRepositoryTypeEnum.windowsRegistry, new
                    RFA_String("HKEY_LOCAL_MACHINE\\SOFTWARE\\Reuters\\RFA\\Default"));
                if (!loadRetVal)
                {

```

```

        System.Console.WriteLine("Cannot load configuration depository by StagingConfigDatabase.");
        Environment.Exit(-1);
    }

    // Acquire Application ConfigDatabase
    Reuters.RFA.Config.ConfigDatabase configDatabase = ConfigDatabase.Acquire(new
        RFA_String("RFA"));
    if (configDatabase == null)
    {
        System.Console.WriteLine("Cannot create ConfigDatabase.");
        Environment.Exit(-1);
    }

    //Merge configuration from the StagingConfigDatabase into ConfigDatabase with "Default"
    namespace
    configDatabase.Merge(stagingConfigDatabase);

    // Shutdown Staging Config Database. It is no longer needed.
    stagingConfigDatabase.Destroy();

    // Create new StagingConfigurationDatabase for Namespace StarterConsumer
    stagingConfigDatabase = StagingConfigDatabase.Create();
    if (stagingConfigDatabase == null)
    {
        System.Console.WriteLine("Cannot create StagingConfigDatabase for StarterConsumer.");
        Environment.Exit(-1);
    }

    bool retVal = stagingConfigDatabase.Load(ConfigRepositoryTypeEnum.windowsRegistry, new
        RFA_String("HKEY_LOCAL_MACHINE\\SOFTWARE\\Reuters\\RFA\\StarterConsumer"));
    if (!retVal)
    {
        System.Console.WriteLine("Cannot load StarterConsumer configuration depository by
            StagingConfigDatabase.");
        Environment.Exit(-1);
    }

    //Merge configuration from the StagingConfigDatabase into ConfigDatabase with "StarterConsumer"
    namespace
    configDatabase.Merge(stagingConfigDatabase, new RFA_String("StarterConsumer"));

    // --- Retrieve Root Config Tree ---
    ConfigTree rootConfigTree = configDatabase.ConfigTree;

    // --- Iterate over Config Tree of StarterConsumer Namespace ---
    {
        // Retrieve a node handle to StarterConsumer namespace and cast to a ConfigTree handle
        ConfigTree tree = (ConfigTree)(rootConfigTree.GetNode(new RFA_String("StarterConsumer")));

        // Create an Iterator from the StarterConsumer node.
        ConfigNodeIterator it = tree.CreateIterator();
        if (it == null)
        {
            System.Console.WriteLine("Cannot create ConfigNodeIterator.");
            Environment.Exit(-1);
        }

        // Iterate over the Sessions node and extract full name of each Session.
        ConfigNode node = null;
        for (it.Start(); !(it.Off()); it.Forth())

```



```
        {
            node = it.Value;
            System.Console.WriteLine(string.Format("FullName: {0}", node.FullName.ToString()));
        }

        // Destory iterator. It is no longer needed.
        it.Destroy();
    }

    // --- Shutdown Config Database
    configDatabase.Release();
}
catch (Reuters.RFA.Common.InvalidUsageException e)
{
    System.Console.WriteLine(e.Status.StatusText.ToString());
}
}
}
```

Example 115: Merging Multiple Namespaces from a Windows Registry Configuration

Chapter 9 SessionLayer Package

9.1 SessionLayer Package Concepts

The **SessionLayer Package** implements the functionality needed by applications to receive and send data. The Session Layer hides the differences between RTDS components (e.g., an ADS or RDF Direct) so that application developers can concentrate on their application's functionality instead of coding to differences among these systems.

9.1.1 General Session Layer Concepts

The concepts related to the Session Layer Package are listed in the table below.

CONCEPT	DESCRIPTION
Connection	Encapsulates connectivity to a back-end system such as the ADS, ADH, RDF Direct, and other legacy components.
Event Source	An object that typically reacts to an Interest Specification by sending one or more events. For more details see section 5.2.1, Event Distribution Model.
Session	An Event Source Factory that encapsulates one or more Connections that share the same resources, such as a thread context.
Consume (subscribe)	The process of using an Event Stream to obtain Events containing information about a particular entity (e.g., item interest). A consuming application typically registers interest to receive Market Data refreshes and updates.
Provide (publish)	The process of submitting information to the RTDS about a particular entity. A providing application typically provides refreshes, updates and status messages.
Post (contribute)	The process of submitting information to an RTDS component about a particular entity (e.g., Market Information Items) ²⁷ . A posting application typically posts refreshes, updates or status messages to a head-end system.
User Validation Mechanism	The application's method of identifying the user. Applications are required to provide the user credentials in order to receive permissioned data.
Publisher Identification Mechanism	The application's method of identifying the publisher of data. Provider applications can provide Publisher Principal Identity on refreshes, updates, and status messages, while consumer applications provide Publisher Principal Identity on Post Message.
Concrete service	Named grouping of Market Items provided from a single back-end system.
Service group	A combination of multiple services into what appears to the user to be a single service. It has its own name and allows for item routing and recovery. Service Groups may be supplied from one or multiple providers.
Dictionary	A definition of type or formatting information that an application can use for direction on how to encode or decode specific pieces of market data.

Table 67: General Session Layer Concepts

²⁷ Contributions are Market Information-specific but are described here for consistency with publishing.

RFA applications can both request information and make information available for use by a back-end system, head-end system, or another application.

A Consumer application can receive information it is interested in by opening an Event Stream. (See section 5.2.1.) While the Event Stream is open, the application can receive Events related to the particular entity it desires information about. For example, an application may request market information for **TRI.N** and receive events related to it.

When a Provider application makes information available to be used by a back-end system or a different application, it does not use an Event Stream but submits the information instead.

The user may need to be validated by RFA before accessing the data. The application does so by sending a login request to an upstream component and receives login permission in return.

The application may wish to obtain status related to Connections. For example, an application may wish to obtain status as to whether a connection is up or down. To obtain this status, an application opens an Event Stream specifying status for connections. The application then receives Events relating to the specified status.

Connections encapsulate connectivity to a back-end system, such as an ADS, RDF Direct, etc. With the exception of status, RFA does not expose connections. However, RFA exposes interfaces to the Session, which consists of one or more connections. All connections associated with a session appear to the application as a single connection.

The Session interface is an Event Source Factory. As the term factory implies, an application uses the Session interface to create Event Sources. These Event Sources provide access to the back-end systems. An Event Source provides a specific capability, such as the ability to request market information.

The Session Layer Package supports QoS. As defined in the Common Package, QoS specifies a method of classifying Services provided by market data delivery infrastructure. When making market information available, an application may specify the corresponding QoS.

When requesting market information, an application may specify the desired QoS and indicate how the QoS may change over time. The application specifies the desired QoS when opening the Event Stream, but receives the actual QoS from Events associated with the Event Stream.

Internally, the Session performs the core business logic of RFA and operates in a single thread of control. Using 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.

An application can also create multiple sessions. Using multiple Session instances allows segmenting information—i.e., to separate different types of market information. Multiple Sessions can share a single Connection, a feature known as Connection Sharing. An application may use any reasonable number of Sessions to create one or more Event Sources of the same or different types. An application that uses multiple components may choose to use a single Session instance or multiple Session instances.

9.1.1.1 Consumers and Providers

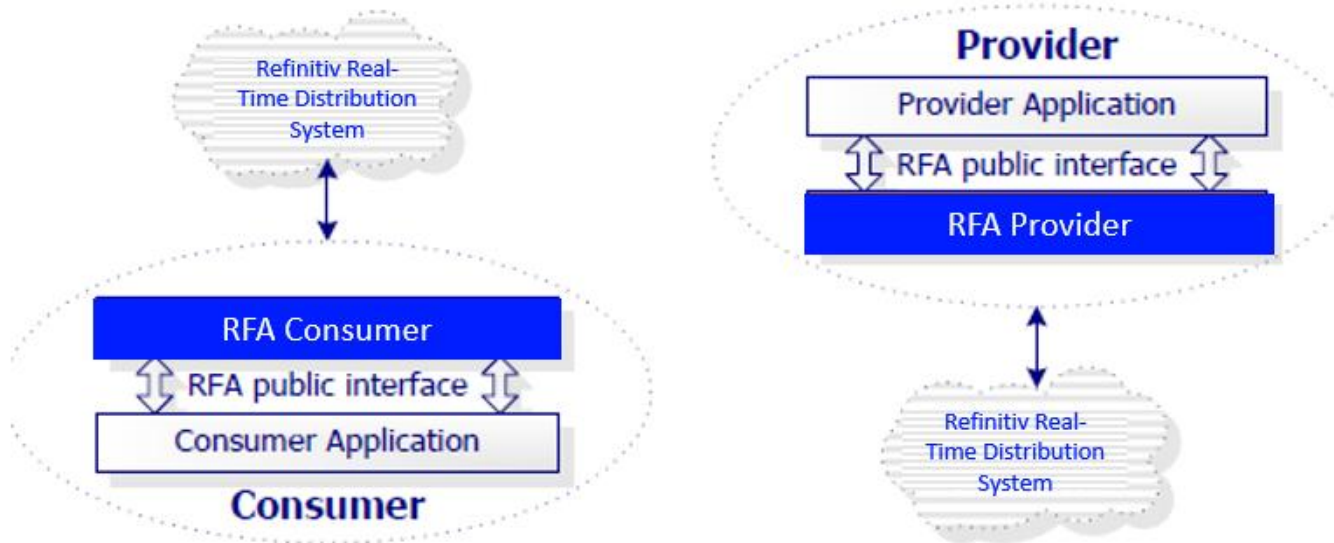


Figure 70: Consumer and Provider Applications Using a Session

An OMM Consumer is an Event Source that consumes services. The Session Layer implements an OMM Consumer that can be used to make Interest Specifications such as an [OMMItemIntSpec](#) (i.e., OMM Item Interest Specification). One or more OMM Consumers can be 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 such as the concepts of an Item Update or an Item State.

An OMM Provider is an Event Source that is a provider of services. The Session Layer implements an OMM Provider that can be used to create Interest Specifications such as an [OMMClientSessionIntSpec](#). Similar to a Consumer, one or more OMM Providers can be associated with a specific Session and uses the back-end systems. OMM Providers also understand market information.

The OMM Consumer and OMM Provider are specialized versions of an Event Source. They are entities that react to an Interest Specification by sending one or more asynchronous responses.

Sessions, Events, and specific Event Sources are represented in the Session Layer as separate interfaces. The connection is only visible as a configurable entity; it does not have an interface of its own.

9.1.1.2 Application Threading Model

One of the fundamental requirements for the Session Layer is to provide a thread-safe and thread-aware implementation.

The Session Layer meets this requirement by following the approach to threading defined by the Event Distribution mechanism (see Section 5.2.1, Event Distribution Model for details) and leaving decisions such as the number of application threads and their usage up to application developers.

With very few exceptions, the Session Layer doesn't call application code in the context of a Session Layer thread. Instead, it's an application's responsibility to dispatch Events generated by the Session Layer and delivered to the application using the Event Distribution mechanism. By dispatching Events in its own context the application maintains full control of its threads. This feature is known as Unrestricted Time Quantum.

The application can also maintain callback affinity—i.e., it can make sure that all callbacks are executed in the same thread context that was used to create specific application-defined objects. The application maintains callback affinity by calling [Dispatch\(\)](#) from the same thread context that was used to create the application-defined objects.

The Session Layer delivers events by posting them to an Event Queue specified by the application while making the request. The application initiates event processing by dispatching them from the Event Queue. When an Event is

dispatched, the Session Layer calls the Event Processing Client associated with that particular Event, and passes the Event to the Client.

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

The ability to freely use multiple threads for making requests and dispatching events is what makes the Session Layer thread aware, while the ability to make calls on Session Layer interfaces from multiple application threads is what makes the Session Layer thread safe. All Session Layer interfaces are thread-safe at the class level²⁸. Some Session Layer interfaces are thread-safe at the object level²⁹. For more information about which interfaces are class-level safe and object-level safe, see the *RFA Reference Manual .NET Edition*.

For a simplified picture of the Session Layer threading model, refer to Figure 28 in section 5.1.1.8, Example of the Event Distribution Model.

9.1.1.3 Low Latency and High Throughput

The OMM Consumer offers performant options of low-latency and high-throughput response for images and updates at the expense of affecting the following features:

- Processor scalability³⁰
- Unrestricted Time Quantum
- Connection Sharing
- Controlled dispatching (see Section 5.1.1.7)

The OMM Consumer offers two performance options which can be used individually or together, optimizing RFA for either throughput or latency. The first is a configuration parameter, **threadModel**, that can be set to **Single** or **Dual**, controlling the number of threads in the API at runtime. The second is a pair of event processing models, the Client Model and the Callback Model, which are controllable by specifying an event queue, or not when calling the [CreateOMMConsumer\(\)](#) interface.

These options are covered briefly in the sections below but are discussed in more detail in section 14.3, Threading Model and OMM Provider performance is discussed in section 14.5, Configuring RFA Provider for Performance.

9.1.1.3.1 threadModel

By default the **threadModel**³¹ configuration parameter is set to **Dual**. When in this mode RFA optimizes the adapter thread and its associated queue for maximizing Response Message throughput in RFA. RFA also runs in (at least) three threads: one for the application, one for the Session Layer, and one for the Adapter.

If the parameter is set to **Single**, RFA optimizes the thread model to minimize latency. RFA bypasses the internal queue between the Session Layer and the Adapter, and RSSL messages pass directly from the Adapter to the client's [ProcessEvent\(\)](#), if there is no event queue specified on the [RegisterClient\(\)](#) call (hence the Callback Model is used.)

For more information on configuring Consumers and Providers for performance see Sections 14.4 and 14.5.

9.1.1.3.2 Client and Callback Models

Use of the Callback Model differs from the Client Model in that the Session Layer thread calls the application to deliver the events instead of the application relinquishing control of a thread for [Dispatch\(\)](#). The application invokes this feature by

²⁸ 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.

²⁹ 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.

³⁰ If Horizontal Scaling is not enabled

³¹ Prior to RFA 7.4, ThreadModel was known as OMMPerfMode, which could be set to Latency (single threaded) or throughput (dual threaded). In RFA 7.4 this configuration parameter is still allowed although it is now deprecated.

specifying **NULL** for an [EventQueue](#) when registering for interest. The application may receive callbacks any time after registering for interest until the event stream is closed. Any pending callback will still occur following the closing of an event stream.

Using the Callback Model affects the four features mentioned above in the following ways:

- **Unrestricted Time Quantum:** Use of the Callback Model requires the application not to hold onto the API thread for a significant period of time so that the API can efficiently handle its Session Layer and Adapter. High-performance applications are expected to minimize or offload CPU-intensive activities.
- **Controlled Dispatching:** Since the application does not relinquish control of a thread, the application does not call [Dispatch\(\)](#) as there is no benefit when using the Callback Model. Since all callbacks are asynchronous, using a notification client is unnecessary and unsupported.
- **Processor Scalability:** The Callback Model is less scalable than the Client Model. Processor scalability is deferred to the application and its configuration when the Client model is used.
- **Connection Sharing:** Use of the Callback Model does not allow connection sharing.

NOTE: The application may choose to use this feature with some event streams and not other event streams. The application may use completion events to determine the final event sent to the application.

For more information, see section 14.3, Threading Model.

9.1.1.4 Horizontal Scaling

Horizontal Scaling enables RFA applications to use multiple instances of the Consumer and Provider on multi-core processors to dynamically scale the number of Session instances that applications use. Since each instance of a horizontally-scaled adapter processes messages on its own connection (independently of other adapter instances), applications can utilize this feature to dramatically increase Response Message and Request Message throughput. This feature is available for OMM Consumer as well as OMM Provider (interactive and non-interactive) applications.

For more details, refer to section 14.4.6, Horizontal Scaling (Client Model) and the *RFA Configuration Guide .NET Edition*.

9.1.1.5 Pending Request Queue

Item requests made by an application (i.e., outbound messages) are placed in a pending request queue provided by the Session Layer. The Session Layer does not send these messages until the desired back-end systems have initialized.

Combined with the capability to initiate the dispatching of events, pending request queuing allows applications to immediately send messages to the Session Layer after initialization. Applications do not need to be concerned whether the Session Layer, including connections to back-end systems, has finished initializing. The pending request queuing time-out period is configurable on a per-Session basis.

9.1.1.6 Session Sharing and Multiple Event Sources

Session sharing allows applications to share back-end system resources, such as connections, among components within a single application. Multiple threads within a single application can share a single Session if they specify the same name when acquiring a Session.

If the application is not concerned with back-end system resources, or wants to minimize thread synchronization between multiple threads, then multiple Session instances can be used. Each Session then has a unique configuration defined within the configuration database.

When multiple application components share a single Session, each component should create and use a separate Event Source (e.g., OMM Consumer, OMM Provider. Multiple threads in the same application that share a single Session would create separate Consumers that would manage the consuming items for each component.

Multiple application components that share a single Session should also use separate Event Queues for dispatching. This allows each component to perform its own dispatching without impacting other components.

9.1.1.7 Event Distribution with Multiple Threads

While RFA supports event distribution using multiple threads, it does not guarantee the order of the operations when they are used. For example; if two threads provide to the same item at the same time, the API does not guarantee the ordering to the underlying API.

RFA also support relinquishing interest in a client from a thread other than the one that calls dispatch. However, applications that unregister a client from a thread other than the dispatching thread should be aware that the dispatching thread may be inside an event handler for the same interest that is being canceled. This condition should be handled by waiting for an Event with `IsEventStreamClosed` set to `true` for confirmation that the interest has been canceled within the API. The type of that Event should not matter; in most cases it will be a Completion Event.

9.1.2 OMM Concepts

OMM provides capabilities for an application to consume or provide information. The Session Layer introduces several concepts related to OMM which are defined in the following table.

CONCEPT	DESCRIPTION
Consumer	A service-consuming application that sends requests and receives responses.
Provider	A service-providing application that receives requests sends responses.
Hybrid	An application that acts as both a Consumer and Provider, receiving both requests and responses, forwarding them, and tuning the contents when desired.
OMM Consumer	An Event Source capable of requesting information which: <ul style="list-style-type: none"> • Sends requests and Posts • Receives responses and Acks
OMM Provider	An Event Source capable of making information available which: <ul style="list-style-type: none"> • Receives requests and Posts • Sends responses and Acks
OMM Item Event	An event with information about the item
OMM Item Interest Specification	Specification for interest in item events
OMM Connection Interest Specification	Specification for interest in connection events
Message Model Type	The type identifying the specific Message Model

Table 68: OMM Concepts

The Session Layer allows an application to access OMM data from an RDF Direct 1.x or RTDS.

An application can either consume or provide market information. A consumer application is first required to log in by passing user credentials in order to gain access to permissioned market information. It may be necessary for the application to obtain a list of available back-end services—e.g., data dictionary, directory of services, etc. In general, the application is a consumer, which consumes services, and the back-end system is a provider, which provides services.

Information is transmitted between the consumer application and provider application via messages. Messages are usually Request Message and Response Message, but can also be Generic Message, Post Message, and Ack Message.

Consumer applications can send request and Post Message and receive Response Message and Ack Message. Provider applications can receive Request Message and Post Message and send Response Message and Ack Message. Hybrid applications receive Request Message from a consumer application and forward the same Request Message to another provider application. Similarly, a hybrid application receives Response Message from a provider and forward the same Response Message to the consumer application.

A consumer application typically uses a Request Message to login, obtain service directory, obtain an initial image, and/or obtain interest after the initial image. A consumer application may also use a Request Message for changing (reissuing) the specification of open interest. Typically a provider application uses a Response Message to provide market data information, changes in market data information and status information.

Each message belongs to a particular message model type (e.g., [MMT_LOGIN](#), [MMT_MARKET_PRICE](#)) defined in the RDM class. The request message is encapsulated in an OMM Item Interest Specification and the response message is encapsulated in an OMM Item Event.

The application specifies the request attributes for a message. The request attributes vary based on the message model type—e.g., the username attribute for a Request Message that is required for the Login Message Model is not required for a Request Message belonging to the Directory Message Model. For a detailed list of which messages require which attributes, refer to the *RFA RDM Usage Guide .NET Edition*.

The application can request market information by setting the appropriate message model type, specifying the request attributes, item name and service name. The item name is a string and is a valid RIC (e.g., **IBM.N**), and its length should not be longer than 255 characters. The service name is a string representing the service (e.g., **DIRECT_FEED**) from which the item information is obtained.

The connection status and login status are delivered to the application. The connection status indicates if the connection is up or down. The login status provides the stream state, data state, etc. The service status for the services is provided by the Directory Message Model.

The Session Layer Package supports QoS. The application may extract the QoS from Response Messages encapsulated in Events associated to an Event Stream.

Usage of the event queue is configurable when consuming data. It can be skipped to achieve low latency. For additional details on improving performance see section 14.3, Threading Model.

Every Event produced by the Event Sources is associated with an Event Stream. The Session Layer provides the Event Sources [OMMConsumer](#) and [OMMProvider](#). The [OMMConsumer](#) identifies the Event Stream via a Handle, while the [OMMProvider](#) identifies the Event Stream via a Request Token.

9.1.2.1 OMM Item State

The Session Layer may inform applications of Response Status for reasons such as a change in data health or an item being closed. If the Session Layer cannot satisfy an interest specification it always sends at least one OMM Item Event containing a Response Message with a Response Status that has stream state of [Closed](#).

The Response Status conveys item status through separation of the stream state (e.g., [Open](#), [Closed](#)) and data state (e.g., [OK](#), [Suspect](#)).

Response Status is applicable to both the OMM Provider, which sends it, and the OMM Consumer, which receives it.

For more information on Response Status, see section 7.1.14, Response Status (Stream States and Data States) and Figure 60.

9.1.3 Events and Cmds

Events and **Cmd** are wrapper objects used for Request Message, Response Message, Generic Messages, Post Message and Ack Message. Events represent inbound messages ([RespMsg](#), [GenericMsg](#), [AckMsg](#), [ReqMsg](#), [PostMsg](#)) from the network to the consumer, the consumer client session and the connection. Cmds are the Wrapper classes that contain Generic Message ([GenericMsg](#)), Post Message ([PostMsg](#)), Ack Message ([AckMsg](#)) or Response Message ([RespMsg](#)) that need to be sent to the network.

The application needs to register Interest Specification to receive Event from RFA. The following table depicts the event-related interface.

EVENT	INTEREST SPECIFICATION	CMD	APPLICATION
OMMConnectionEvent	OMMConnectionIntSpec,		Consumer
	OMMListenerConnectionIntSpec		Provider
OMMCmdErrorEvent	OMMCmdErrorIntSpec		Consumer and Provider
OMMItemEvent	OMMItemIntSpec	OMMHandleItemCmd	Consumer
		OMMItemCmd	Non-Interactive Provider
OMMActiveClientSessionEvent	OMMClientSessionListenerIntSpec	OMMClientSessionCmd	Interactive Provider
OMMInactiveClientSessionEvent	OMMClientSessionListenerIntSpec		Interactive Provider
OMMSolicitedItemEvent	OMMClientSessionIntSpec	OMMSolicitedItemCmd	Interactive Provider
OMMConnectionStatsEvent	OMMConnectionStatsIntSpec		Consumer, Provider, and Interactive Provider

Table 69: Event and Cmd

9.2 SessionLayer Package Usage

9.2.1 Session

Applications that use the Session Layer are known as Consumers, Providers, or Hybrids. Consumers consume services and Providers provide services. Consumers send a request and receive a response, while Providers receive the same request and send the same response. Hybrid applications receive both requests and responses. Hybrid applications then forward requests and responses, tuning the contents when desired.

All application types must correctly configure, initialize (acquire), use, and cleanup (release) the Session Layer in a manner consistent with its interfaces.

9.2.1.1 Session Initialization

Typically, one of the application's first activities is **Initialize Session**. To initialize a Session, the application calls the static **Acquire()** method as follows:

```
session = Session.Acquire(cfgVariables.SessionName);
```

Example 116: Acquire Session

The static **Acquire()** method requires at least one parameter, which is the Session name. In this case, the application specifies the Session name from the configuration. The Session name both references configuration information and provides identification for purposes such as logging. The static **Acquire()** method returns a Session object.

RFA may share the instance returned by **Acquire()** among other application components. This is known as **Instance Sharing**. RFA bases a unique instance of a Session on its name. Benefits of Instance Sharing include alleviating the need for the application to pass around a Session reference among multiple application components and the ability to minimize resource consumption.

9.2.1.2 Session Cleanup

After initializing a Session, the application typically next performs operations on Session Layer interfaces. These operations commonly involve access to Market Information through one or more of the various Event Sources (namely OMM Consumer, OMM Provider). Subsequent sections describe these Event Sources.

After no longer using the Session Layer interfaces, the application should shutdown and cleanup the session via the **Release()** method as follows:

```
if (session != null)
{
    session.Release();
    session = null;
}
```

Example 117: Release Session

NOTE: 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.

9.2.2 Events and Cmds

9.2.2.1 Event Registration

To register for events, the application need to call `RegisterClient()` method by passing the one of the Interest Specifications.

9.2.2.1.1 Registering OMMConnectionEvent

An application may optionally register to receive connection-related Events. Because RFA automatically recovers connections and their associated Event Streams, the main value to receiving this Event type is so that the application can notify the user of the condition through its own user interface.

To receive conection-related Events, the application must register for the `OMMConnectionEvent` by calling the `RegisterClient()` method using the `OMMConnectionIntSpec`.

```
OMMConnectionIntSpec connectionIntSpec = new OMMConnectionIntSpec();
ommConnIntSpecHandle = ommConsumer.RegisterClient(eventQueue, connectionIntSpec, this, null);
```

Example 118: Register OMMConnectionEvent

9.2.2.1.2 Registering OMMCmdErrorEvent

Before submitting a Generic Message or a Post Message, the application can register for the error event. The error event is returned to an application whenever a Generic Message or a Post Message is submitted but then fails somewhere between the `Submit()` call and before the message is written to the transport.

To receive Generic Message or Post Message submission failure events, the application must register for the `OMMCmdErrorEvent` by calling the `RegisterClient()` method using the `OMMErrorIntSpec`.

9.2.2.1.3 Registering OMMItemEvent

The `OMMItemEvent` is a wrapper around a `RespMsg`, `GenericMsg` or `AckMsg` that can obtained via `Msg` property. The application must register for the `OMMItemEvent` by calling the `RegisterClient()` method using the `OMMItemIntSpec` that contained the `ReqMsg`.

9.2.2.1.4 Registering ClientSessionEvent

OMMProvider will not receive any inbound consumer client session connection attempts. The application needs to register with the OMM Provider to open a listening port for consuming client session requests to connect. This is done by calling `RegisterClient()` mthod using the `OMMClientSessionListenerIntSpec`.

Once registered, the application will be able to receive two types of Client Session Events as follows:

- `OMMActiveClientSessionEvent`: Received when a new client session is made from a consumer.
- `OMMInactiveClientSessionEvent`: Received when the consumer client session has been terminated

9.2.2.1.5 Registering OMMSolicitedItemEvent

This is used for accepting a client session request event ([OMMActiveClientSessionEvent](#)). Once the provider application has registered for the [OMMClientSessionIntSpec](#), the application will then be able to receive [OMMSolicitedItemEvent](#) coming from the consuming client session. The client session handle to be accepted is set via the [ClientSessionHandle](#) property on the [OMMClientSessionIntSpec](#).

The application must register for the [OMMSolicitedEvent](#) by calling the [RegisterClient\(\)](#) method using the [OMMClientSessionIntSpec](#).

9.2.2.1.6 Registering OMMConnectionStatsEvent

The application can register with [OMMConnectionStatsIntSpec](#) to receive connection statistics information from [OMMConnectionStatsEvent](#), which periodically notifies from either default or user-defined interval. [OMMConnectionStatsIntSpec](#) support specifying an interest for a specific connection name, multiple connection names, or all connection name.

9.2.3 OMMConsumer

The OMMConsumer is an Event Source that consumes services that contain market data and other information. It supports several concrete Interest Specifications and Event interfaces. Some of its related interfaces contain messages.

The following table depicts OMMConsumer-related interfaces.

INTERFACE	ABSTRACTION	MESSAGE	DESCRIPTION
OMMConsumer	EventSource	None	Event Source that supports consuming services having market information and other information.
OMMItemIntSpec	Interestspec	ReqMsg	Specification for interest in an item.
OMMConnectionIntSpec	Interestspec	None	Specification for interest in connection events.
OMMItemEvent	Event	RespMsg	An event that contains an item.
OMMCmdErrorEvent	Event	None	Event that indicates an error in a command.
OMMConnectionEvent	Event	None	An event that contains connection information.
CompletionEvent	Event	None	Signifies the final event from an event stream.
OMMConnectionStatsIntSpec	Interestspec	None	Specification for interest in a connection statistics event.
OMMConnectionStatsEvent	Event	None	The event contains information on connection statistics (bytes on wire).

Table 70: OMMConsumer Interface

9.2.3.1 OMMConsumer Initialization

To initialize Event Source, the application uses the Session object and calls the [CreateOMMConsumer\(\)](#) method as follows:

```
ommConsumer = session.CreateOMMConsumer(new RFA_String(appName));
```

Example 119: Create Event Source

The `CreateOMMConsumer()` method also accepts an optional second parameter which specifies whether Completion Events should be sent. By default, Completion Events are not generated. The above example shows only the first parameter, which is the name of the Event Source. This name provides identification for purposes such as logging.

9.2.3.2 Event Registration

The application needs to register events with the particular Interest Specification as follows:

- `OMMItemEvent`
- `OMMConnectionEvent`
- `OMMCmdErrorEvent`
- `OMMConnectionStatsEvent`

For more information about event registration, refer to section 9.2.2.1, Event Registration.

9.2.3.3 Submit Message

The application can send Generic Message and Post Message to the provider. Generic Message and Post Message can be sent on the existing stream. After the application establishes a stream, the application must send Generic Message or Post Message by calling `submit()` method using `HandleItemCmd` with the particular message.

The application submit Generic Message to the provider as follows:

```
OMMHandleItemCmd handleCmd = new OMMHandleItemCmd();

handleCmd.Handle = itemHandle;

handleCmd.Msg = genericMsg;

ommConsumer.Submit(handleCmd);
```

Example 120: Submit Generic Message

9.2.3.4 Modify Event Stream

The application can also reissue the Interest Specification by calling `ReissueClient()` with the handle that was returned by event registration. By reissuing the Interest Specification the application can change a Batch request, change a View request, change the stream from streaming to paused, change the item stream priority, or change the login credentials.

The application reissue the Interest Specification as follows:

```
ommConsumer.ReissueClient(handle, interestSpec);
```

Example 121: Reissue the Interest specification

For further detail about changing Batch request, refer to section 13.1.

9.2.3.5 Event Unregistration

When the application no longer has interest in a particular entity, it may relinquish interest via the `UnregisterClient()` method on the Event Source.

The application unregister the event as follows:

```
ommConsumer.UnregisterClient(handle);
```

Example 122: Unregister Event

9.2.3.6 OMMConsumer Cleanup

After no longer using the Event Source, the application should Destroy Event Source via the `Destroy()` method as follows:

```
if (ommConsumer != null)
{
    // ...

    ommConsumer.Destroy();
    ommConsumer = null;
}
```

Example 123: Destroy Event Source

NOTE: 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.

9.2.4 Interactive OMMProvider

The `OMMProvider` is an Event Source that provides services that contain market data and other information. The `OMMProvider` supports several concrete Interest Specifications, Events, and Command interfaces. Some of the `OMMProvider`-related interfaces contain messages.

The following table depicts the interactive provider-related interfaces

INTERFACE	ABSTRACTION	MESSAGE	DESCRIPTION
OMMProvider	EventSource	None	Event Source that supports providing services having market information and other information.
OMMClientSessionListenerIntSpec	InterestSpec	None	Specification of interest in listening for client session requests.
OMMClientSessionIntSpec	InterestSpec	None	Specification of interest to accept a client session and receive item requests.
OMMErrorIntSpec	InterestSpec	None	Specification of interest in command errors.
OMMListenerConnectionIntSpec	InterestSpec	None	Specification of interest in connection events from listening ports.
OMMconnectionStatsIntSpec	InterestSpec	None	Specification for interest in a connection statistics event.
OMMClientSessionCmd	Command	None	Specifies the Client Session to be inactive.
OMMSolicitedItemCmd	Command	RespMsg	Command that contains an item.
OMMActiveClientSessionEvent	Event	None	Event that indicates a request for a Client Session.

INTERFACE	ABSTRACTION	MESSAGE	DESCRIPTION
OMMInactiveClientSessionEvent	Event	None	Event that indicates the Client Session was disconnected.
OMMSolicitedItemEvent	Event	ReqMsg	Event that indicates a request for an item.
OMMCmdErrorEvent	Event	None	Event that indicates an error in a command.
OMMConnectionEvent	Event	None	An event that contains connection information.
Completion Event	Event	None	Signifies the final event from an event stream.
OMMConnectionStatsEvent	Event	None	Contains connection statistics (bytes on the wire).

Table 71: Interactive OMMProvider Interface

9.2.4.1 Interaction Type

The interaction type flags ([InteractionTypeFlag](#)) of the [ReqMsg](#) dictates the request type. Different combinations of these two flags can represent streaming, non-streaming, change of interest or close requests. These flags are not only used to set interest for the initial request, but these flags can be used to change interest in subsequent requests (same request token). See section 9.2.4.1.1, **Invalid Case** for situations that are invalid.

- **InitialImage:** This flag indicates whether or not the request wants a refresh of the image. If this flag is set, the provider must respond by sending a refresh. If it is not set, no refresh image should be sent.
- **InterestAfterRefresh:** This flag indicates whether or not the request wants to receive updates on this stream after the complete refresh has been sent.
- **Pause:** This flag indicates whether the request wants to pause the stream.
 - For more details of Interaction Types, see section 7.1.8.1, Interaction Type.

NOTE: In the case where neither of these flags is set, consider this to be a close request for that request token. Subsequent requests for the same request token can have a changed interaction type.

9.2.4.1.1 Invalid Case

- The request is invalid if the first request does not have [InitialImage](#) set. It is invalid to have only [InterestAfterRefresh](#) set on the initial request from the client session. In the case that this occurs, ignore this request.
- The request is invalid if a request initially had [InterestAfterRefresh](#) set, and then a subsequent request for that same request token is received without either [InterestAfterRefresh](#) set or [Pause](#) set. To handle this case, the provider application should send back a Response Message with response status set to [ClosedRecover](#), discontinue sending any data on that request token and then remove reference to that request token.

9.2.4.2 Client Session Handles and Request Tokens

The client session handle and request token need to be maintained by the provider application. They are used when providing responses back to the consumer client session. The application needs to maintain a relationship in which multiple request tokens are associated with their respective client session handles.

9.2.4.2.1 Client Session Handle

The client session handle represents a unique client session-e.g., an OMM consuming application. Each client session has one client session handle, which typically has multiple request tokens associated with it.

For more information about Client Session Handles, see section 12.2.4.2, Handling Consumer Client Session Events (Requests).

NOTE: Client session handles are unique and will never be duplicated. Once closed, client session handles can be eventually reused.

9.2.4.2.2 Request Token

This request token object represents unique consumer client session item requests. These request tokens are associated with only the [OMMSolicitedItemEvent](#) event. These item requests can be any of the message model types.

All request tokens from [OMMSolicitedItemEvent](#) events of interaction type “streaming” (see section 9.2.4.1, Interaction Type) need to be maintained in the provider application because they represent an open and updating item. These request tokens must be associated with the client session handle on the event. The lifespan of request tokens of this type will be valid until either the provider application closes the stream or the provider finishes processing the close request from the consumer client session for that request token.

These request tokens will be used when sending corresponding data for the particular request. For more details, see Section 12.2, Interactive Provider.

Not all request tokens need to be maintained by the provider application. Request tokens of interaction type “Non-Streaming” (see section 9.2.4.1, Interaction Type) can be discarded after the “complete” corresponding refresh has been submitted to the OMM Provider. The “complete” refresh message is specified with the [IndicationMaskFlag.RefreshComplete](#) flag being set on the [RespMsg](#). Note that depending upon the size of the refresh, the application may choose to send the refresh in multiple [Submit\(\)](#) calls.

For every new request, the request tokens will be unique. However, it is possible to receive a previously received request token under some circumstances such as an item being re-issued. Request tokens, once closed, can also be eventually reused. An example of this would be a priority change.

If an application attempts to send data on a Request Token that is closed, it may receive an [OMMCmdErrorEvent](#) in response. The application will receive the error event one time per provider on the first submit after the token has been closed. All data sent on a closed Request Token will be dropped.

Providers should not attempt to submit data in the following cases:

- After the associated session has become inactive
- After the application has unregistered the client session associated with the token
- After a complete refresh has been submitted for a non-streaming request
- After the provider has received a close request for an item.

NOTE: For events that only have a method to obtain the request token ([RequestToken](#) property) the corresponding client session handle for that event can still be accessed via the [Handle](#) property on the event.

9.2.4.3 OMMProvider Initialization

To initialize an OMMProvider, the application uses the Session handle and call the `CreateOMMProvider()` method as follows:

```
ommProvider = session.CreateOMMProvider(new RFA_String(appName));
```

Example 124: Initialize Event Source

The `CreateOMMProvider()` method also accepts an optional second parameter which specifies whether or not Completion Events should be sent. By default, Completion Events are not generated. The above example shows only the first parameter, which is the name of the Event Source. This name provides identification for purposes such as logging. To provide flexibility for OMM providing applications, RFA supports multiple OMM Providers per Session, as well as multiple RSSL_PROV type connections per OMM Provider. Thus a single OMM Provider can provide OMM data to OMM consuming applications over several connections.

9.2.4.4 Event Registration

The interactive provider application needs to register events with the particular Interest Specification as follows:

- OMMActiveClientSessionEvent
- OMMInactiveClientSessionEvent
- OMMSolicitedItemEvent
- OMMConnectionEvent
- OMMCmdErrorEvent
- OMMConnectionStatsEvent

For more information about event registration, refer to section 9.2.2.1, Event Registration.

9.2.4.5 Submit Message

The application can send Response Message, Generic Message and Ack Message to the provider.

9.2.4.5.1 Response Message

The application must send Response Message by calling `Submit()` method using `OMMSolicitedItemCmd` with the `RespMsg` and the request token as follows:

```
OMMSolicitedItemCmd itemCmd = new OMMSolicitedItemCmd();

itemCmd.Msg = respMsg;

itemCmd.RequestToken = requestToken;

ommProvider.Submit(itemCmd);
```

Example 125: Submit Response Message

9.2.4.5.2 Generic Message

The application must send Generic Message by calling `Submit()` method using the currently available `OMMSolicitedItemCmd` with `GenericMsg` and the request token as follows:

```
OMMSolicitedItemCmd ommSolicitedCmd = new OMMSolicitedItemCmd();

ommSolicitedCmd.Msg = genericMsg;

ommSolicitedCmd.RequestToken = requestToken;

ommProvider.Submit(ommSolicitedCmd);
```

Example 126: Submit Generic Message

9.2.4.5.3 Ack Message

Ack Message must be sent on the stream as the one on which the Post Message it is acknowledging was received. The application can send Ack Message by calling `Submit()` method using `OMMSolicitedItemCmd` with the `AckMsg` and the request token as follows:

```
OMMSolicitedItemCmd ommSolicitedCmd = new OMMSolicitedItemCmd();

ommSolicitedCmd.Msg = ackMsg;

ommSolicitedCmd.RequestToken = requestToken;

ommProvider.Submit(ommSolicitedCmd);
```

Example 127: Submit Ack Message

9.2.4.6 Unregistration Event

When the application no longer has interest in a particular entity, it may relinquish interest via the `UnregisterClient()` method on the Event Source.

The application unregister the event as follows:

```
ommProvider.UnregisterClient(ommClientSessionIntSpecHandle);
```

Example 128: Unregister Event

9.2.4.7 CleanUp

After no longer using the Event Source, the application should Destroy Event Source via the `Destroy()` method as follows:

```
if (ommProvider != null)
{
    // ...

    ommProvider.Destroy();
    ommProvider = null;
}
```

Example 129: Destroy Event Source

9.2.5 Non-Interactive OMMProvider

The following table depicts the non-interactive provider-released interfaces.

INTERFACE	ABSTRACTION	MESSAGE	DESCRIPTION
OMMProvider	EventSource	None	Event Source that allows clients to send OMM data.
OMMItemIntSpec	Interestspec	None	Specification of interest in login responses.
OMMErrorIntSpec	Interestspec	None	Specification of interest in command errors.
OMMConnectionIntSpec	Interestspec	None	Specification of interest in connection events from RFA adapter.
OMMConnectionStatsIntSpec	Interestspec	None	Specification for interest in Connection Statistics Event.
OMMItemCmd	Command	RespMsg, GenericMsg	Command that contains an item.
OMMItemEvent	Event	RespMsg, GenericMsg	Event that indicates login status if a RespMsg or an Event that contains a GenericMsg is received.
OMMCmdErrorEvent	Event	None	Event that indicates an error in a command.
OMMConnectionEvent	Event	None	An event that contains connection information.
OMMConnectionStatsEvent	Event	None	Contains connection statistics (as the number of bytes on the wire).

Table 72: Non-Interactive OMMProvider Interface

9.2.5.1 Login Request

An OMM non-interactive provider application must send a login request and receive a login success response message before providing any OMM data.

9.2.5.2 Item Tokens

An OMM non-interactive provider application is required to uniquely identify every published item. Items may be identified by usage of *Item Token* or *AttribInfo* objects³². The application is responsible for obtaining and assigning item tokens to items and for maintaining the relationships throughout the lifecycle of the application and items. The application obtains the item tokens by calling the *GenerateItemToken()* method on the *OMMProvider* interface. The application sets the item token on each published *OMMItemCmd* through the *ItemToken* property before calling the *Submit()* method to publish the item.

The lifespan of an item token ends when the application sends a final message for this item, which could be a refreshcomplete with non-streaming status, a closed status, a closed recover status, or a redirected status. In these cases, the non-interactive provider should internally clean up the item token. The application receives a submit failure, an *OMMCmdErrorEvent* if it uses item token after closing the associated item. Additionally, the lifespan of item tokens for all items ends when the application unregisters the login stream or logs off.

NOTE: By default, the *GenerateItemToken()* method returns item token references useable for all but the login domain. Application should use *GenerateItemToken(true)* to obtain an item token for use on messages on the login domain.

The OMM non-interactive provider does not support the *MMT_DICTIONARY* domain.

Usage of item tokens on the login domain is limited to the sending of generic messages only.

Some components, depending on their specific functionality and configuration, require that non-interactive provider applications publish attributes (*AttribInfo*) in all update and status messages. Please refer to the component into which you publish to see whether it is required.

9.2.5.3 OMMProvider Initialization

The initialization method of the Non-Interactive Provider is the same as that for interactive provider. To initialize the Non-Interactive provider, the application uses the Session handle and call the *CreateOMMProvider()* method as follows:

```
ommProvider = session.CreateOMMProvider(new RFA_String(appName));
```

Example 130: Initialize Event Source

The *CreateOMMProvider()* method also accepts an optional second parameter which specifies whether or not Completion Events should be sent. By default, Completion Events are not generated. The above example shows only the first parameter, which is the name of the Event Source. This name provides identification for purposes such as logging. To provide flexibility for OMM providing applications, RFA supports multiple OMM Providers per Session, as well as multiple RSSL_PROV type connections per OMM Provider. Thus a single OMM Provider can provide OMM data to OMM consuming applications over several connections.

³² Prior to RFA .NET 7.2, the application could choose to identify each published item by setting the item token or *AttribInfo*. In RFA .NET 7.2 this choice was deprecated. Refinitiv recommends that you use the item token identifier instead.

9.2.5.4 Event Registration

The non-interactive provider application needs to register events with the particular Interest Specification as follows:

- `OMMItemEvent`
- `OMMConnectionEvent`
- `OMMCmdErrorEvent`

For more information about event registration, refer to section 9.2.2.1, Event Registration.

9.2.5.5 Submit Message

The application can send Response Message and Generic Message to the provider.

9.2.5.5.1 Response Message

The application must send Response Message by calling `Submit()` method using `OMMItemCmd` with the `RespMsg` and the item token as follows:

```
OMMItemCmd itemCmd = new OMMItemCmd();  
  
itemCmd.Msg = respMsg;  
  
itemCmd.ItemToken = itemToken;  
  
ommProvider.Submit(itemCmd);
```

Example 131: Submit Response Message

9.2.5.5.2 Generic Message

The application must send Generic Message by calling `Submit()` method using the currently available `OMMItemCmd` with `GenericMsg` and the item token as follows:

```
OMMItemCmd ommItemCmd = new OMMItemCmd();  
  
ommItemCmd.Msg = genericMsg;  
  
ommItemCmd.ItemToken = itemToken;  
  
ommProvider.Submit(ommItemCmd);
```

Example 132: Submit Generic Message

9.2.5.6 Unregistration Event

When the application no longer has interest in a particular entity, it may relinquish interest via the `UnregisterClient()` method on the Event Source.

The application unregister the event as follows:

```
ommProvider.UnregisterClient(ommConnIntSpecHandle);
```

Example 133: Unregister Event

9.2.5.7 Cleanup

After no longer using the Event Source, the application should Destroy Event Source via the `Destroy()` method as follows:

```
if (ommProvider != null)
{
    // ...

    ommProvider.Destroy();
    ommProvider = null;
}
```

Example 134: Destroy Event Source

Chapter 10 Logger Package

10.1 Logger Package Concepts

The **Logger Package** is one of the fundamental APIs within RFA. It provides the Logger Package a means to manage log events. Applications and other RFA packages (e.g., the SessionLayer Package) may submit log events to the Logger Package. The Logger Package saves the log events to persistent storage areas such as the Windows Event Log or a flat file.

An application may programmatically register to receive log events that have been submitted by an RFA package or by the application itself. Typically, an application will receive log events to perform special processing, such as forwarding the log events to the display or to a log file.

The Logger Package leverages the Event Distribution Model to allow applications to register for log events. It provides the ability to receive asynchronous notifications in the same (or different) thread contexts.

The Logger Package introduces several concepts that relate to event logging. The following table defines several Logger Package concepts.

CONCEPT	DESCRIPTION
Application Logger	An Event Source Factory that controls distribution of log events to persistent storage areas and applications interested in receiving log events
Application Logger Monitor	An Event Source used to programmatically register interest in receiving log events.
Component Logger	Provides the operation to submit log events.
Log	Process of submitting a log event to the RFA Logger Package.
Logger Notify Event	An Event containing message text, message identification and severity of a log event.

Table 73: Logger Package Concepts

The Application Logger is an Event Source Factory used to create both Application Logger Monitors and Component Loggers. It depends on both the Common and Config Packages—the former for its event distribution mechanism and the latter for its configuration information.

Applications can use the Application Logger Monitor to open an Event Stream. While the Event Stream is open, the application may receive Logger Notify Events from other RFA packages, or the application itself. A Logger Notify Event contains information such as message text, message identification and severity.

An application first registers to receive log events, and subsequently receives log events as they become available. Receiving log events is an asynchronous operation.

The following diagram illustrates the relationship between key Logger Package components.

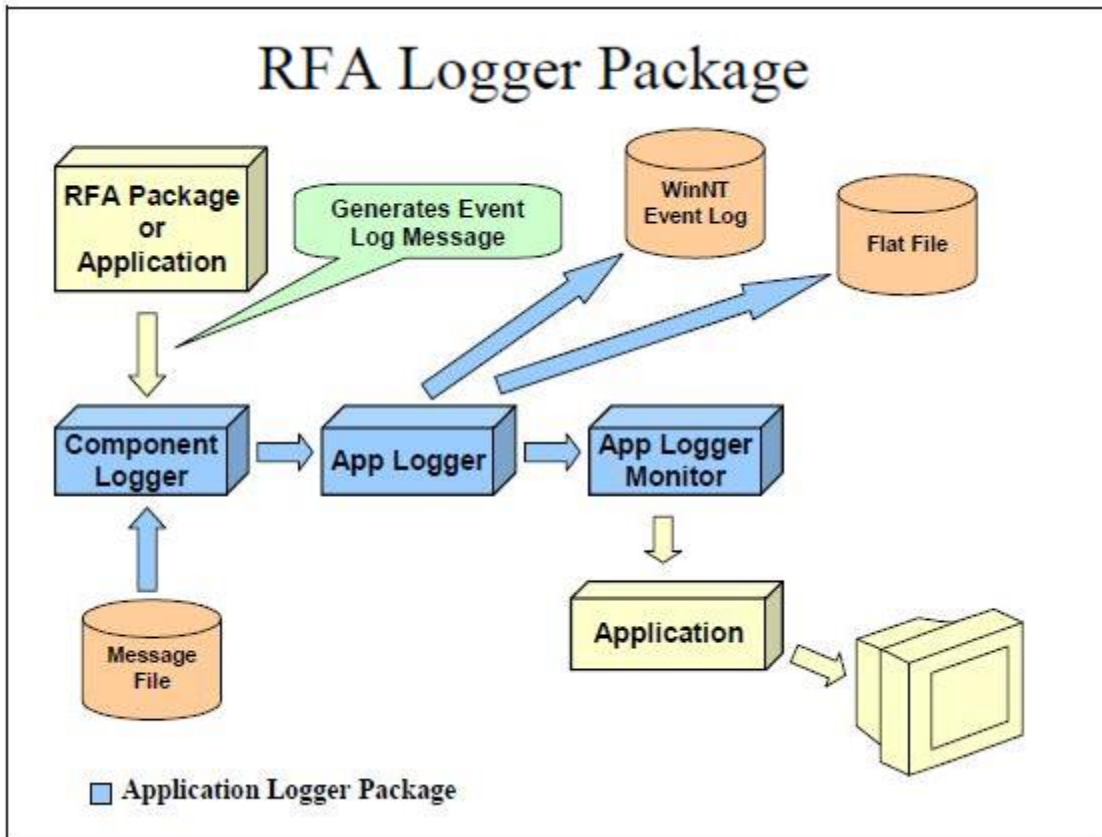


Figure 71: Logger Package

The Component Logger provides an application the ability to submit log events. Submitting a log event is a synchronous operation. Either applications or RFA may submit log events. An application may specify additional event information such as message text, message identification, and severity. Applications may also define the format of new log events using the Microsoft Message Compiler-formatted *Message Files*.

Log events distributed to applications are formatted based on a Component Logger's message file, referenced from its configuration. Logged events distributed to a flat file are formatted. However, log events distributed to the Windows Event Log are unformatted. (The Windows Event Viewer formats these log events.)

Applications can also initialize the component logger with a populated map of logger messages via the [LogMsgMap](#) interface. Using a [LogMsgMap](#) is useful for storing the log messages internally instead of loading them from an external Message Compiler file. The implementation of a [LogMsgMap](#) is customer specific so long as it maps log IDs to messages and implements the following method:

```
RFA_String GetMsg( uint id );
```

Example 135: Implement LogMsgMap.GetMsg() method

10.1.1 Threading Model

The Logger Package provides applications full control over an application's threading model. With very few exceptions, the Logger Package never calls application code in the context of a Logger Package thread. See section 5.2.1, Event Distribution Model.

The Application Logger Monitor delivers log events by posting them to an Event Queue. The application initiates event processing by dispatching them from the Event Queue. When a log event is dispatched, the Application Logger Monitor passes the log event to the corresponding Event Handler. It is the application's responsibility to dispatch log events generated by the Application Logger Monitor. By dispatching log events in its own context, the application maintains full control over the usage of its threads.

10.2 Logger Package Usage

10.2.1 Configuring the Logger Package

To initialize the configuration for the Logger Package, the application populates the Config Database as described in section 8.2.3, Populate Config Database. The Application Logger and Component Logger require access to the Config Database specified by the Context name (which must be "RFA"). Before using any interfaces in the Logger Package, a Configuration Database with the same name must have been instantiated and populated with the configuration information required by the Logger Package.

Configuration for the Application Logger resides in the `\Logger\AppLogger` node and includes configuration parameters to enable and disable the various persistent storage areas (e.g., `windowsLoggerEnabled`).

Each Component Logger configures itself at startup (or when instantiated) by retrieving its configuration information from the Configuration Database. It uses its own name to know which sub-tree within the Configuration Database hierarchy it must use to configure itself. For example, the configuration location for the SessionLayer Component Logger is `\Logger\ComponentLoggers\SessionCore\`.

Because the Application Logger name is tied to the name of the Configuration Database, there is only one occurrence of the Application Logger configuration per database instance. However, because multiple instances of Component Loggers are allowed per Application Logger, there might be multiple occurrences of a Component Logger configuration within a single Configuration Database. The configuration for these components is shown in the following diagram.

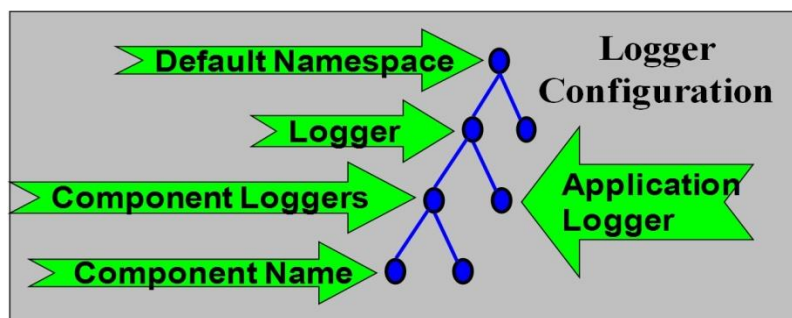


Figure 72: Logger Package Configuration

The configuration parameters of each Component Logger include a message file parameter. This parameter points to a loadable library containing text messages indexed by the `LogID`. This configuration can be superseded by the configuration parameter `useInternalLogStrings`, which embeds text messages in RFA libraries.

Tuning of the Windows Event logger may be required—e.g., to avoid the scenario of excessive Log Events negatively impacting application performance. For specific Logger configuration parameters and configuration information, see the *RFA Configuration Guide .NET Edition*.

10.2.2 Logging in Application

Typically, if an application wants to perform logging, it must perform these general steps.

PHASE	RFA INTERFACE	DESCRIPTION
Initialization	ApplicationLogger.Acquire()	Initializes the Application Logger.
Initialization	ApplicationLogger.CreateApplicationLoggerMonitor()	Initialize the Application Logger Monitor.
Initialization	ApplicationLogger.CreateComponentLogger()	Creates a Component Logger.
Initialization	AppLoggerMonitor.RegisterLoggerClient()	Registers the logger monitor for interest in receiving logger notification events.
Run-time	ComponentLogger.Log()	Logs a pre-formatted message.
Run-Time	ProcessEvent()	If the event type is LoggerEventTypeEnum , specifies a LoggerNotifyEvent that can be acted on.
Shutdown	N/A	Delete the component logger (if necessary)
Shutdown	AppLoggerMonitor.UnregisterLoggerClient() AppLoggerMonitor.Destroy()	Unregistered and deletes the Application Logger Monitor.
Shutdown	ApplicationLogger.Release()	Release the Application Logger.

Table 74: Logging in Applications

10.2.3 Application Logger

The following diagram shows the activities to manage an Application Logger. An application must initialize an Application Logger to submit log events or receive log events. The diagram alters Figure 29 by showing Logger-specific actions as shaded icons.

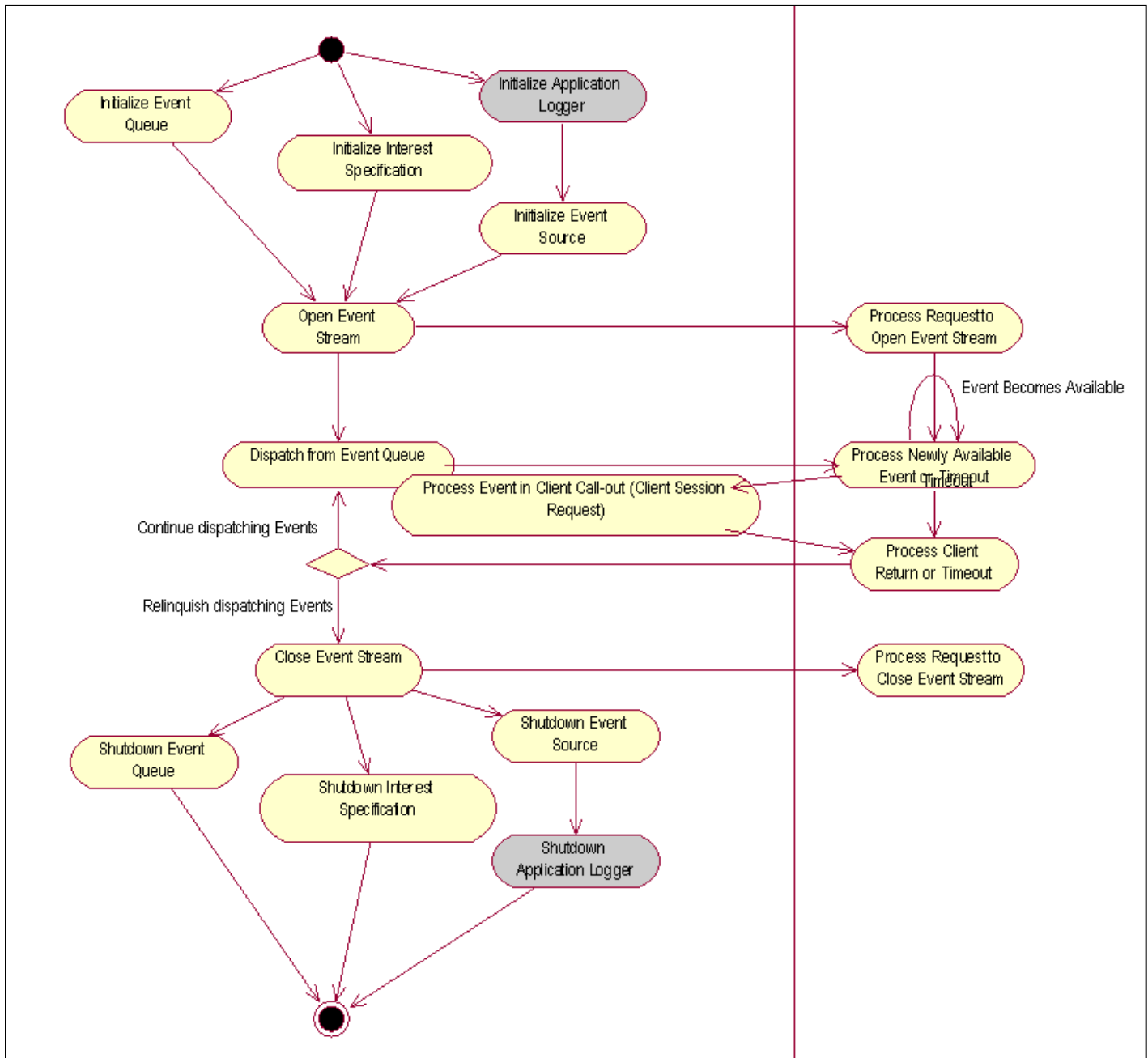


Figure 73: Application Logger

10.2.3.1 Initializing Application Logger

Typically one of the application's first activities is to Initialize the Application Logger, as shown below. To initialize an Application Logger, the application calls the static `Acquire()` method as follows:

```
// --- Initialize Application Logger ---  
ApplicationLogger applicationLogger = ApplicationLogger.Acquire(Context.Name);
```

Example 136: Initialize Application Logger

The static `Acquire()` method requires at least one parameter, which is the Application Logger name. The Application logger name instructs the Application Logger to obtain configuration information from the Config Database instance of the same name. In this case, the application specifies the Application Logger name provided by the `Context` (which will always be "RFA"). Other RFA packages (e.g., the SessionLayer Package) also use the same Application Logger instance. `Acquire()` returns a reference to an Application Logger.

RFA may share the instance returned by `Acquire()` among the other application components. This is known as Instance Sharing. RFA bases a unique instance of a Config Database on its name. Benefits of Instance Sharing include alleviating the need for the application to pass around an Application Logger reference among multiple application components and the ability to minimize resource consumption.

10.2.3.2 Shutting Down Application Logger

After no longer using these other Application Logger interfaces, the application should perform the Shutdown Application Logger activity via the `Release()` method as follows:

```
// --- Shutdown Application Logger ---  
applicationLogger.Release();
```

Example 137: Shutdown Application Logger

10.2.4 Application Logger Monitor

The following diagram shows typical activities for using the Application Logger Monitor Event Source. This diagram alters Figure 73 by depicting specific Application Logger Monitor activities as shaded icons. Applications can obtain interest in log events by performing these actions.

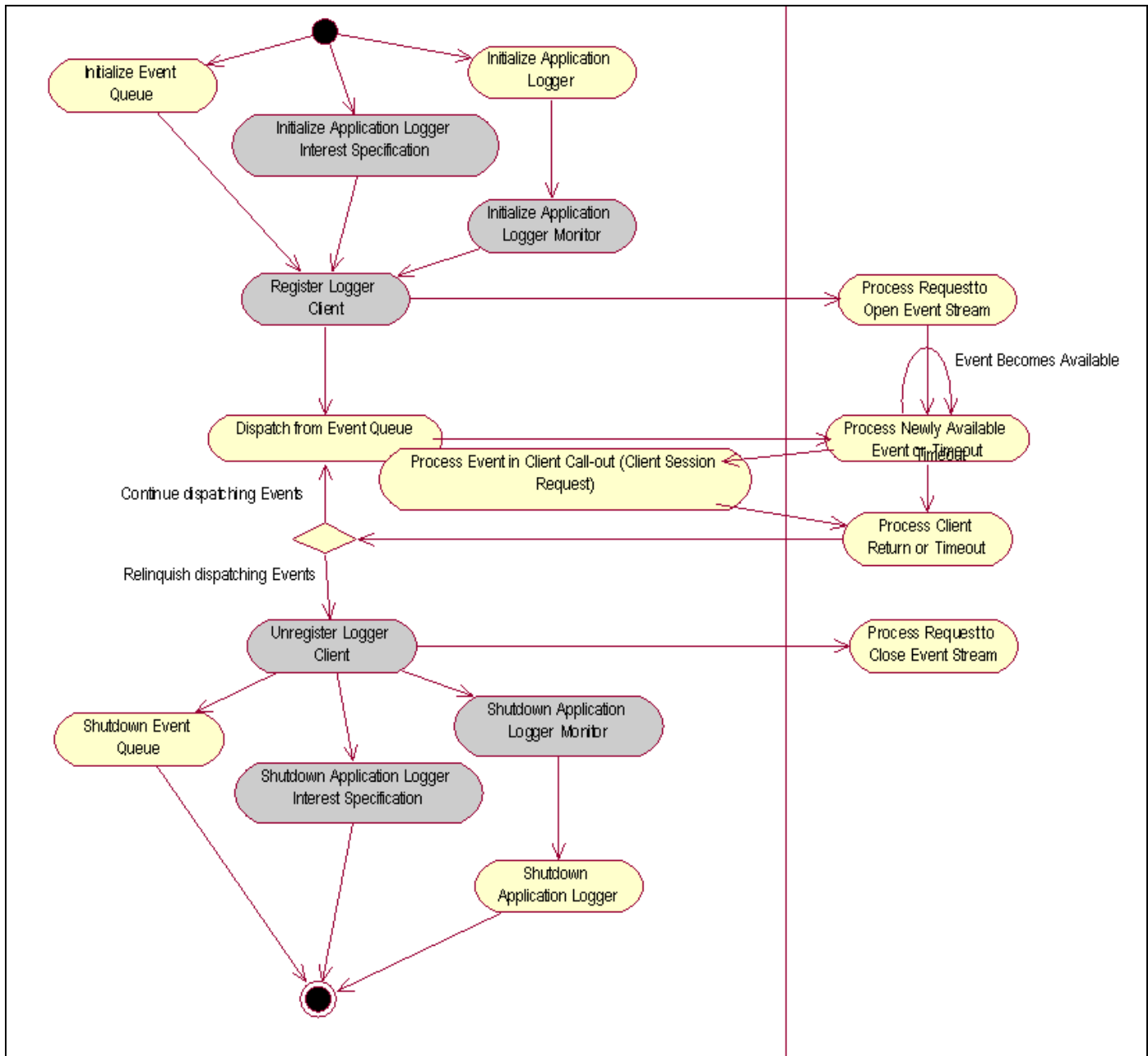


Figure 74: Application Logger Monitor: Register Client

10.2.4.1 Initializing Application Logger Monitor

Assuming an application has initialized an Application Logger the next step is to perform the Initialize Application Logger Monitor activity as shown *above in* Figure 74. To initialize an Application Logger Monitor, the application uses the Application Logger handle and calls the `CreateApplicationLoggerMonitor()` method as follows:

```
// --- Initialize Application Logger Monitor ---

AppLoggerMonitor appLoggerMonitor = applicationLogger.CreateApplicationLoggerMonitor(new
RFA_String("DemoApplicationLoggerMonitor"), false);
```

Example 138: Initialize Application Logger Monitor

The `CreateApplicationLoggerMonitor()` method accepts two parameters. The first parameter is the name of the Application Logger. Because it was created from an Application Logger with the name of the `Context` (which is always “RFA”), the Application Logger Monitor can receive log events submitted by other RFA packages.

The second parameter (`false`) is a bool that indicates not to use Completion Events. See Section 5.1.1.5, Completion Events for details on using Completion Events.

10.2.4.2 Registering for Log Events

Assuming an application has initialized an Application Logger Monitor, the application next performs the Initialize Application Logger Interest Specification and Register Logger Client actions shown in Figure 74.

Before registering for log events, an application must define the code to process log events when RFA invokes the Client callback method (i.e., `ProcessEvent()`). First, the application defines code to process log events by deriving from the Client interface as shown in the example below:

```
// --- MyAppLoggerMonitorClient ---
public class MyAppLoggerMonitorClient : Client
{
    public void ProcessEvent(Event rfaEvent)
    {
        switch (rfaEvent.Type)
        {
            case LoggerEventTypeEnum.LoggerNotifyEvent:
            {
                LoggerNotifyEvent logEvent = (LoggerNotifyEvent)rfaEvent;
                Console.WriteLine("Component:" + logEvent.ComponentName.ToString() + " LogID:" +
                    logEvent.LogID + " Severity:" + logEvent.Severity);
                Console.WriteLine(" Text:" + logEvent.MessageText.ToString());
                break;
            }
            default:
                break;
        }

        if (rfaEvent.IsEventStreamClosed)
            Console.WriteLine("Event stream is closed");
    }
}
```

Example 139: Application Logger Monitor Client

In this example, the application implements the `ProcessEvent()` by switching on the Event type and calling a performing an Event processing method. In this case there is only one Event type (`LoggerEventTypeEnum`). However, if the application were to use this Client to process other Events, it would define additional case statements.

In the `LoggerEventTypeEnum` case statement, the application first downcasts the event to a `LoggerNotifyEvent`. Next, the application outputs information provided in the `LoggerNotifyEvent`.

After processing a log event, the application detects whether the Event Stream has been closed by the `IsEventStreamClosed` property. Because the application did not specify Completion Events when it created the Application Logger Monitor, RFA will not send a Completion Event when the Event Stream is closed. However, an application can use this same code to process Completion Events.

If the application did specify Completion Events when creating the Application Logger Monitor, RFA will send a Completion Event when the Event Stream is closed. In this case, a Completion Event will cause branching to default in the switch statement. In this example, default does no processing. However, by nature of the Completion Event, the `IsEventStreamClosed` will always be true and the application is guaranteed it will receive no more Events for this Event Stream. See section 5.2.1.5, Using Event Distribution Model in a Multiple-thread Context for details on using Completion Events.

After defining the code to process log events, the application registers to receive log events. In the following example, the application first creates an `AppLoggerInterestSpec` Interest Specification. Next, the application sets the minimum severity of the log event. In this case the application sets severity to be `CommonErrorSeverityTypeEnum.Information`, which implies the application receives log events having at least the severity of informational.

```
// --- Register for Application Logger Events ---
// Create Application Logger Interest Specification
AppLoggerInterestSpec appLoggerInterestSpec = new AppLoggerInterestSpec();

// Set Minimum Severity to Information
appLoggerInterestSpec.MinSeverity = CommonErrorSeverityTypeEnum.Information;

// Register Client using Application Logger Interest Specification
long AppLoggerInterestSpecHandle = appLoggerMonitor.RegisterLoggerClient(eventQueue, appLoggerInterestSpec,
myAppLoggerMonitorClient, null);
```

Example 140: Register for Application Logger Events

Next the application opens an Event Stream by calling `RegisterLoggerClient()` using the handle to the Application Logger Monitor (`appLoggerMonitor`). As with the opening of any Event Stream, this method accepts four parameters:

- `eventQueue` dereferences a handle to the Event Queue that will receive the events.
- `appLoggerInterestSpec` is the Interest Specification described in the previous paragraph.
- `myAppLoggerMonitorClient` is the Client described in this section.
- The fourth parameter is the Closure, as described in the Common Package. (See section 5.1.1.6, Closures.)

As indicated in the Common Package, the return value of a method that opens an Event Stream is always a Handle.

10.2.4.3 Closing an Event Stream for the Application Logger Monitor

When the application no longer has interest in receiving log events, it may relinquish interest via the Unregister Logger Client activity as in Figure 74. The application relinquishes interest by calling `UnregisterLoggerClient()` on the Event Source. The following example closes the Event Stream opened for log events in section 10.2.4.2, Registering for Log Events.

```
// --- Unregister for Application Logger Events ---  
appLoggerMonitor.UnregisterLoggerClient( AppLoggerInterestSpecHandle );
```

Example 141: Unregister for Application Logger Events

In this example, the application closes the Event Stream by calling the `UnregisterLoggerClient()` method on the Event Source (`appLoggerMonitor`). This method accepts one parameter. This parameter (`AppLoggerInterestSpecHandle`) is a reference to the Handle that was returned by RFA from the `RegisterLoggerClient()` call.

10.2.4.4 Shutting Down Application Logger Monitor

When the application no longer needs the Application Logger Monitor, it should perform the Shutdown Application Logger Monitor activity as shown in Figure 74. The application performs this activity via the `Destroy()` method as follows:

```
// --- Shutdown Component Logger ---  
componentLogger.Destroy();
```

Example 142: Shutdown Application Logger Monitor

10.2.5 Component Logger

The following diagram shows typical actions using a Component Logger. This diagram depicts Component Logger-specific actions as shaded icons. Applications can perform these actions to submit log events.

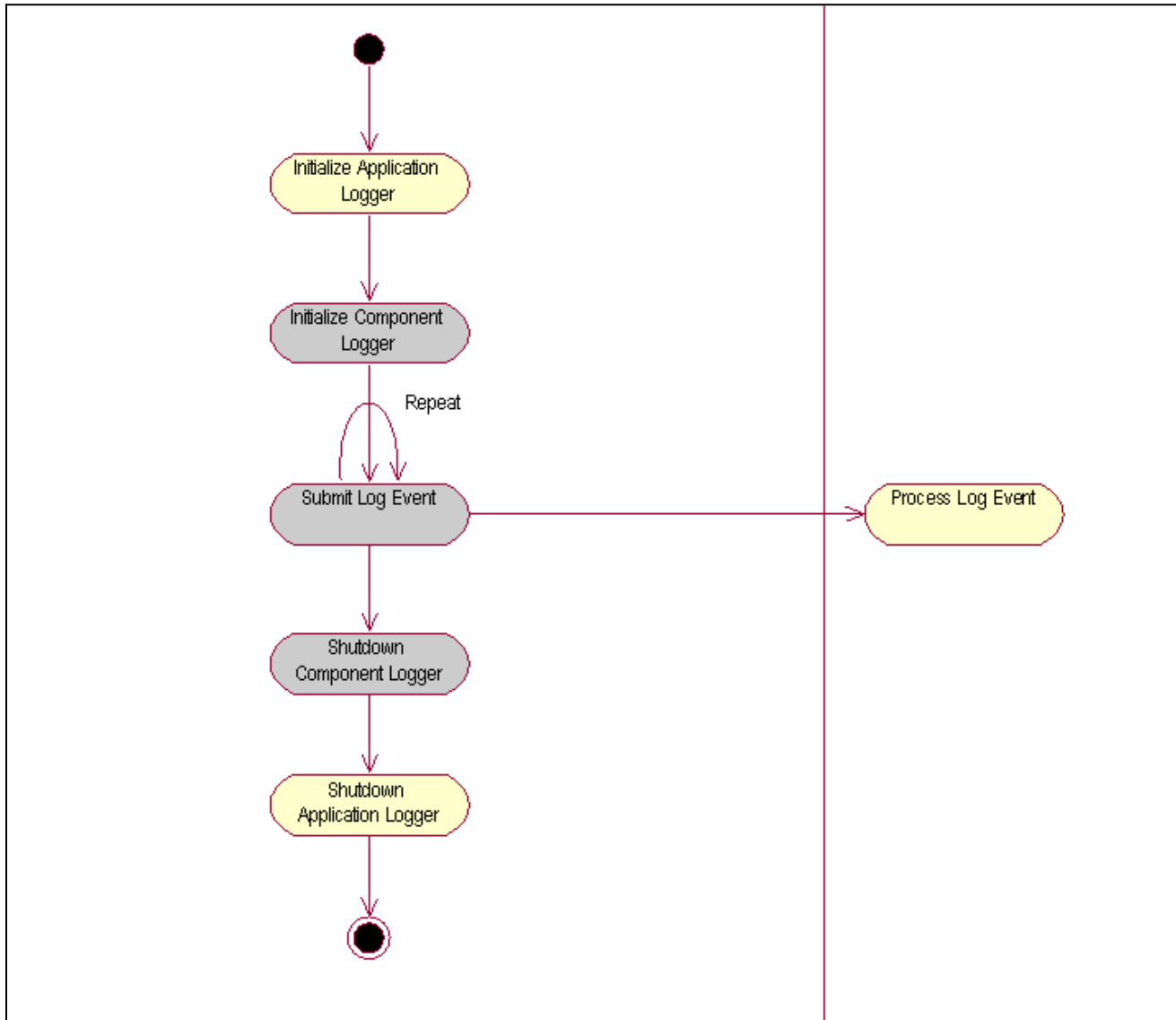


Figure 75: Component Logger Lifecycle

10.2.5.1 Initializing Component Logger

Assuming an application has initialized an Application Logger, the next step is to perform the Initialize Component Logger action as shown in Figure 75. To initialize a Component Logger, the application uses the Application Logger reference and calls the `CreateComponentLogger()` method as follows:

```
// --- Initialize Component Logger ---
ComponentLogger componentLogger = applicationLogger.CreateComponentLogger( new RFA_String("LogTest")) ;
```

Example 143: Initialize Component Logger

The `CreateComponentLogger()` accepts two parameters. The first is the name of the Component Logger. This name is arbitrary but usually references configuration information for this specific Component Logger. The second parameter is optional and defaults to `NULL`. If present, it passes a reference to a `LogMsgMap` as described in section 10.2.5.5, Using a `LogMsgMap`.

10.2.5.2 Submitting Log Events

Assuming an application has initialized a Component Logger, the application typically next performs the Submit Log Event activity as shown in Figure 75.

Once the application has created a Component Logger it can submit log events as shown in the example below:

```
// --- Submit Log Event ---
List<RFA_String> parameter = new List<RFA_String>();
RFA_String name1 = new RFA_String("Parameter1");
RFA_String name2 = new RFA_String("Parameter2");
parameter.Add(name1);
parameter.Add(name2);

if (componentLogger.Log(GENERIC_TWO, CommonErrorSeverityTypeEnum.Information, parameter))
    Console.WriteLine("Submitted log event");
else
    Console.WriteLine("Error upon submitting log event");
```

Example 144: Submit Log Events

The application uses a reference to the Component Logger (`componentLogger`) and calls `Log()`. This method accepts a variable number of arguments. These arguments are a log ID (`GENERIC_TWO`), error severity (`CommonErrorSeverityTypeEnum.Information`), and a variable number of application-defined strings ("Parameter1" and "Parameter2"). The log ID references a specific event message in the message file as described in section 10.2.5.4, Using a Message File. The error severity indicates the level of criticality. The remaining arguments are optional and may be any application-defined strings.

If the optional parameters in the variable argument list are specified they will be formatted along with the message text and sent to the Logger Monitor or flat file. However, if the log events are being sent to the Windows Event Log, the Component Logger forwards these optional parameters to the Windows Event Log, which then formats the log events.

The severity of each log event is specified with the second parameter. (In this example, `CommonErrorSeverityTypeEnum.Information`) Severity is classified as follows:

- **Error** indicates an unrecoverable event such as missing configuration parameters.
- **Warning** indicates a recoverable event such as a loss of connection to a back-end system.
- **Information** indicates general information such as debugging information.
- **Success** indicates a successful attempt.

The `Log()` method returns a `bool` indicating whether the log event was successfully submitted. The example outputs a status message based on the return value.

10.2.5.3 Shutting Down Component Logger

When the application no longer needs the Component Logger, it should perform the Shutdown Component Logger activity as in Figure 75. The application shuts down the Component Logger via the `Destroy()` method as follows:

```
// --- Shutdown Component Logger ---  
componentLogger.Destroy();
```

Example 145: Shutdown Component Logger

10.2.5.4 Using a Message File

A Component Logger is required for an application to submit log events. RFA provide LogTest example to show functionality of Logger. The following diagram illustrates message file build process in LogTest example.

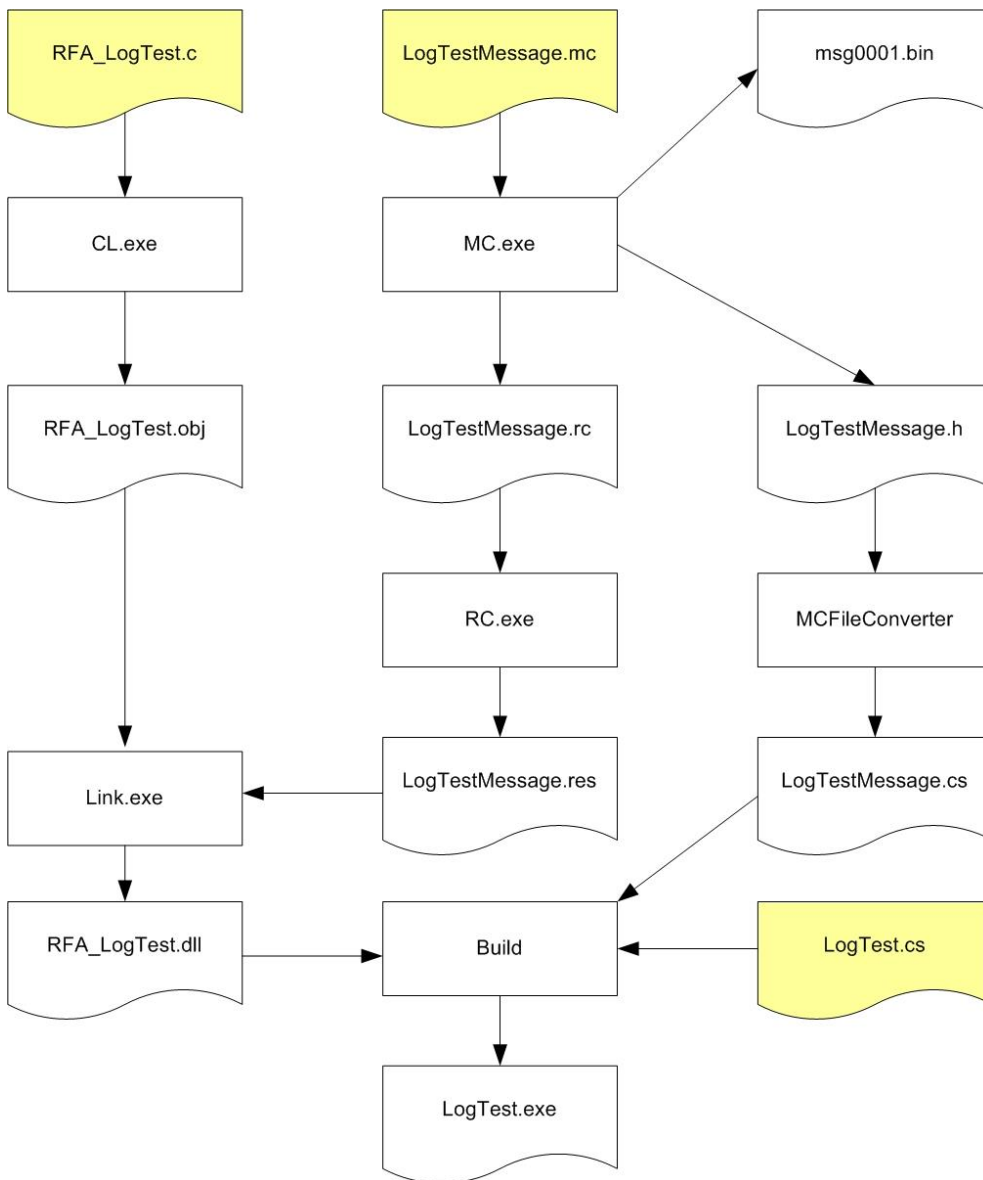


Figure 76: Message Compiler Build Process

To Generate DLL file, the application must use Microsoft Message Compiler (MC.exe) and Microsoft Windows Resource Compiler (RC.exe) to generate the resource file (LogTestMessage.res). DLL file will be linked from the object file (RFA_LogTest.obj) and the resource file (LogTestMessage.res).

The header file (LogTestMessage.h) that generate from Microsoft Message Compiler must be converted to source file (LogTestMessage.cs) with MCFileConverter that provided in LogTest example.

After the LogTestMessage.cs file is created, the message identification numbers on that file are used to build the LogTest example in order to use the message formats defined in the DLL file at runtime.

The following is the partial example of *.MC file containing messages that user can use to log the events:

```
MessageId=1
```

```

Severity=Informational
Facility=Application
SymbolicName=LM_GENERIC_ONE_USER
Language=English
user defined: %1
.
MessageId=2
Severity=Informational
Facility=Application
SymbolicName=LM_GENERIC_TWO_USER
Language=English
user defined: P[1]:%1, P[2]:%2
.
MessageId=3
Severity=Error
Facility=Application
SymbolicName=LM_ERROR_USER
Language=English
User defined error : %1
.
MessageId=4
Severity=Informational
Facility=Application
SymbolicName=LM_GENERIC_SIX
Language=English
user defined: P[1]:%1, P[2]:%2,P[3]:%3, P[4]:%4,P[5]:%5, P[6]:%6
.
MessageId=5
Severity=Informational
Facility=Application
SymbolicName=LM_GENERIC_TEN
Language=English
user defined: P[1]:%1, P[2]:%2,P[3]:%3, P[4]:%4,P[5]:%5, P[6]:%6,P[7]:%7, P[8]:%8,P[9]:%9, P[10]:%10
.

```

Example 146: Formatting in a Message Compiler (.MC) file

Each `MessageId` section ends with period (.) symbol. The last line before each period (.) is the log message to display, with `%1`, `%2` representing the number of parameters to be replaced in the formatted string.

Applications can submit the log event using the symbol name, as shown below:

```

// --- Submit Log Event ---

// Start logging messages
bool ret;
List<RFA_String> argList = new List<RFA_String>();

// ...

argList.Clear();
argList.Add(new RFA_String("one"));
argList.Add(new RFA_String("two"));
argList.Add(new RFA_String("three"));
argList.Add(new RFA_String("four"));
argList.Add(new RFA_String("five"));
argList.Add(new RFA_String("six"));
ret = componentLogger.Log(LogTestMessages.LM_GENERIC_SIX, CommonErrorSeverityTypeEnum.Information, argList);

```

Example 147: Submitting a log event

As seen above, when user submit the log event with a symbolicName LM_GENERIC_SIX the format string containing %1, %2, %3, %4, %5 and %6 are replaced by “one”, “two” “three” “four” “five” and “six.”

A call to `Log()` is the same whether the application is using Windows message compiler string resources, dynamically loaded and parsed `.MC` files, or a `LogMsgMap`.

10.2.5.5 Using a LogMsgMap

Applications can also initialize the Component Logger with a `LogMsgMap`, which is a customer-specific implementation of a Map of log message IDs and corresponding messages. For example:

```
RFA_String appLoggerName = new RFA_String( "TestApp" );
ApplicationLogger appLogger = ApplicationLogger.Acquire( appLoggerName );

// The definition and population of a LogMsgMapImpl is customer-specific
// LogMsgMapImpl is implementation class of LogMsgMap
LogMsgMapImpl logMsgMap = new LogMsgMapImpl();

// Assume that this is a function that loads log IDs and strings into the message map
LoadLoggerStrings( logMsgMap );

ComponentLogger componentLogger = appLogger.CreateComponentLogger( componentLoggerName, logMsgMap );
```

Example 148: Using a LogMsgMap

The default value of a `LogMsgMap` is NULL, and by default the component logger initializes without one (i.e., it uses resource strings embedded in DLLs).

NOTE: Use of the `LogMsgMap` also requires that the Logger configuration parameter `useInternalLogStrings` be set to `true`. Also, using internal log strings and logging via the Windows Event Log are mutually exclusive.

10.2.5.6 Using the Windows Event Logger

Error events can be also directed to the Windows Event Logger and stored there. This is useful, for example, for remotely collecting information from networked machines running RFA applications. When the Windows Event Logger is activated (via the configuration parameter `windowsLoggerEnabled`) RFA registers the DLL that contains the message with the event logger. Only the message ID, severity, and optional message arguments are sent to the event logger.

Windows stores this information about the library the log message came from in the registry under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\<DLLName>`. If the DLL used to generate the message is moved or replaced, the Event Log will still show the Event ID, severity, and optional arguments, but will not be able to show the original log message.

NOTE: Use of Windows Event logging requires administrative privileges.

10.2.6 Logger Internationalization

The Logger Package provides multiple language support of log events submitted by either the application or RFA package. One may modify the text of log events by editing the `.MC` files directly. Since the `.MC` files don't support Unicode text, the changes must be limited to using ASCII characters.

NOTE: Only the message text may be modified, not the parameters submitted by an application or RFA package.

10.3 Logger Usage Guidelines

10.3.1 Control Log Event Fanout

Log events sent from a single Component Logger fan out to all Application Logger Monitors that were created from the same Application Logger. An application can control fanout based on Application Logger instances.

An application may choose to separate log events submitted by RFA packages from log events submitted by the application. To control this fanout, the application can create two instances of an Application Logger Monitor. Alternatively, the application can submit log events to the same Application Logger instance as used by RFA packages. In this case, log events from both the application and RFA packages will be combined into a single stream of log events.

10.3.2 State and Status Code in Status Interface

Descendents of the Status Interface typically provide State and Status Codes. Typical applications are expected to programmatically react to State, but not Status Code. There is no such expectation of log events.

10.3.3 RFA Application Logger Name

The name given to the Application Logger that is used by all RFA-compliant software packages is “RFA.” All RFA-compliant software packages that submit log events rely on this Application Logger instance.

10.3.4 Mixed Component Logger Configuration

Creating one Component Logger with a `LogMsgMap` and another without one is an unsupported configuration. Component Loggers either all need to load and parse log messages from files or use internal strings.

10.3.5 RFA Application Logger Shutdown

On shutdown, the application must call `Release()` on the `AppLogger` for each call to `Acquire()`. The application must also `Destroy()` any Application Logger Monitors or Component Loggers in the reverse order they were created.

The application should call `Uninitialize()` on the Context, which causes the Context to release its reference to the “RFA” `AppLogger`.

10.3.6 Default Namespace

Namespaces are represented in the Configuration Database as a `ConfigTree` that is a child of the Root Config Tree. The Logger Package relies on a specific namespace known as the “Default” namespace. The Default namespace must reside in the “RFA” Configuration Database and contain the proper configuration for the Logger Package.

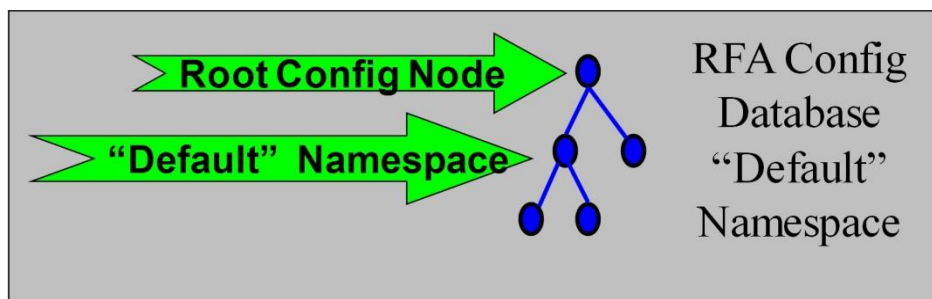


Figure 77: RFA Configuration Database “Default” Namespace

Chapter 11 Implementing OMM Consumer

OMM Consumer application can consume market information or provide market information (Posting) to RTDS, OMM Provider or RDF Direct. This chapter describes how to create the OMM Consumer application and provides example in each section. For more information, refer to StarterConsumer example in RFA package.

11.1 Application Lifecycle

The StarterConsumer is a starter consumer example application distributed with the RFA package to demonstrate the usage of the OMM Consumer. The diagram below illustrates the full lifecycle of the StarterConsumer example.

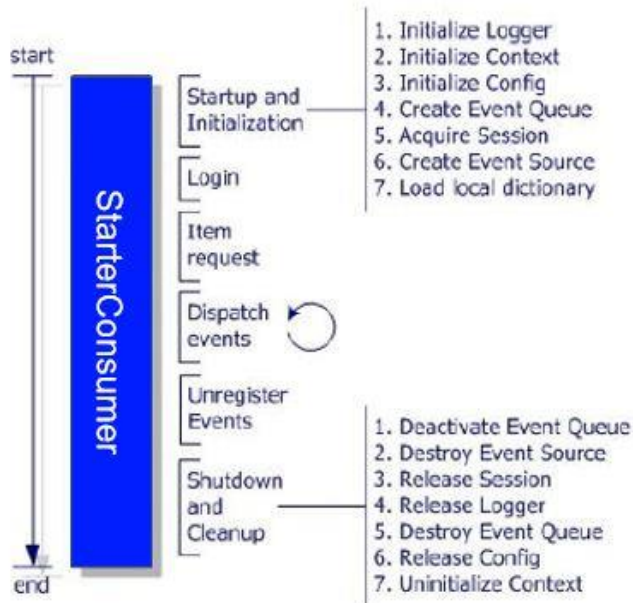


Figure 78: The lifecycle of StarterConsumer example

The following activity diagram depicts high-level activities involved in Initialization , creating Request Message, registering event, event processing, sending message, unregistering event, and clean up.

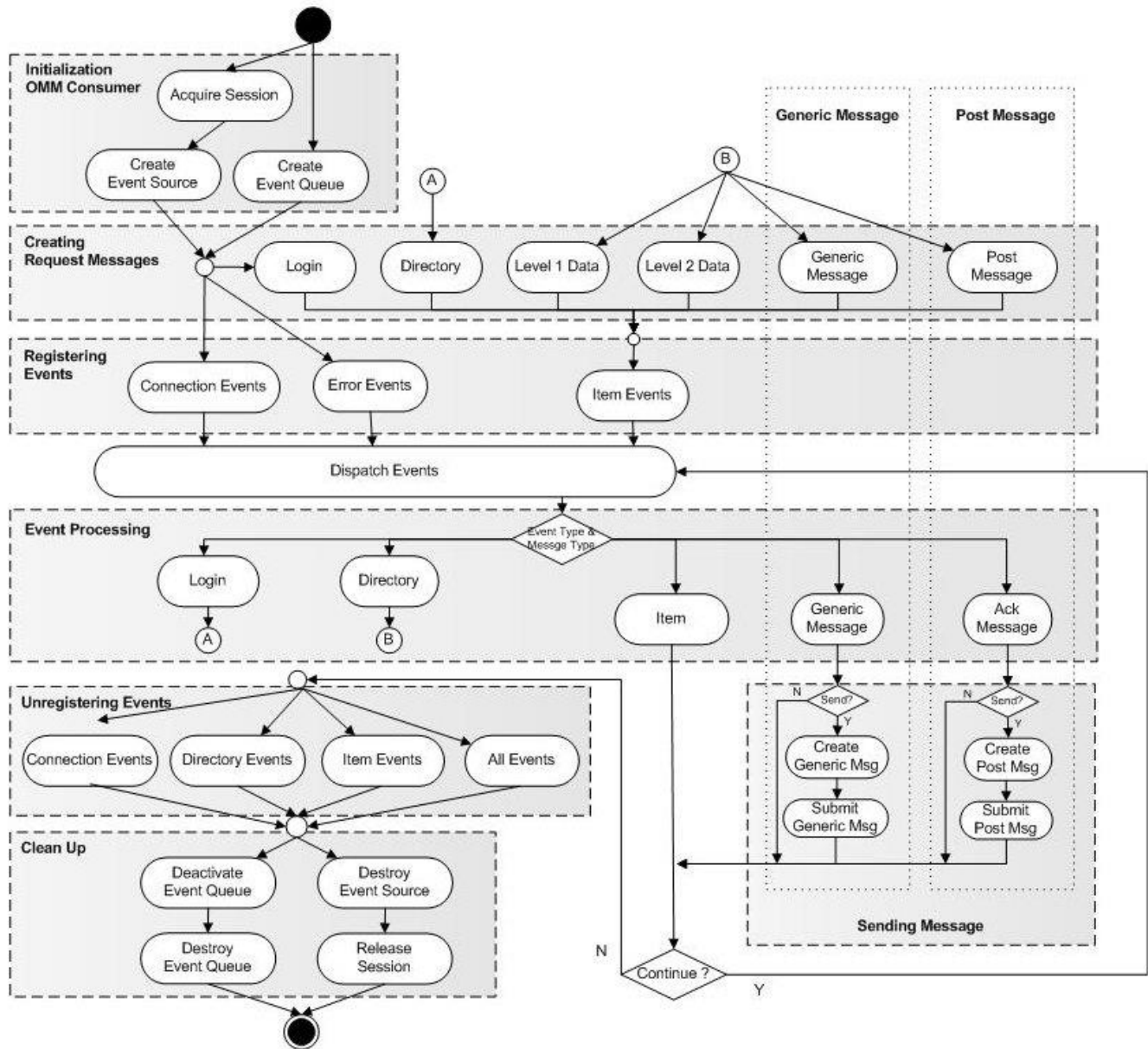


Figure 79: OMM Consumer

11.2 Initialization OMM Consumer

The initialization phase is the first step to initialize a Session which is used to establish connectivity with various back-end systems. After that the session is used to create an OMM Consumer event source for registering events and submitting data. Finally, an Event Queue is created for dispatching events from OMM Consumer. The following is the activity snippet diagram for Initialization OMM Consumer.

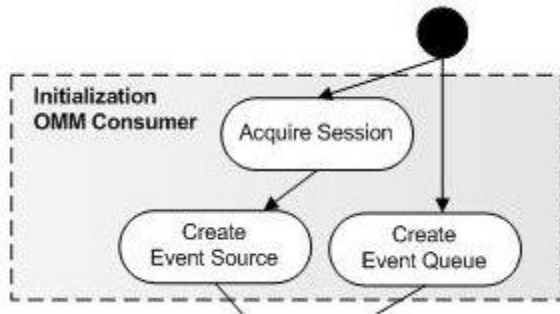


Figure 80: Initialization OMM Consumer

11.2.1 Acquire Session

Typically, one of the application's first activities is **Initialize Session**. To initialize a Session, the application calls the static `Acquire()` method as follows:

```
session = Session.Acquire(cfgVariables.SessionName);
```

Example 149: Acquire Session

The static `Acquire()` method requires at least one parameter, which is the Session name. In this case, the application specifies the Session name from the configuration. The Session name both references configuration information and provides identification for purposes such as logging. The static `Acquire()` method returns a Session object.

11.2.2 Create Event Source

To initialize Event Source, the application uses the Session object and calls the `CreateOMMConsumer()` method as follows:

```
ommConsumer = session.CreateOMMConsumer(new RFA_String(appName));
```

Example 150: Create Event Source

The `CreateOMMConsumer()` method also accepts an optional second parameter which specifies whether Completion Events should be sent. By default, Completion Events are not generated. The above example shows only the first parameter, which is the name of the Event Source. This name provides identification for purposes such as logging.

11.2.3 Create Event Queue

To initialize Event Queue, the application calls the static `Create()` method as follows:

```
eventQueue = EventQueue.Create(new RFA_String("myEventQueue"));
```

Example 151: Create Event Queue

The static `Create()` method requires at least one parameter, which is the Event Queue name. In this case, the application specifies the Event Queue name of `myEventQueue`. The static `Create()` method also accepts an optional second parameter which specifies whether or not turn on statistics for this Event Queue. The Event Queue name both references configuration information and provides identification for purposes such as logging. The static `Create()` method returns a `EventQueue` object.

11.3 Process for Creating Request Message

After the initialization phase is completed, the application needs to create Request Message. For more information about the request message, refer to section 7.2.1.1, Encoding Request Message.

The following is the activity snippet diagram for creating Request Message.

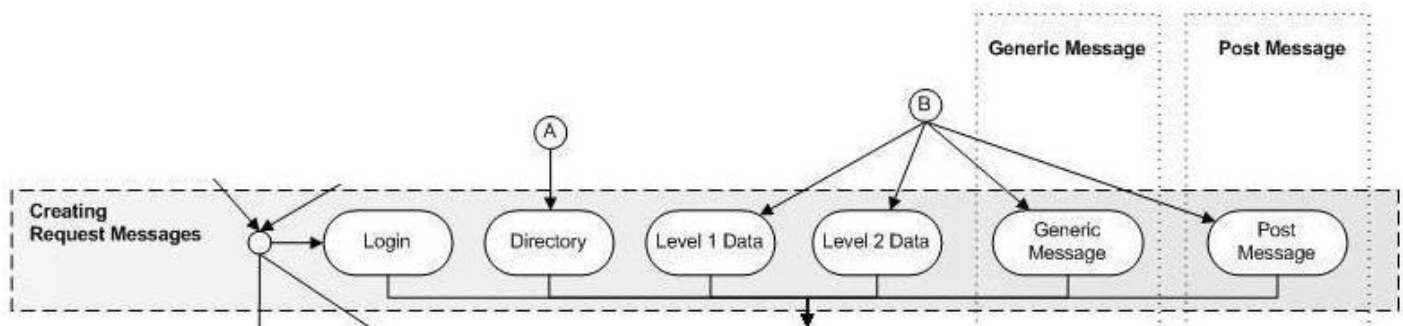


Figure 81: Process for creating Request Message

- In general, requests to the [OMMConsumer](#) are made through a single [InterestSpec](#) type, and responses are received in a single Event type. To make a request to the [OMMConsumer](#), the application needs to do the following:
- Create a Request Message ([Reuters.RFA.Message.ReqMsg](#)). The request message is subclassed from [Reuters.RFA.Common.Msg](#).
- Set the Message Model type of the request using the [MsgModelType](#) property. Possible values are defined in the *RFA Reference Manual .NET Edition*.
- Create a request attribute object ([Reuters.RFA.Message.AttribInfo](#)).
- Populate the [AttribInfo](#) object with request attributes. Refer to the *RFA RDM Usage Guide .NET Edition* for details regarding the meaning and applicability of each attribute for each Message Model.
- Populate the request message with the [AttribInfo](#) object using the [AttribInfo](#) property.
- Create the [OMMItemIntSpec](#) InterestSpec and populate it with the request message using the [Msg](#) property. The InterestSpec essentially acts as a wrapper around the request message.
- Call the [RegisterClient\(\)](#) method of the OMM Consumer Event Source, passing in the Event Queue to which responses should be sent, the InterestSpec described in the previous bullet, the Client whose callback will receive the responses and the Closure. The returned Handle has to be retained and will be used for closing the new Event stream.

The InterestSpec, Request Message, and request attribute objects may be reused for making subsequent requests. The application can release these objects at any time after calling [RegisterClient\(\)](#) because RFA creates its own copy in the [RegisterClient\(\)](#) call.

11.3.1 Login

In order to make requests through the OMM Consumer Event Source, the application must successfully log in to the back-end infrastructure. The Message Model used for this purpose is `MMT_LOGIN`.

The process of submitting a login request follows the general request making process outlined in Section 11.3, Process for Creating Request Message. The request attributes ApplicationID and Position are encoded as an ElementList. See Section 6.2.1.5, Encoding ElementList for details. These steps are shown in the code fragment below. The request message which is declared in this block are reused in the other requests demonstrated later in this section.

```
ReqMsg reqMsg = new ReqMsg();
reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

AttribInfo attribInfo = new AttribInfo();
attribInfo.NameType = RDM.Login.USER_ID_TYPES.USER_NAME;
attribInfo.Name = cfgVariables.UserName;

ElementList elementList = new ElementList();
ElementEntry element = new ElementEntry();
DataBuffer elementData = new DataBuffer();
ElementListWriteIterator elwiter = new ElementListWriteIterator();
elwiter.Start(elementList);

element.Name = RDM.Login.ENAME_APP_ID;
elementData.SetFromString(cfgVariables.AppId, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);

element.Name = RDM.Login.ENAME_POSITION;
elementData.SetFromString(cfgVariables.Position, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);
elwiter.Complete();
attribInfo.Attrib = elementList;
reqMsg.AttribInfo = attribInfo;
```

Example 152: Create Login Request Message

NOTE: If an application makes a request for market information (including Service Directory requests) prior to a successful login, the requests will be queued internally until a successful login response is received. RFA does not attempt to connect to the back-end infrastructure until a login request is received. There can only be one user logged in per Event Source. RFA automatically attempts to re-log in if the connection to the back-end is lost and recovered.

11.3.2 Directory

The application may wish to discover what services are available from the infrastructure. This is accomplished by specifying interest in Service Directory Events using the Message Model [MMT_DIRECTORY](#).

By default, information about all available services is returned. If an application wishes to make a request for information pertaining to a specific service only, it can use the [ServiceName](#) property of the request attribute object ([Reuters.RFA.Message.AttribInfo](#)).

The [DataMask](#) property accepts a bit mask that determines what information is returned for each service. The bit mask values for the Service Filter are defined in the *RFA Reference Manual .NET Edition* ([Reuters.RFA.RDM.Directory.SERVICE_DIRECTORY_FILTER_FLAGS](#)). The data associated with each specified bit mask value is returned in a separate Element List structure contained in a FilterEntry. The [ServiceInfo](#) Element List contains the name and capabilities of the source. The [ServiceState](#) Element List contains information related to the availability of the service.

The code fragment below shows an example of requesting a Service Directory from Consumer example. For general notes on making requests, refer to Section 11.3.

```
reqMsg.Clear();
attribInfo.Clear();

// By Setting the serviceName on the request we can request Directory information
// just for the specified service. For now, we will request entire directory.
// Uncomment the 2 following lines to request a Directory for a single service.
// RFA_String tempServiceName = new RFA_String( appConfig.ServiceName );
// attribInfo.ServiceName = tempServiceName ;

// Constructed according to the RDM Usage Guide
reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DIRECTORY;

reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
    ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

attribInfo.DataMask = appConfig.DirectoryReqFilter;
reqMsg.AttribInfo = attribInfo;
```

Example 153: Create Directory Request Message

11.3.3 Dictionary

After receiving a directory, a consumer application may request a dictionary for that directory from the provider or load dictionary from file. For more information, refer to Section 13.15.1.

11.3.4 Level 1 Data (Market Price)

After the application has initialized the Event Source and logged in, the application typically requests market information. This code fragment shows how to make a request for Level 1 market information using the Message Model [MT_MARKET_PRICE](#). The code sets the service name and item name on the request attributes, populates the InterestSpec, and calls [RegisterClient\(\)](#).

```
ReqMsg reqMsg = new ReqMsg();
AttribInfo attribInfo = new AttribInfo();
```

```
// msgModelType value is RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE
reqMsg.MsgModelType = msgModelType;
// interType value is ReqMsg.InteractionTypeFlag.InitialImage |
ReqMsg.InteractionTypeFlag.InterestAfterRefresh
reqMsg.InteractionType = interType;

attribInfo.NameType = RDM.RDM.INSTRUMENT_NAME_TYPES.INSTRUMENT_NAME_RIC;
attribInfo.Name = itemName;
attribInfo.ServiceName = serviceName;

reqMsg.AttribInfo = attribInfo;
reqMsg.IndicationMask |= indicationMask;
```

Example 154: Create Market Price Request Message

There are additional methods on the [AttribInfo](#) object to specify Interaction Type (streaming or snapshot) and requested QoS. Refer to the *RFA Reference Manual .NET Edition*.

11.3.5 Level 2 Data (Market by Order, Market by Price, Market Maker and Symbol List)

Requests for Level 2 market information are similar to the Level 1 market information request shown in Example 154, except that the following Message Models are used. For more details, see the *RFA RDM Usage Guide .NET Edition*.

MMT_MARKET_BY_ORDER	Provides access to a detailed order book for an instrument, containing individual instead of aggregated orders.
MMT_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.
MMT_MARKET_MAKER	Provides access to each individual market maker's best bid/ask quotations for an instrument.
MMT_SYMBOL_LIST	Provides access to list of symbols. This list may be for an index or could define the universe of symbols that the service provided.

Table 75: Message Model Types

11.3.6 Create a Stream for Sending Generic Message

Using Request Message to create a stream for sending Generic Message is similar to the Level 1 Data (Market Price) shown in Example 154. The application must set Message Model Type to the user-defined domain. For more information, refer to the `StarterConsumer_GenericMsg` example in the RFA package.

```
// GenericMsgModelType value is 200
reqMsg.MsgModelType = GenericMsgModelType;
```

Example 155: Set Message Model Type for Generic Message

11.3.7 Create a Stream for Sending Post Message

Using Request Message to create a stream for sending Post Message is similar to the Level 1 and Level 2 Data shown in Example 154. The application must set Message Model Type according to the message model type of Post Message. For more information, refer to the `StarterConsumer_PostMsg` example in the RFA package.

11.4 Registering for Events from OMM Consumer

After the request message is created, the application needs to register with OMMConsumer with a particular Interest Specification. To register for events, the application needs to call the `RegisterClient()` method of the OMM Consumer by passing one of the Interest Specifications. The type of events that would be received will correspond with the specific Interest Specification. The following is the activity snippet diagram for registering event.



Figure 82: Registering for Events from OMM Consumer

11.4.1 Registering for Item Events

The application opens an Event Stream by calling the `RegisterClient()` method of the OMM Consumer Event Source (`ommConsumer`). As with opening of any Event Stream, this method accepts up to four parameters. The first parameter (`eventQueue`) is a handle to the Event Queue to which the Item Event will be sent. The second parameter (`ommItemIntSpec`) is the Interest Specification. The third parameter (`this`) is the Client, which will receive the Event callback. Finally, the fourth parameter is the Closure (`itemName`) as described in section 5.1.1.6, Closures. The return value of `RegisterClient()` is a Handle to the newly opened Event Stream.

The application registers for Item Events as follows:

```
OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ommItemIntSpec.Msg = reqMsg;

handle = ommConsumer.RegisterClient(eventQueue, ommItemIntSpec, this, itemName);
```

Example 156: Register Item Event

11.4.2 Registering for Connection Events

An application may optionally register to receive connection-related Events. Because RFA automatically recovers connections and their associated Event Streams, the main value to receiving this Event type is so that the application can notify the user of the condition through its own user interface.

The application opens an Event Stream by calling the `RegisterClient()` method of the OMM Consumer Event Source (`OMMConsumer`). As with opening of any Event Stream, this method accepts up to four parameters. The first parameter (`eventQueue`) is a handle to the Event Queue to which the connection Event will be sent. The second parameter (`ommConnectionIntSpec`) is the Interest Specification. The third parameter (`this`) is the Client, which will receive the Event callback. Finally, the fourth parameter is the Closure as described in Section 5.1.1.6. The return value of `RegisterClient()` is a Handle to the newly opened Event Stream.

The application registers for Connection Status Events as follows:

```
OMMConnectionIntSpec connectionIntSpec = new OMMConnectionIntSpec();
ommConnIntSpecHandle = ommConsumer.RegisterClient(eventQueue, connectionIntSpec, this, null);
```

Example 157: Register Connection Event

11.4.3 Registering for Connection Statics Events

An application can register to receive connection statistics events that provide connection statistics. Connection statistics are the number of bytes read and written on the wire provided by RFA, periodically over duration specified by the application when registering for this event.

The application opens an event stream by using a pointer to an OMM consumer event source (`ommConsumer`) to call `RegisterClient()`, which can accept the following parameters:

- `eventQueue`: An event queue to which the connection event is sent.
- `ommConnectionStatsIntSpec`: The interest specification. The application can specify a connection name or a list of comma-separated connection names using the `ConnectionName` property from the `OMMConnectionStatsIntSpec` class, for which statistics are desired. The application can also specify the duration for which statistics are gathered using the `StatsInterval` property from `OMMConnectionStatsIntSpec` class.
- `this`: Specifies the client that receives the event callback.
- A closure parameter as described in the Common Package.

The return value of `RegisterClient()` is a handle value to the `OMMConnectionStatsIntSpec`.

The application can release the interest specification at any time because RFA retains a copy on the call to `RegisterClient()`.

In addition to registering for connection statistics events, the application must define the code to process these events whenever RFA invokes the client callback method (i.e., `ProcessEvent()`). The application defines the code to process these events by deriving from the `Client` interface and implementing the `ProcessEvent()` method. The connection statistics events are generated by RFA periodically based on the time period specified by the application on the connection statistics interest specification and provided to the application separately for each connection specified on the interest specification. The client can use the `OMMConnectionStatsEvent.ConnectionName` property to determine which connection experienced the event. If the application does not specify the duration for 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 connections listed in the `Session\connectionList` parameter.

In the following example, the application implements `ProcessEvent()` by switching on the event type and outputs information about the event. If the application were to use this client to process other types of events, it would define additional case statements. The application registers for and processes connection statistics events as follows:

```
// Register for connection statistics events
public void ProcessEvent( Reuters.RFA.Common.Event evnt )
{
    switch ( evnt.Type )
    {
        case Reuters.RFA.SessionLayer.SessionLayerEventTypeEnum.OMMConnectionStatsEvent:
        {
            OMMConnectionStatsEvent connectionStatsEvent = (OMMConnectionStatsEvent)evnt;
            System.Console.WriteLine("Received Connection statistics Event from {0} on Handle {1} with bytes Read = {2} and bytes Written = {3}",
                connectionStatsEvent.ConnectionName, connectionStatsEvent.Handle, connectionStatsEvent.BytesRead,
                connectionStatsEvent.BytesWritten);
        }
        break;
        // Process other event types...
    }
}
```

11.4.4 Registering for Error Events

Before submitting a Generic Message or a Post Message, the application can register for the error event. The error event is returned to an application whenever a Generic Message or a Post Message is submitted but then fails somewhere between the `Submit()` call and before the message is written to the transport. To receive Generic Message or Post Message submission failure events, an application must register for an `OMMErrorIntSpec`.

After registering for the `OMMErrorIntSpec` the application will receive an `OMMCmdErrorEvent` for each failure of a Generic Message or a Post Message submission. Errors are reported only if there is a failure before the message is written to the transport. Because consumers are the ones that send sessions, applications cannot register before the login request message. For more information, refer to section 11.6.3, Processing Error Event.

From `OMMCmdErrorEvent`, applications can get the `CmdID`, `OMMErrorStatus`, and closure associated with the command.

The applications register for Error Events as follows:

```
OMMErrorIntSpec ommErrorIntSpec = new OMMErrorIntSpec();
ommErrIntSpecHandle = ommConsumer.RegisterClient(eventQueue, ommErrorIntSpec, this, null);
```

Example 158: Register Error Event

For more examples, refer to `StarterConsumer_GenericeMsg` or `StarterConsumer_Post` example in the RFA package.

11.5 Dispatching Event Queue

Assuming the application has completed the previous activities, it typically next performs the **Dispatch from Event Queue** activity. Dispatch is an application action that obtains the next available Event. To dispatch from the Event Queue, the application uses the Event Queue handle and calls the `Dispatch()` method as follows:

```
while ((!CtrlBreakHandler.IsTerminated()) && (currentTime < endTime || endTime == startTime))
{
    if (eventQueue != null)
    {
        int dispatchReturn = eventQueue.Dispatch(10);
        if ((dispatchReturn == Dispatchable.DispatchReturnEnum.NothingDispatchedInActive) ||
            (dispatchReturn == Dispatchable.DispatchReturnEnum.NothingDispatchedNoActiveEventStreams))
        {
            break;
        }
    }
    currentTime = System.DateTime.Now;
}
```

Example 159: Dispatch Event Queue

In this example, the application dispatch event by calling the `Dispatch()` method. This method accept one parameter which is the timeout value. For more information, refer to section 5.2.1.1.2, Dispatch from Event Queue.

11.6 Event Processing

After the application register event and dispatch event, the application will receive event. The following activity snippet diagram depicts the relationship between registering and processing events for OMM Consumer.

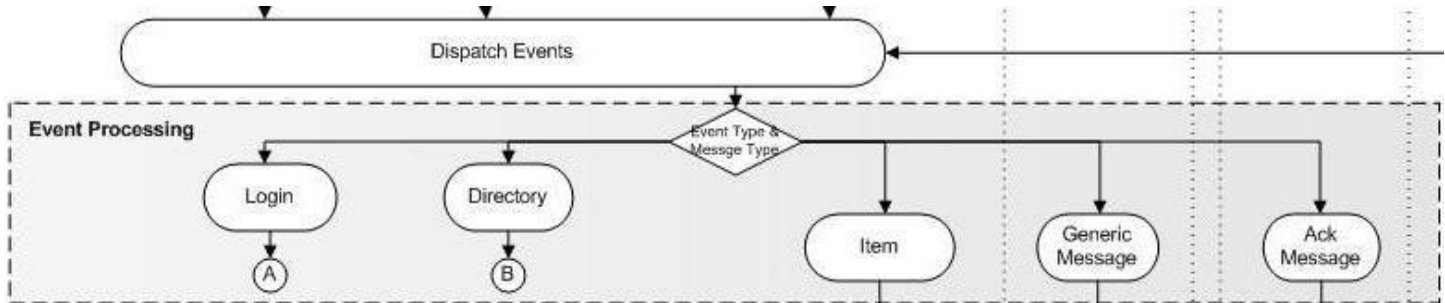


Figure 83: Event Processing

In addition to registering for Events, the application must define the code to process the Events when RFA invokes the Client call-out method (i.e., `ProcessEvent()`). The application defines the code to process these Events by deriving from the `Client` interface and implementing the `ProcessEvent()` method.

```

public void ProcessEvent(Event evt)
{
    switch (evt.Type)
    {
        case SessionLayerEventTypeEnum.ConnectionEvent:
            ProcessConnectionEvent(evt as OMMConnectionEvent);
            break;
        case SessionLayerEventTypeEnum.OMMItemEvent:
            ProcessOMMItemEvent(evt as OMMItemEvent);
            break;
        case LoggerEventTypeEnum.LoggerNotifyEvent:
            ProcessLoggerNotifyEvent(evt as LoggerNotifyEvent);
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received event with unknown event type: {0}",
                evt.Type));
            break;
    }

    if (evt.IsEventStreamClosed)
    {
        RFA_String text = OMMStrings.EventTypeToString(evt.Type);
        AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("<- Received Event Stream Closed Event, event type: {0}
            handle: {1}", text.ToString(), evt.Handle));
    }
}
  
```

Example 160: Process Item Event Callback

11.6.1 Processing Item Event

The application uses the return value of the `MsgType` property of `OMMItemEvent` to determine the message type (`RespMsg`, `GenericMsg`, `AckMsg`). If the type is `GenericMsg`, the application should downcast the return value of `OMMItemEvent`'s `Msg` property to `GenericMsg` (for details on processing Generic Messages, refer to section 11.6.5, Processing Generic Message). If the type is `AckMsg`, the application should downcast the return value of `OMMItemEvent`'s `Msg` property to `AckMsg` (for details on processing Ack Messages, refer to section 11.6.6, Processing Ack Message). If the type is `RespMsg`, the application should downcast the message to `RespMsg`.

```
private void ProcessOMMItemEvent(OMMItemEvent evnt)
{
    Msg msg = evnt.Msg;
    switch (msg.MsgType)
    {
        case MsgTypeEnum.RespMsg:
            ProcessRespMsg(evnt, msg as RespMsg);
            break;
        case MsgTypeEnum.GenericMsg:
            ProcessGenericMsg(evnt, msg as GenericMsg);
            break;
        case MsgTypeEnum.AckMsg:
            ProcessAckMsg(evnt, msg as AckMsg);
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received event with unknown message type: {0}",
                msg.MsgType));
            break;
    }
}
```

Example 161: Process Message from Item Event Callback

The application uses the return value of the received message's `MsgModelType` property to determine the Message Model (e.g. `MMT_LOGIN`, `MMT_MARKET_PRICE`). The received message's data payload and attributes may be parsed differently for each Message Model. For details, refer to the *RFA RDM Usage Guide .NET Edition*.

As with other Events, the OMM response Event (and its contained message) 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 Event (or its contained message) using the `Clone()` method. If the user chooses to make a copy of the encoded buffer of the response event without using `Clone()`, then they must ensure that the size of the buffer used is padded by 7 bytes. RFA performs optimizations during decoding, requiring the buffer size to be slightly larger than the encoded size.

Other Event types that an OMM consumer can receive in addition to `OMMItemEvent` are `ConnectionEvent`, `CompEvent`, `LoggerNotifyEvent`, and `OMMCmdErrorEvent` (related to the `Submit()` call).

11.6.2 Processing Connection Event

The application uses the return value of the `Status` property of `OMMConnectionEvent` to determine the connection state (`ConnectionStatus.StateEnum.Up` or `ConnectionStatus.StateEnum.Down`).

The `OMMConnectionEvent` interface also allows the application to receive more information related to the connection, such as active host name, port, and the version information of connected component.

```
private void ProcessConnectionEvent(OMMConnectionEvent ommConnectionEvent)
{
    ConnectionStatus aStatus = ommConnectionEvent.Status;
    RFA_String aName = ommConnectionEvent.ConnectionName;
    RFA_String logText = new RFA_String(consumerName);

    logText.Append(": <- Received Connection Event for ");
    logText.Append(aName);
    logText.Append(" : Connection ");
    logText.Append(((aStatus.State == ConnectionStatus.StateEnum.Up) ? "Up" : "Down"));
    logText.Append(" event with status code of : ");

    logText.Append(OMMStrings.ConnectionStatusCodeToString(aStatus.StatusCode));
    logText.Append("\n    ConnectedHostName : ");
    logText.Append(ommConnectionEvent.ConnectedHostName);
    logText.Append("\n    ConnectedPort : ");
    logText.Append(ommConnectionEvent.ConnectedPort);
    if (ommConnectionEvent.ConnectedComponentVersion.Count > 0)
    {
        logText.Append("\n    ConnectedComponentVersion : ");
        logText.Append(ommConnectionEvent.ConnectedComponentVersion[0]);
    }

    logText.Append("\r\n\r\n");

    AppUtil.Log(AppUtil.LEVEL.INFO, logText.ToString());
}
```

Example 162: Process Message from Connection Event Callback

For warm standby failover, RFA notifies the application when the connection switches to a new active server through connection Events. Though the connection status is still `ConnectionStatus.StateEnum.Up`, the status code is `ConnectionStatus.StateEnum.ServersSwitched`.

11.6.3 Processing Error Event

The application uses the return value of the `Staus` property of `OMMCmdErrorEvent` to determine the connection state (`OMMCmdErrorEvent.StateEnum.Failure`), connection status (`OMMCmdErrorEvent.StatusCodeEnum.None` or `OMMCmdErrorEvent.StatusCodeEnum.NoResources`), and status text.

```
private void ProcessOMMCmdErrorEvent(OMMCmdErrorEvent evnt)
{
    AppUtil.Log(AppUtil.LEVEL.ERR,
        string.Format("<- Received OMMCmdErrorEvent:\r\n    Cmd ID: {0}\r\n    State: {1}\r\n    StatusCode: {2}\r\n    StatusText: {3}\r\n",
            evnt.CmdID, evnt.Status.State, evnt.Status.StatusCode, evnt.Status.StatusText.ToString());
}
```

Example 163: Process Message from Error Event Callback

11.6.4 Processing Response Message

The application receives Response Message encapsulated in OMM Item Event objects. The application retrieves the response message from the Event and processes them based on the Message Model type.

11.6.4.1 Processing Login Response Message

Login Response Message contains the Login status. The following method outputs the Login response received.

```
private void ProcessLoginResponse(OMMItemEvent evnt, RespMsg respMsg)
{
    RespStatus status = respMsg.RespStatus;
    RFA_String text = OMMStrings.RespStatusToString(status);

    //For a Login Response examine the stream state and data state:
    //If stream state is open and data state is OK then we have a successful Login
    //If stream state is open and data state is NOT OK then we have a pending Login
    switch (respMsg.RespType)
    {
        case RespMsg.RespTypeEnum.Refresh:
            if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
            {
                if ((status.StreamState == RespStatus.StreamStateEnum.Open) &&
                    (status.DataState == RespStatus.DataStateEnum.Ok))
                {
                    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received MMT_LOGIN Refresh - Login Accepted{0}", text.ToString()));
                    ProcessLoginSuccess();
                }
                else if (status.StreamState == RespStatus.StreamStateEnum.Open)
                {
                    AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received MMT_LOGIN Refresh - Login Pending{0}", text.ToString()));
                }
                else
                {
                    AppUtil.Log(AppUtil.LEVEL.ERR, string.Format("<- Received MMT_LOGIN Refresh - Login Denied{0}", text.ToString()));
                }
            }
    }
}
```

```

        break;
    case RespMsg.RespTypeEnum.Status:
        if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
        {
            if ((status.StreamState == RespStatus.StreamStateEnum.Open) &&
                (status.DataState == RespStatus.DataStateEnum.Ok))
            {
                AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received MMT_LOGIN Refresh -
Login Accepted{0}", text.ToString()));
                ProcessLoginSuccess();
            }
            else if (status.StreamState == RespStatus.StreamStateEnum.Open)
            {
                AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received MMT_LOGIN Refresh -
Login Pending{0}", text.ToString()));
            }
            else
            {
                AppUtil.Log(AppUtil.LEVEL.ERR, string.Format("<- Received MMT_LOGIN Refresh -
Login Denied{0}", text.ToString()));
            }
        }
        else
        {
            AppUtil.Log(AppUtil.LEVEL.ERR, "<- Received MMT_LOGIN Status - No RespStatus");
        }
        break;
    case RespMsg.RespTypeEnum.Update:
        // in the future we could receive an update with a new permission profile
        AppUtil.Log(AppUtil.LEVEL.ERR, "<- Received MMT_LOGIN Update");
        break;
    default:
        AppUtil.Log(AppUtil.LEVEL.ERR, string.Format("<- Received a non-supported MMT_LOGIN
RespMsg type: {0}", respMsg.RespType));
        break;
    }
}

```

Example 164: Process Login Response Message

11.6.4.2 Processing Directory Response Message

The following method processes the directory response message, extracts the data received in the message, and decodes it.

```

private void ProcessDirectoryResponse(OMMItemEvent evnt, RespMsg respMsg)
{
    RFA_String text = new RFA_String("<- Received MMT_DIRECTORY ");
    text.Append(OMMStrings.MsgRespTypeToString(respMsg.RespType));
    if ((respMsg.IndicationMask & RespMsg.IndicationMaskFlag.RefreshComplete) != 0)
    {
        text.Append(" Complete Msg");
        if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
        {
            RespStatus status = respMsg.RespStatus;
            text.Append(OMMStrings.RespStatusToString(status));
        }
        AppUtil.Log(AppUtil.LEVEL.INFO, text.ToString());
    }
}

```

```

    }
    else
    {
        if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
        {
            RespStatus status = respMsg.RespStatus;
            text.Append(OMMStrings.RespStatusToString(status));
        }
        AppUtil.Log(AppUtil.LEVEL.INFO, text.ToString());
    }
}

```

Example 165: Process Directory Response Message

For an explanation of the decoding Response Message Payload, refer to section 6.2.2, Data Decoding.

11.6.4.3 Processing ConsumerStatus/SourceMirroring Data from the Directory Response Message

In the source directory response message created by the provider application, the Source Directory Info FilterEntry has an element '**AcceptingConsumerStatus**' that specifies whether the source application accepts Generic Message with the name '**ConsumerStatus**'. For further details, refer to the Consumer example, Provider_Interactive example and *RFA RDM Usage Guide .NET Edition*.

11.6.4.4 Processing Response Messages Containing Market Information

The following method processes the response message containing market information based on the type of the Message Model. In the following example, the data received in the message is extracted and decoded. The availability of response attributes is determined through the [AttribInfo.HintMask](#) property.

```

private void ProcessMarketPriceResponse(OMMItemEvent evnt, RespMsg respMsg)
{
    RFA_String text = new RFA_String("<- Received MMT_MARKET_PRICE ");
    text.Append(OMMStrings.MsgRespTypeToString(respMsg.RespType));
    if (evnt.Closure != null)
    {
        text.Append(" ");
        text.Append(evnt.Closure as RFA_String);
    }

    if ((respMsg.HintMask & RespMsg.HintMaskFlag.AttribInfo) != 0)
    {
        AttribInfo attribInfo = respMsg.AttribInfo;
        byte hint = attribInfo.HintMask;
        if ((hint & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            text.Append("\r\n    serviceName : ");
            text.Append(attribInfo.ServiceName);
        }
        if ((hint & AttribInfo.HintMaskFlag.Name) != 0)
        {
            text.Append("\r\n    symbolName : ");
            text.Append(attribInfo.Name);
        }
    }
    if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
    {
        RespStatus status = respMsg.RespStatus;
        text.Append(OMMStrings.RespStatusToString(status));
    }
    AppUtil.Log(AppUtil.LEVEL.TRACE, text.ToString());
}

```



```

if ((respMsg.HintMask & RespMsg.HintMaskFlag.Payload) != 0)
{
    decoder.DecodeData(respMsg.Payload);
}
}

```

Example 166: Process Market Information Response Message

Refer to Section 7.2.2.4 for an explanation of the decoding response message payload.

11.6.5 Processing Generic Message

The application receives Generic Message encapsulated in `OMMItemEvent` objects. The application retrieves the Generic Message from the Event and processes them based on the message type. If the `OMMItemEvent`'s `MsgType` property returns `GenericMsg`, then `OMMItemEvent` represents a Generic Message associated with a user defined Message Model Type (i.e., not a Refinitiv Domain Model). Generic Message can be processed as defined by the user-defined domain rules. Generic Message is supported for any existing Message Model Types defined in the . For details on processing Generic Message, refer to Section 7.2.2.7.

```

private void ProcessGenericMsg(OMMItemEvent evnt, GenericMsg genericMsg)
{
    //This application ignores events with Generic Message regardless of msg's domain
    AppUtil.Log(AppUtil.LEVEL.WARN, "<- Received unhandled OMMItemEvent with GenericMsg");
}

```

Example 167: Process Generic Message

11.6.6 Processing Ack Message

The application receives **Ack Message** encapsulated in `OMMItemEvent` objects. After confirming the event's message type by using `OMMItemEvent.Msg.MsgType` property to be equal to `AckMsg`, the application downcasts the event's message (obtained from `OMMItemEvent.Msg`) to `AckMsg` through the use of the `(AckMsg) msg` operator. For further details, refer to Section 7.2.2.6. The application uses **AckID** and **SeqNum** obtained from the `AckMsg` to associate the `AckMsg` message with the relevant Post Message.

```

private void ProcessAckMsg(OMMItemEvent evnt, AckMsg ackMsg)
{
    //This application ignores events with Ack Message regardless of msg's domain
    AppUtil.Log(AppUtil.LEVEL.WARN, "<- Received OMMItemEvent with Ack Msg");
}

```

Example 168: Process Ack Message

11.7 Process for Sending Messages using OMM Consumer

The OMM Consumer application can send Generic Message and Post Message to the provider. The following activity snippet diagram depicts the process for sending Generic Message and Post Message.

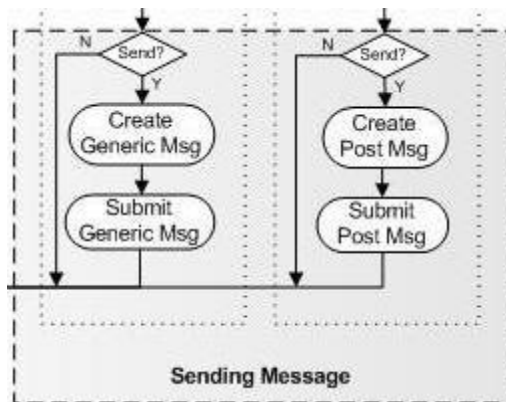


Figure 84: Process for sending Message using OMM Consumer

11.7.1 Generic Message

Generic Message can be sent on an existing stream. After the consuming application establishes a stream, to send a Generic Message the consuming application must create and submit Generic Message.

11.7.1.1 Creating Generic Message

The Generic Message is subclassed from `Reuters.RFA.Common.Msg`. To create Generic Message, the application needs to do the following:

- Set the message's model type using the `MsgModelType` property.
- Create an attribute object (`Reuters.RFA.Message.AttribInfo`).
- Populate the `AttribInfo` object with attributes. Refer to the for details regarding the meaning and applicability of each attribute for the Generic Message.
- Populate the generic message with the `AttribInfo` object using the `AttribInfo` property.
- Populate the generic message with its payload using `payload` poroperty. For details on creating `payload`, refer to the and Section 7.2.1.5.
- Populate the generic message with Indication Mask.

```

GenericMsg genericMsg = new GenericMsg();

// cfgVariables.GenericMsgModelType value is 200
genericMsg.MsgModelType = (byte)cfgVariables.GenericMsgModelType;

AttribInfo attribInfo = new AttribInfo();

attribInfo.NameType = RDM.RDM.INSTRUMENT_NAME_TYPES.INSTRUMENT_NAME_RIC;

attribInfo.Name = new RFA_String("SPEAK");

genericMsg.AttribInfo = attribInfo;

genericMsg.Payload = payLoad;

```

```
genericMsg.IndicationMask = GenericMsg.IndicationMaskFlag.MessageComplete;
```

Example 169: Create Generic Message

11.7.1.2 Submitting Generic Message

To submit Generic Message, the application needs to do the following:

- Create the `OMMHandleItemCmd`.
- Set the `Handle` property with the item handle that returns when the initial request is made establishing the stream.
- Populate `OMMHandleItemCmd` with the generic message using the `Msg` property. The `OMMHandleItemCmd` essentially acts as a wrapper around the generic message.
- Submit `HandleCmd` using the `OMMConsumer.Submit()` method.

```
OMMHandleItemCmd handleCmd = new OMMHandleItemCmd();  
  
handleCmd.Handle = itemHandle;  
  
handleCmd.Msg = genericMsg;  
  
ommConsumer.Submit(handleCmd);
```

Example 170: Submit Generic Message

11.7.2 Post Message

Post Message can be sent on any existing stream. An application can use any item stream for sending on-stream Post Message that correspond to the stream's item, and the login stream for sending off-stream Post Message that can correspond to any item, whether or not the application has an open stream for it. After the consuming application establishes a stream, to send a Post Message the consuming application must create and submit Post Message.

11.7.2.1 Creating Post Message

- The post message is subclassed from [Reuters.RFA.Common.Msg](#). To create Post Message, the application needs to do the following:
- Set the message's model type using the [MsgModelType](#) property. These can be RDM or user defined types.
- Create an attribute object ([Reuters.RFA.Message.AttribInfo](#)).
- Populate the [AttribInfo](#) object with attributes. For details regarding the meaning and applicability of each attribute for the Post Message, refer to the .
- Populate the post message with the [AttribInfo](#) object using the [AttribInfo](#) property.
- Populate the post message with its payload using the [Payload](#) poroperty. For details on creating [Payload](#), refer to Section 7.2.1.3.
- Populate the post message with Indication Mask.
- Populate the post message with PostID, Sequence number, PermissionData, and Post User Rights Mask are optional. For details on creating [PermissionData](#), refer to Section 7.2.1.3.

```
PostMsg postMsg = new PostMsg();

//This application encodes a single part post msg only on any specified domain
// domain value is RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE
postMsg.MsgModelType = domain;
postMsg.IndicationMask = PostMsg.IndicationMaskFlag.MessageInit |
PostMsg.IndicationMaskFlag.MessageComplete;

//Since attrib info is optional...
if (attribInfo != null)
{
    postMsg.AttribInfo = attribInfo;
}

//Since sequence number is optional, if value passed in is negative it is not set on post msg
if (seqNum >= 0)
{
    postMsg.SeqNum = (uint)seqNum;
}

//Since post id is optional, if value passed in negative
//It is not set on post msg, this also means that ack is not requested
if (postID >= 0)
{
    postMsg.IndicationMask = (byte)(postMsg.IndicationMask|PostMsg.IndicationMaskFlag.wantAck);
    postMsg.PostID = (uint)postID;
}

// Permission data may be included in the PostMsg for the user to be permissioned to post
// information using the PostMsg.PermissionData property.

// Set the optional Post User Rights: sets the Post User Rights Mask to Create records in a
// cache with this post or modifies permission data for records already in a cache with this post.
```

```

postMsg.PostUserRightsMask = (PostMsg.PostUserRightsMaskFlag.Create |
PostMsg.PostUserRightsMaskFlag.ModifyPermissionData);

//...

postMsg.Payload = fl;

```

Example 171: Create Post Message

11.7.2.2 Submitting Post Message

To submit Post Message, the application needs to do the following:

- Create the `OMMHandleItemCmd`.
- Set the `Handle` property with the item handle for on-stream posting, or set the `Handle` property with the login handle for off-stream posting.
- Populate `OMMHandleItemCmd` with the generic message using the `Msg` property. The `OMMHandleItemCmd` essentially acts as a wrapper around the generic message.
- Submit `HandleCmd` using the `OMMConsumer.Submit()` method.

```

OMMHandleItemCmd handleCmd = new OMMHandleItemCmd();
handleCmd.Handle = handle;
handleCmd.Msg = postMsg;

ommConsumer.Submit(handleCmd);

```

Example 172: Submit Posting Message

11.8 Unregistering Events from OMM Consumer

When the application no longer has interest in a particular entity, it may relinquish interest via the `UnregisterClient()` method on the Event Source. The flowing snippet activity diagram depicts the unregistering events from OMM Consumer.

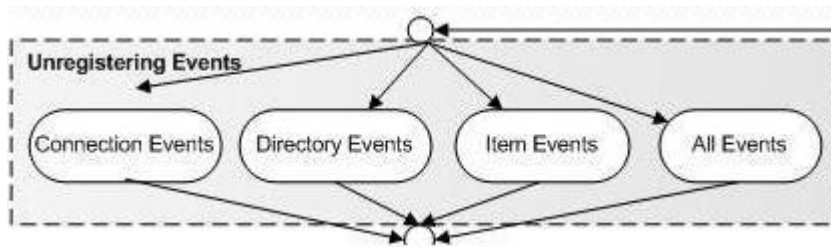


Figure 85: Unregistering Events from OMM Consumer

11.8.1 Unregister Connection Events

The following code closes the Event Stream opened for Connection Events in section 11.4.2, Registering for Connection Events.

```

if (ommConnIntSpecHandle != 0)
{
    ommConsumer.UnregisterClient(ommConnIntSpecHandle);
    ommConnIntSpecHandle = 0;
}

// ...

if (eventQueue != null)
{
    eventQueue.Destroy();
    eventQueue = null;
}
  
```

Example 173: Unregister Connection Event

In the code above, the application closes the Event Stream by calling the `UnregisterClient()` method on the Event Source (`OMMConsumer`). This method accepts one parameter - a reference to the Handle returned from the `RegisterClient()` call.

11.8.2 Unregister Market Information Events

This code closes the Event Stream which was opened for market information in section 11.4.1, Registering for Item Events.

```
// Unregister Item  
oMMConsumer.UnregisterClient( itemHandle );
```

Example 174: Unregister Market Information Event

In this example, the application closes a single Event Stream by calling the `UnregisterClient()` method on the Event Source (`oMMConsumer`). This method accepts one parameter – a reference to the Handle returned from the `RegisterClient()` call.

It is important to note that if the Event Stream had already been closed by RFA (as determinable by the `IsEventStreamClosed` property on an Event during Event processing), then the application should 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.

11.8.3 Unregister All Event Streams at Once

As an alternative to unregistering on a per-handle basis, an application may call the `UnregisterClient()` method to close all Event Streams that were opened for an OMM Consumer. Note, this will not close the login stream for the event source.

11.9 Cleaning Up

The remaining cleanup code is shown below. The only part related to the OMM Consumer is the destruction of the Event Source using the `Destroy()` method. The following activity snippet diagram depicts the sequence for cleaning up an OMM Consumer application.

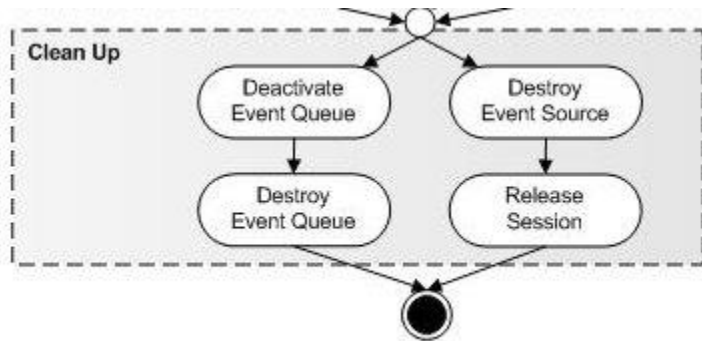


Figure 86: Cleaning Up

11.9.1 Deactivate Event Queue

After no longer using the Event Queue, the application should deactivate Event Queue via the `Deactivate()` method as follows:

```
if (eventQueue != null)
{
    eventQueue.Deactivate();
}
```

Example 175: Deactivate Event Queue

11.9.2 Destroy Event Queue

After no longer using the Event Queue, the application should destroy Event Queue via the `Destroy()` method as follows:

```
if (eventQueue != null)
{
    eventQueue.Destroy();
    eventQueue = null;
}
```

Example 176: Destroy Event Queue

11.9.3 Destroy Event Source

After no longer using the Event Source, the application should Destroy Event Queue via the `Destroy()` method as follows:

```
if (ommConsumer != null)
{
    // ...

    ommConsumer.Destroy();
    ommConsumer = null;
}
```

Example 177: Destroy Event Source

11.9.4 Release Session

After initializing a Session, the application typically next performs operations on Session Layer interfaces. These operations commonly involve access to Market Information through one or more of the various Event Sources (namely OMM Consumer, OMM Provider). Subsequent sections describe these Event Sources.

NOTE: 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.

After no longer using the Session Layer interfaces, the application should perform the **Shutdown Session** activity in Section 11.2.1 via the `Release()` method as follows:

```
if (session != null)
{
    session.Release();
    session = null;
}
```

Example 178: Release Session

Chapter 12 Implementing OMM Provider

The purpose of the OMM Provider is to provide interactive and non-interactive means of sending OMM data to OMM consumer clients. The main different between these two is Interactive Provider acts as a server in client-server relationship which waits for clients's request before sending data whereas Non-Interactive Provider acts as client to connect and send data to consumer clients. Furthermore, Non-Interactive Provider is required to send a login message to a consumer client for authorization before sending data.

12.1 Application Lifecycle

The two diagrams below illustrate the basic lifecycle of an example OMM Interactive and Non-Interactive Providers. This chapter describes about EventQueue, Session, and Event Source handling for these two providers.



Figure 87: StarterProvider_Interactive life cycle

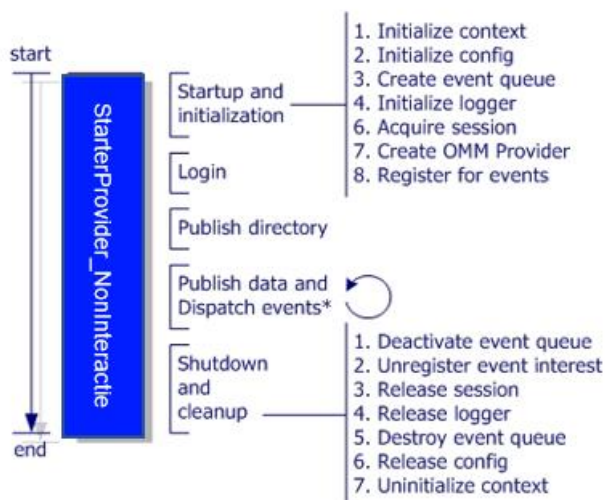


Figure 88: StarterProvider-NonInteractive Life Cycle

12.2 Interactive Provider

The following activity diagram depicts high-level activities for implementing Interactive Provider involved in initialization OMM Provider, registering events, event processing, sending messages, unregistering events, and cleaning up.

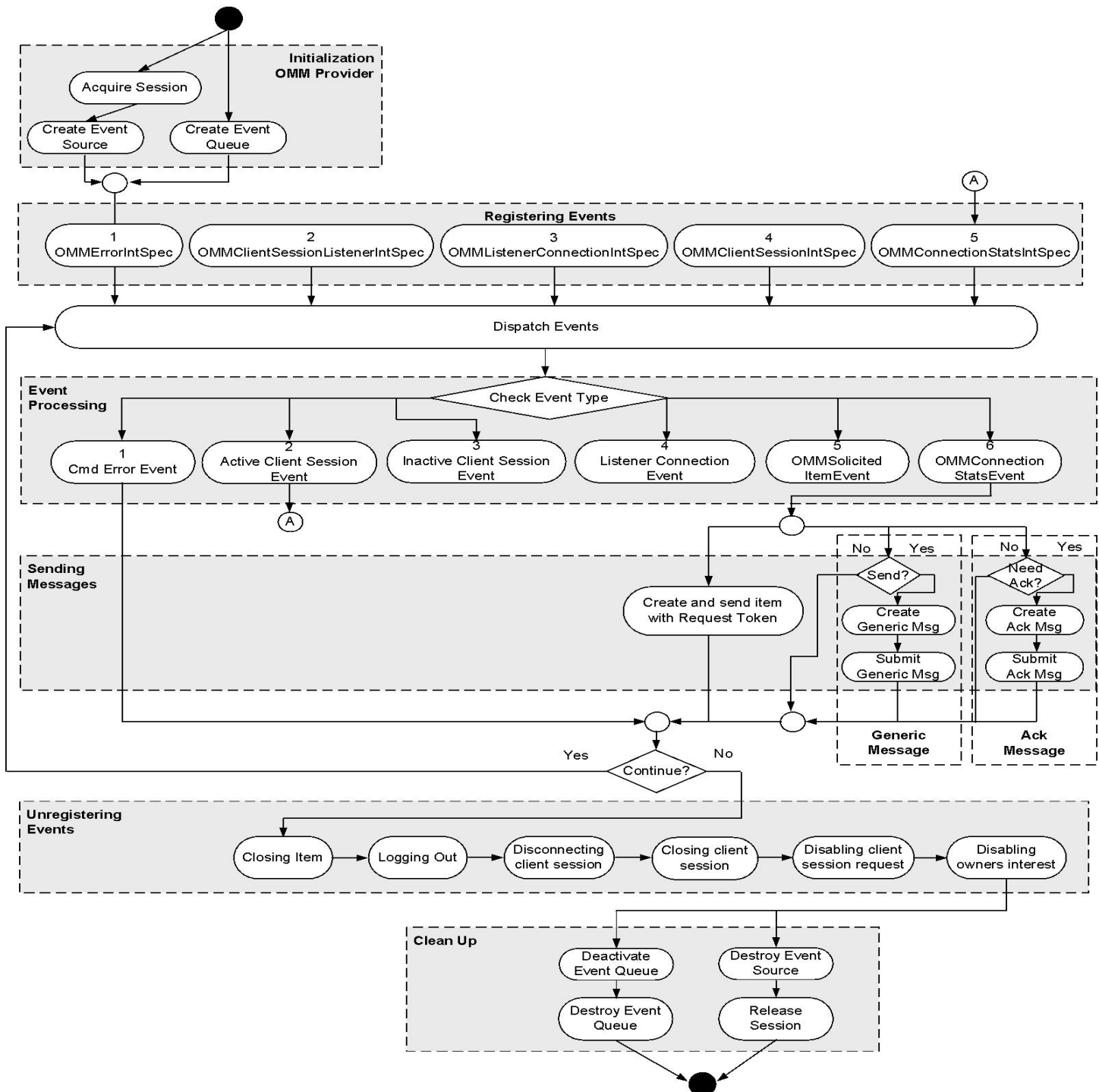


Figure 89: High level activity diagram of implementing Interactive Provider

12.2.1 Initialization OMM Provider

The initialization phase of implementing OMM provider application is to initialize a Session which is used to establish connectivity with various back-end systems. After that the session is used to create an OMMProvider event source for registering events and submitting data. Finally, an Event Queue is created for dispatching events from OMM Provider. The following is the snippet activity diagram for Initialization OMM Provider.

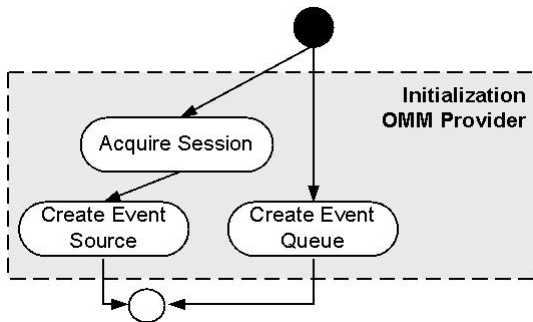


Figure 90: Initialization OMM Provider

12.2.1.1 Acquire Session

Before creating an OMMProvider, an application must acquire a session by calling the `Acquire()` static method of Session class as follows:

```
session = Session.Acquire(cfgVariables.SessionName);
```

Example 179: Initialize Session

The static `Acquire()` method requires at least one parameter, which is the session name. In this case, the application specifies the Session name of `Session1`. The Session name references the configuration information and provides identification for purposes such as logging. The static `Acquire()` method returns a Session object.

12.2.1.2 Create Event Source

To initialize an OMMProvider, the application uses the Session object and call the `CreateOMMProvider()` method as follows:

```
ommProvider = session.CreateOMMProvider(new RFA_String(appName));
```

Example 180: Initialize Event Source

The `CreateOMMProvider()` method also accepts an optional second parameter which specifies whether or not Completion Events should be sent. By default, Completion Events are not generated. The above example shows only the first parameter, which is the name of the Event Source. This name provides identification for purposes such as logging. To provide flexibility for OMM providing applications, RFA supports multiple OMM Providers per Session, as well as multiple RSSL_PROV type connections per OMM Provider. Thus a single OMM Provider can provide OMM data to OMM consuming applications over several connections.

12.2.1.3 Initializing Event Queue

To initialize Event Queue, the application calls the `Create()` static method of `EventQueue` class as follows:

```
eventQueue = EventQueue.Create(new RFA_String("myEventQueue"));
```

Example 181: Initialize Event Queue

The static `Create()` method requires at least one parameter, which is the Event Queue name. In this case, the application specifies `myEventQueue` as event queue name.. The static `Create()` method also accepts an optional second parameter which specifies whether or not turn on statistics for this Event Queue. The Event Queue name both references configuration information and provides identification for purposes such as logging. The static `Create()` method returns a Event Queue object.

12.2.2 Registering Events for Interactive Provider

The following snippet activity diagram depicts the relationship between calling `RegisterClient()` for Interest Specifications of interactive provider and the event which will receive from these registrations.

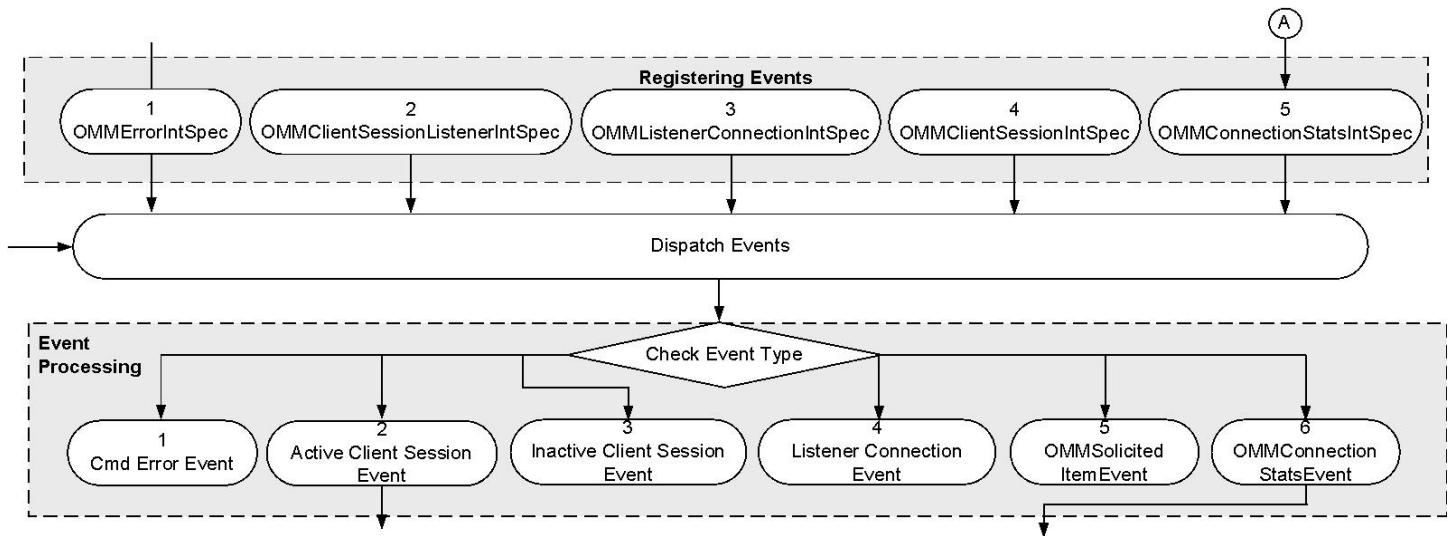


Figure 91: Registering Events for Interactive Provider

As RFA supports multiple OMM Providers per session and only one OMM Provider can accept or reject client session requests on a listening connection, the application must specify unique listener name(s) for each OMM Provider.

- **OMMClientSessionListenerIntSpec:** After OMM Provider has been created, it will not receive any inbound consumer client session connection attempts ([OMMActiveClientSessionEvent](#)). The application needs to register with the OMM Provider to open a listening port for consuming client session requests to connect. This is done by calling [RegisterClient\(\)](#) using the [OMMClientSessionListenerIntSpec](#). Once registered, the application will be able to receive client session request events. The [ListenerName](#) property of the [OMMClientSessionListenerIntSpec](#) class specifies the listening connection name(s) (an empty name, or a comma delimited list of names) in which the OMM Provider is interested. This OMM Provider is responsible for accepting or rejecting client session requests on those connections. The specified listener names must constitute a subset of all the RSSL_PROV type connections listed on the Session\connectionList config parameter.
- **OMMListenerConnectionIntSpec:** This is used to receive connection events related to problems creating the listener port(s). To ensure that the providing application receives all connection events related to the listener port(s), Interest Specification must be registered before calling [RegisterClient\(\)](#) with [OMMClientSessionListenerIntSpec](#). As above, the [ListenerName](#) property of the [OMMListenerConnectionIntSpec](#) class specifies listening connection name(s) (an empty name, or a comma delimited list of names) in which the OMM Provider is interested.

NOTE: This is different than the [OMMConnectionIntSpec](#) for the OMM Consumer. Registering for this Interest Specification can only be done once per event source.

- **OMMClientSessionIntSpec:** This is used for accepting a client session request ([OMMActiveClientSessionEvent](#)) events. Once the providing application has registered for this Interest Specification, the application will then be able to receive [OMMSolicitedItemEvent](#) coming from the consuming client session. The act of registering this officially accepts the client session request. The client session handle to be accepted is set via the [ClientSessionHandle](#) property on the [OMMClientSessionIntSpec](#).
- **OMMErrorIntSpec:** This is used to register so that the provider application can receive error events ([OMMCmdErrorEvent](#)) related to the call to [Submit\(\)](#) of the OMM Provider.
- **OMMConnectionStatsIntSpec:** This is used for receiving connection statistics events for a connection or a list of connections. The application receives the number of bytes read and written on the wire by OMM Provider periodically with [OMMConnectionStatsEvent](#).

When calling the [RegisterClient\(\)](#) method to specify interest, up to four parameters may be specified:

- The first parameter is a handle to the Event Queue to which the connection Event will be sent.
- The second parameter is the Interest Specification.
- The third parameter is the Client, which will receive the Event callback.
- The fourth parameter is the Closure as described in the section 5.1.1.6, Closures.

The return value of [RegisterClient\(\)](#) is a handle value to the newly opened Event Stream.

The application can release the Interest Specification at any time after the return from the [RegisterClient\(\)](#) because RFA retains a copy of the Interest Specification (as passed in the [RegisterClient\(\)](#) call).

12.2.2.1 Initial Steps for Registering for Events

Typically after the creating of the OMM Provider, the providing application should register for `OMMListenerConnectionIntSpec`, `OMMClientSessionListenerIntSpec` and `OMMErrorIntSpec`. This following example sets the providing application so that it can receive all event types from the OMM Provider.

```
// Register for Listener Connection events (these are event related to the server listen port)
OMMListenerConnectionIntSpec connectionIntSpec = new OMMListenerConnectionIntSpec();
ommConnIntSpecHandle = ommProvider.RegisterClient(eventQueue, connectionIntSpec, this);

// Register to receive new client session connection event (inbound clients)
OMMClientSessionListenerIntSpec intListenerSpec = new OMMClientSessionListenerIntSpec();
ommClientSessionIntSpecHandle = ommProvider.RegisterClient(eventQueue, intListenerspec, this);

// Register for Cmd Error events (These events are sent back if the Submit() call fails)
OMMErrorIntSpec ommErrorIntSpec = new OMMErrorIntSpec();
ommErrIntSpecHandle = ommProvider.RegisterClient(eventQueue, ommErrorIntSpec, this, null);
```

Example 182: Register for OMMListenerConnectionIntSpec, OMMClientSessionListenerSpec and OMMErrorIntSpec

The following example illustrates the use of two OMM Providers in one Session. Each OMM Provider, `ommProvider1` and `ommProvider2`, uses different and unique listener names. This example does not include in `StarterProvider_Interactive`.

```
// create two event sources on the same session
OMMProvider ommProvider1 = session.CreateOMMProvider( new RFA_String("myOMMProvider_1") );
OMMProvider ommProvider2 = session.CreateOMMProvider( new RFA_String("myOMMProvider_2") );

// register for all events that ommProvider1 is interested in on connection Connection_RSSL_Prov_1

// Register for Listener Connection events (these are event related to the server listen port)
OMMListenerConnectionIntSpec intConnSpec1 = new OMMListenerConnectionIntSpec();

RFA_String listenerName1 = new RFA_String("Connection_RSSL_PROV_1");
intConnSpec1.ListenerName = listenerName1;

Int64 listenerConnectionHandle1 = ommProvider1.RegisterClient( eventQueue, intConnSpec1, this );

// Register to receive new client session connection event (inbound clients)
OMMClientSessionListenerIntSpec intListenerspec1 = new OMMClientSessionListenerIntSpec();

intConnSpec1.ListenerName = listenerName1;    // same listenerName as above

Int64 listenerHandle1 = ommProvider1.RegisterClient( eventQueue, intListenerspec1, this );

// Register for Cmd Error events (These events are sent back if the Submit() call fails)
OMMErrorIntSpec intErrSpec1 = new OMMErrorIntSpec();
Int64 errHandle1 = ommProvider1.RegisterClient( eventQueue, intErrSpec1, this );

//...

// register for all events that ommProvider2 is interested in on connection Connection_RSSL_Prov_2
// and Connection_RSSL_PROV_3

// Register for Listener Connection events (these are event related to the server listen port)
OMMListenerConnectionIntSpec intConnSpec2 = new OMMListenerConnectionIntSpec();

RFA_String listenerName2 = new RFA_String("Connection_RSSL_PROV_2,Connection_RSSL_PROV_3");
```



```

intConnSpec2.ListenerName = listenerName2;

Int64 listenerConnectionHandle2 = ommProvider2.RegisterClient( eventQueue, intConnSpec2, this );

// Register to receive new client session connection event (inbound clients)
OMMClientSessionListenerIntSpec intListenerSpec2 = new OMMClientSessionListenerIntSpec();

intConnSpec2.ListenerName = listenerName2;    // same listenerName as above

Int64 listenerHandle2 = ommProvider2.RegisterClient( eventQueue, intListenerSpec2, this );

// Register for Cmd Error events (These events are sent back if the Submit() call fails)
OMMErrorIntSpec intErrSpec2 = new OMMErrorIntSpec();
Int64 errHandle2 = ommProvider2.RegisterClient( eventQueue, intErrSpec2, this );

```

Example 183: Register for different listener names of two OMMProvider

Register for `OMMConnectionStatsIntSpec` for a specific connection name or all connection names of two OMMProvider

```

// create two event sources on the same session
OMMProvider ommProvider1 = session.CreateOMMProvider( new RFA_String("myOMMProvider_1") );
OMMProvider ommProvider2 = session.CreateOMMProvider( new RFA_String("myOMMProvider_2") );

// Register to receive Connection statistics event for Connection_RSSL_Prov_1 connection name.
OMMConnectionStatsIntSpec ommConnectionStatsIntSpec1 = new OMMConnectionStatsIntSpec();

RFA_String listenerName = new RFA_String("Connection_RSSL_Prov_1");
ommConnectionStatsIntSpec1.ConnectionName = listenerName;
Int64 handle1 = ommProvider1.RegisterClient( eventQueue, ommConnectionStatsIntSpec1, this );

// Register to receive Connection statistics event for all connection of the session.
OMMConnectionStatsIntSpec ommConnectionStatsIntSpec2 = new OMMConnectionStatsIntSpec();
Int64 handle2 = ommProvider2.RegisterClient( eventQueue, ommConnectionStatsIntSpec2, this );

```

12.2.2.2 Registering to Accept a Client Session

Once the provider application has registered as in the above section, it is ready to receive new consumer client session requests. This section of code occurs after the `OMMActiveClientSessionEvent` event has been received (see Section 12.2.4.1). The code below shows how to accept a client session. The `cliSessHandle` is the handle of the client session.

```

// Create Interest Specification for this client session
OMMClientSessionIntSpec cliSessInterestSpec = new OMMClientSessionIntSpec();
cliSessInterestSpec.ClientSessionHandle = cliSessHandle;

// Accept the session
Long tmpSessHandle = ommProvider.RegisterClient(eventQueue, cliSessInterestSpec, this);

```

Example 184: Accept a client session

12.2.3 Dispatching Event Queue

Assuming that the application has completed the previous activities, typically the next activity is to perform *Dispatch from Event Queue*.

```
int nextTimerVal = 0;
System.DateTime currentTime = System.DateTime.Now;
System.DateTime startTime = currentTime;
System.DateTime endTime = currentTime.AddSeconds(cfgVariables.RunTimeInSeconds);
if (endTime <= startTime)
{
    endTime = startTime;
    AppUtil.Log(AppUtil.LEVEL.INFO,
        string.Format("{0} will attempt to dispatch msgs indefinitely!\r\n    Press Control C to Abort",
            appName));
}
else
{
    AppUtil.Log(AppUtil.LEVEL.INFO,
        string.Format("{0} will attempt to dispatch msgs for the configured time of {1} seconds and then
            Exit!\r\n    Press Control C to Abort",
            appName, (endTime - startTime).TotalSeconds));
}
while ((!CtrlBreakHandler.IsTerminated()) && (currentTime < endTime || endTime == startTime))
{
    // Process all the timers that already expired or will expire within Timer_MinimumInterval
    nextTimerVal = timer.NextTimer();
    if (nextTimerVal != -1 && nextTimerVal <= Timer.Timer_MinimumInterval)
    {
        timer.ProcessExpiredTimers();
    }

    // Dispatch until there is an event within "nextTimerVal" msecs, if nextTimerVal == -1, don't block on the
    // dispatch
    // Also, if there are items to be dispatched keep dispatching them until they are drained from the event
    // queue(s)
    if (nextTimerVal == -1)
    {
        nextTimerVal = 0;
    }
    while (0 <= eventQueue.Dispatch(nextTimerVal))
    {
        currentTime = System.DateTime.Now;
    }
}
```

Example 185: Dispatch Event Queue

In this example, the application dispatches an event by calling the `Dispatch()` method. This method accepts one parameter which is the timeout value. For more information, refer to [Section 5.2.1.1](#).

12.2.4 Event Processing for Interactive Provider

In addition to registering Events, the application must define the code to process these Events when RFA invokes the Client call-out method. The application defines the code to process these Events by deriving from the `Client` interface and implementing the `ProcessEvent()` method.

The following snippet activity diagram depicts the process of checking the event types of interactive provider before handling each event.

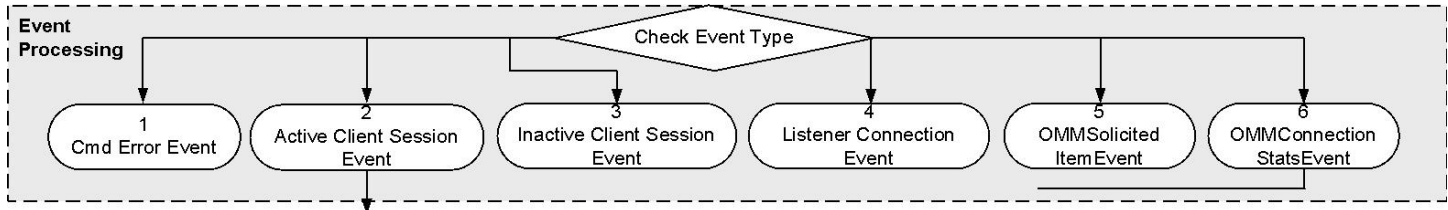


Figure 92: The event processing for interactive provider

The responsibility of Interactive Provider is to handle events which are registered to receive and Interaction types from consumer client along with Cmds to submit data back to clients. Moreover, client session handle and request token are used to handle with client's session and each request of a session handle respectively.

12.2.4.1 Handling ProcessEvent() in the application

The best practise for processing events is to separate each event handling into a function so the application can handle them separately by switching on the Event type in the application's implementation of `ProcessEvent()`. If the application uses the Client to process other types of Events, it would define additional case statements.

The implementation example of `ProcessEvent()` is shown as follows:

```

public void ProcessEvent(Event evt)
{
    switch (evt.Type)
    {
        case SessionLayerEventTypeEnum.ConnectionEvent:
            ProcessConnectionEvent(evt as OMMConnectionEvent);
            break;
        case SessionLayerEventTypeEnum.OMMSolicitedItemEvent:
            ProcessOMMSolicitedItemEvent(evt as OMMSolicitedItemEvent);
            break;
        case SessionLayerEventTypeEnum.OMMCmdErrorEvent:
            ProcessOMMCmdErrorEvent(evt as OMMCmdErrorEvent);
            break;
        case SessionLayerEventTypeEnum.OMMActiveClientSessionEvent:
            ProcessOMMActiveClientSessionEvent(evt as OMMActiveClientSessionEvent);
            break;
        case SessionLayerEventTypeEnum.OMMInactiveClientSessionEvent:
            ProcessOMMInactiveClientSessionEvent(evt as OMMInactiveClientSessionEvent);
            break;
        case LoggerEventTypeEnum.LoggerNotifyEvent:
            ProcessLoggerNotifyEvent(evt as LoggerNotifyEvent);
            break;
        case SessionLayerEventTypeEnum.OMMConnectionStatsEvent:
            ProcessOMMConnectionStatsEvent(evt as OMMConnectionStatsEvent);
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received unknown Event type: {0} handle: {1}",
                evt.Type, evt.Handle));
            break;
    }
}
  
```

```

}
if (evnt.IsEventStreamClosed)
{
    AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("<- Received Stream Closed Event type: {0} handle: {1}",
        OMMStrings.EventTypeToString(evnt.Type).ToString(), evnt.Handle));
}
}

```

Example 186: Handling ProcessEvent()

12.2.4.2 Handling Consumer Client Session Events (Requests)

After the providing application has successfully registered for Events with [OMMListenerConnectionIntSpec](#) (see section 12.2.2, Registering Events for Interactive Provider), the application can then receive consumer client session requests. At the time a client session makes an attempt to connect to the OMM Provider, this Event will become available in the applications Event queue. When [Dispatch\(\)](#) is called on the Event queue, that was specified on the [RegisterClient\(\)](#) method call, the provider's [ProcessEvent\(\)](#) method will be called with one of the following Events. Note that the client session handle is available on the [ClientSessionHandle](#) of these Events.

The client session handle will need to be maintained by the provider application, because this will be the unique identifier to represent that consumer client session. In addition, subsequent [OMMSolicitedItemEvent](#) will be received for each client session handle. The request token associated with the [OMMSolicitedItemEvent](#) will need to be associated with the client session handle specified in that event.

12.2.4.2.1 OMMActiveClientSessionEvent (New Client Session Request)

When the [OMMActiveClientSessionEvent](#) event is received, the provider applications responsibility is to either accept or reject this client session. If the provider does not respond within a specific timeout, the client session channel will be dropped by the consuming side because it did not receive a response within the timeout. The provider application can use the [OMMActiveClientSessionEvent.ListenerName](#) property to learn on which **RSSL_Prov-type** connection the event occurred.

- Accepting the Client Session Request:

The process to accept a client session is that the provider application needs to create an [OMMClientSessionIntSpec](#), populate the client session handle on the Interest Specification and call [RegisterClient\(\)](#) with that Interest Specification. The calling of the [RegisterClient\(\)](#) method will return the client session handle if it succeeds. This process completes the accepting of the client session. At this point, the next step typically will be handling of the login request. See Section 12.2.4.3, Handling Consumer Solicited Item Events: Request Message for the handling of login requests.

NOTE: There can only be one login per client session.

```

private void ProcessOMMActiveClientSessionEvent(OMMActiveClientSessionEvent ommActiveClientSessionEvent)
{
    long clientSessionHandle = ommActiveClientSessionEvent.ClientSessionHandle;

    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received OMMActiveClientSessionEvent: \r\n\tfrom
        Client located on host {0} (IP = {1})\r\n\ton Listener Port
        {2}", ommActiveClientSessionEvent.ClientHostName.ToString(),
        ommActiveClientSessionEvent.ClientIPAddress.ToString(),
        ommActiveClientSessionEvent.ListenerName.ToString()));
    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("\n    ConnectedComponentVersion : {0}\n",
        ommActiveClientSessionEvent.ClientComponentVersion));

    RFA_String reason = new RFA_String("");
    if (ShouldAcceptClientSession(reason))
    {
        ClientSessionMgr csm = new ClientSessionMgr(this, applicationContext);
        csm.Accept(clientSessionHandle);
    }
}

```

```

clientSessions.Add(csm);
if (rejectClientSession)
    AddRejectTimer(clientSessionHandle);

AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("Client Session {0} has been accepted for
    OMMAciveClientSessionEvent", clientSessionHandle));
}
else
{
    // Reject the client session
}
}
}

```

Example 187: Accept Active Client Session Event

- Rejecting the Client Session Request:

The process to reject a client session is that the provider needs to create a `OMMClientSessionCmd`, specify the client session handle, set appropriate status and call `Submit()`. Application can send `OMMClientSessionCmd` to close the client session can be sent at any time after the particular client session has been accepted.

Once the 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.

`OMMAciveClientSessionEvent` event also allows client to retrieve connected client host name, IP address and connected client component version information. The port value is not available to the OMM Provider.

```

private void ProcessOMMAciveClientSessionEvent(OMMAciveClientSessionEvent ommActiveClientSessionEvent)
{
    long clientSessionHandle = ommActiveClientSessionEvent.ClientSessionHandle;

    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received OMMAciveClientSessionEvent: \r\n\tfrom
        Client located on host {0} (IP = {1})\r\n\t on Listener Port
        {2}", ommActiveClientSessionEvent.ClientHostName.ToString(),
        ommActiveClientSessionEvent.ClientIPAddress.ToString(),
        ommActiveClientSessionEvent.ListenerName.ToString()));

    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("\n    ConnectedComponentVersion : {0}\n",
        ommActiveClientSessionEvent.ClientComponentVersion));

    RFA_String reason = new RFA_String("");
    if (ShouldAcceptClientSession(reason))
    {
        // Accept the session
    }
    else
    {
        // Reject the client session
        OMMProvider ommProvider = applicationContext.OMMProvider;
        OMMClientSessionCmd mCliSessRejectCmd = new OMMClientSessionCmd();
        mCliSessRejectCmd.ClientSessionHandle = clientSessionHandle;
        ClientSessionStatus mCliSessRejectStatus = new ClientSessionStatus();
        mCliSessRejectStatus.State = ClientSessionStatus.StateEnum.Inactive;
        mCliSessRejectStatus.StatusCode = ClientSessionStatus.StatusCodeEnum.Reject;
        mCliSessRejectCmd.Status = mCliSessRejectStatus;
        ommProvider.Submit(mCliSessRejectCmd);
        AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("Client Session {0} has been rejected because

```

```

        {1}", +clientSessionHandle, reason.ToString());
    }
}

```

Example 188: Reject Active Client Session Event

12.2.4.2.2 OMMinactiveClientSessionEvent (Client Session connection has been lost)

This Event is received to notify the provider application that the client session has gone away. The client session handle is available on the Event. The following is the responsibility of the provider application.

- Stop sending any data to that client session. The application would then be able to remove any reference to any request tokens associated with that client session.
- Remove references to the client session handle and its associated request tokens.

If an application, which is registered to receive [OMMCmdErrorEvent](#), attempts to send data to a client session after processing its [OMMinactiveClientSessionEvent](#), it may receive [OMMCmdErrorEvent](#) in response. All data sent to a closed client session will be dropped.

[OMMinactiveClientSessionEvent](#) event also allows client to retrieve disconnected client host name, IP address and client component version information. The port value is not available to the OMM Provider.

```

private void ProcessOMMinactiveClientSessionEvent(OMMinactiveClientSessionEvent evnt)
{
    RFA_String infoStr = OMMStrings.InActClientSessEventToString(evnt);
    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received OMMinactiveClientSessionEvent {0}",
        infoStr.ToString()));
    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("\n    ConnectedComponentVersion : {0}\n",
        evnt.ClientComponentVersion));

    // Get client session handle
    long clientSessionHandle = evnt.Handle;

    // remove references to the client session handle
    providerWatchList.RemoveClientWatchList(clientSessionHandle);

    // Display client session handle and corresponding request token info
    if (cfgVariables.DisplayTables)
    {
        DisplayTables();
    }
}

```

Example 189: Process Inactive Client Session Event

12.2.4.3 Handling Consumer Solicited Item Events: Request Message

The [OMMSolicitedItemEvent](#)'s [Msg](#) property returns a [Msg](#). If the [Msg](#)'s [MsgType](#) property returns [ReqMsg](#), then the [OMMSolicitedItemEvent](#) represents a request that is associated with a particular message model type. Once this event is received, a call to [MessageModelType](#) on the [ReqMsg](#) will return the message model type. This message model type will determine what to do next with the particular request. Note that all of the message model types of this type of Event are associated with a single client session. The client session handle is available via the [Handle](#) property on the Event.

Once the message model type is determined, the providing application will need to create a [RespMsg](#) populated according to the type. Specific handling of all message model types are defined in the *RFA RDM Usage Guide .NET Edition*. This document covers all parameters and encoding details for properly responding to each of the message model types.

To encode the payload for the [RespMsg](#), see the Section 7.2.1.2 for details on appropriate creation of the [RespMsg](#).

Once the `RespMsg` is properly populated, it then needs to be sent back to the consuming client session by calling `Submit()` on the OMM Provider. For more information on responding with OMM Data, refer to Section 12.2.5.1.

```
void ProcessOMMSolicitedItemEvent(OMMSolicitedItemEvent rfaEvent )
{
    Msg msg = rfaEvent.Msg;
    ReqMsg reqMsg = (ReqMsg) msg;

    if((reqMsg.InteractionType & ReqMsg.InteractionTypeFlag.InitialImage) == 0 &&
        (reqMsg.InteractionType & ReqMsg.InteractionTypeFlag.InterestAfterRefresh) == 0) // check for close
    {
        ProcessCloseReq(rfaEvent);
    }
    else
    {
        switch (msg.MsgModelType)
        {
            case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN:
            {
                System.Console.WriteLine("Login Request Event received for Client Session Handle: " +
                    rfaEvent.Handle);

                ProcessLoginReq(rfaEvent);
                _bIsLoggedIn = true;
            }
            break;

            case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DIRECTORY:
            {
                if (!_bIsLoggedIn)
                {
                    System.Console.WriteLine("Cannot process MMT_DIRECTORY request due to the failure of
                        login");
                    return;
                }
                System.Console.WriteLine("Directory Request Event received for Client Session Handle: " +
                    rfaEvent.Handle);

                ProcessDirectoryReq(rfaEvent);
            }
            break;
            case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DICTIONARY:
            {
                if (!_bIsLoggedIn)
                {
                    System.Console.WriteLine("Cannot process MMT_DICTIONARY request due to the failure of
                        login");
                    return;
                }
                System.Console.WriteLine("Dictionary Request Event received for Client Session Handle: "
                    + rfaEvent.Handle);

                ProcessDictionaryReq(rfaEvent);
            }
            break;
            case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_MARKET_PRICE:
            {
                if (!_bIsLoggedIn)
                {

```

```

        System.Console.WriteLine("Cannot process MMT_MARKET_PRICE request due to the failure of
            login");
        return;
    }
    System.Console.WriteLine("Market Price Request Event received for Client Session Handle: " +
        rfaEvent.Handle);

    ProcessMarketPriceReq(rfaEvent);
}
break;
default:
    System.Console.WriteLine("Unknown message model type received in
        ProcessOMMSolicitedItemEvent (ignoring)");
    break;
}
}
}

```

Example 190: Process OMMSolicitedItemEvent for Request Messages

Special handling for certain message model types are described below.

- Login Request:

The message model type or `MMT_LOGIN` represents a login request.

Specific information about the user is available from the `AttribInfo` in the `ReqMsg` accessible via the `AttribInfo` property. `AttribInfo` contains the name, name type, permission information, single open, etc.

The provider application is responsible for processing this information to determine whether to accept the login request.

RFA assumes default values for all attributes not specified in the provider's login response. For example, if a provider does not specify `SingleOpen` support in its login response, then RFA assumes the provider supports it (the default behavior). A provider that supports `SingleOpen` handles recovery for the client and never sends a Response Message with a streamstate of `ClosedRecover`. If the provider does attempt to send a response of `ClosedRecover`, RFA returns an error.

- Accepting Login

In the case where the login is to be accepted, the provider application will create a `RespMsg` with `RespType` and `RespStatus` set according to the *RFA RDM Usage Guide .NET Edition*. The provider application would then populate an `OMMSolicitedItemCmd` with this `RespMsg`, set the corresponding request token and call `Submit()` on the OMM Provider.

- Rejecting Login

In the case the provider application needs to reject the login, the provider application will create a `RespMsg` as above, but set the `RespType` and `RespStatus` to the reject semantics specified in the *RFA RDM Usage Guide .NET Edition*. The provider application would then populate an `OMMSolicitedItemCmd` with this `RespMsg`, set the corresponding request token and call `Submit()` on the OMM Provider.

Once the provider application determines that the login is to be logged out (rejected), it is the responsibility of the provider application to clean up all references to request tokens for that particular client session. In addition any incoming requests that may be received after the login reject has been submitted can be ignored.

NOTE: The provider application can reject a login at any time after it has accepted a particular login.

- Directory Request:

The message model type or `MMT_DIRECTORY` represents a directory request.

The Service Directory provides information about all available services in the system. A Consumer typically requests a Directory from a Provider to retrieve information about available services and their capabilities, and it is the responsibility of the Provider to encode and supply the directory. This includes information about supported domain types, the service's state, QoS, and any item group information associated with the service. Interactive Providers should generally supply, at a minimum, the Info, State, and Group filters for the Directory.

The Info filter contains service name for each available service. RFA will automatically populate the service ID.³³ The Provider should populate this with information specific to the service(s) it provides.

- The State filter contains status information for the service. This informs the Consumer whether the service is Up (available) or Down (unavailable).
- The Group filter conveys item group status information, including information about group states, as well as the merging of groups. If a provider determines that a group of items is no longer available, it can convey this information by sending either individual item status messages (for each affected stream) or a Directory message containing the item group status information.

For more information about encoding service directories, see `Provider_Interactive` example distributed in the RFA package.

³³ It is recommended that applications do not attempt to set their own service IDs.

12.2.4.4 Handling Consumer SolicitedItemEvents: Generic Messages

The `OMMSolicitedItemEvent`'s `Msg` property returns a `Msg`. If the `Msg`'s `MsgType` property returns `GenericMsg`, then the `OMMSolicitedItemEvent` represents a Generic Message associated with a message model type.

An OMM Provider application receives a consumer connection status `GenericMsg` on a `MMT_LOGIN` domain to indicate whether it should operate as an active or a standby server. For more details on Generic Message in the warm standby feature, refer to section 13.12, Warm Standby.

```
private void ProcessOMMSolicitedItemEvent(OMMSolicitedItemEvent evnt)
{
    Msg msg = evnt.Msg;
    switch (msg.MsgType)
    {
        case Message.MsgTypeEnum.GenericMsg:
            ProcessGenericMsg(evnt);
            break;
        case Message.MsgTypeEnum.ReqMsg:
            ProcessReqMsg(evnt);
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received unhandled OMMSolicitedItemEvent with
                msgType: {0}", OMMStrings.MsgTypeToString(msg.MsgType).ToString()));
            break;
    }
}
```

Example 191: Process OMMSolicitedItemEvent for Generic Messages

12.2.4.5 Handling Consumer SolicitedItemEvents: Post Message

The application receives `PostMsg` encapsulated in `OMMSolicitedItemEvent` objects. After confirming that the event's message type is equal to `PostMsgEnum` (by using `OMMSolicitedItemEvent.Msg.MsgType`), the application downcasts the event's message (obtained from `OMMSolicitedItemEvent.Msg`) to `PostMsg` by using the `(PostMsg)msg` operator. For further details, refer to Section 7.2.2.2, Decoding Message Type. The Application uses the `PostID`, `SeqNum`, and `MessageModelType` obtained from the `PostMessage` to associate the `AckMsg`, if requested, with this Post Message.

```
private void ProcessOMMSolicitedItemEvent(OMMSolicitedItemEvent evnt)
{
    Msg msg = evnt.Msg;
    switch (msg.MsgType)
    {
        case Message.MsgTypeEnum.GenericMsg:
            ProcessGenericMsg(evnt);
            break;
        case Message.MsgTypeEnum.ReqMsg:
            ProcessReqMsg(evnt);
            break;
        case Message.MsgTypeEnum.PostMsg:
            ProcessPostMsg(evnt);
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received unhandled OMMSolicitedItemEvent with
                msgType: {0}", OMMStrings.MsgTypeToString(msg.MsgType).ToString()));

            break;
    }
}
```

}

Example 192: Process OMMSolicitedItemEvent for Post Message**12.2.4.6 Handling CmdError Events**

The `OMMCmdErrorEvent` represents an error Event that is generated during the `Submit()` call on the OMM Provider.

This Event gives the provider application access to the `Cmd`, `CmdID`, `closure` and `OMMErrorStatus` for the `Cmd` that failed. These can be accessed via `Cmd`, `CmdID`, `SubmitClosure` and `Status` respectively.

No Response Message is required from the provider application for this Event.

```
private void ProcessOMMCmdErrorEvent(OMMCmdErrorEvent evnt)
{
    AppUtil.Log(AppUtil.LEVEL.ERR,
        string.Format("<- Received OMMCmdErrorEvent:\r\n    Cmd ID: {0}\r\n    State: {1}\r\n    StatusCode: {2}\r\n    StatusText: {3}\r\n",
            evnt.CmdID, evnt.Status.State, evnt.Status.StatusCode, evnt.Status.StatusText.ToString()));
}
```

Example 193: Handling CmdError Events**12.2.4.7 Handling Connection Events**

For the OMM Provider, the `OMMConnectionEvent` provides information with respect to the listener connection. This will contain connection name and type information along with the `ConnectionStatus` of the event. This information can be determined if the listener port is up or down. A provider application can use the `OMMConnectionEvent` interface to retrieve connected consumer client information such as client host name, component version information.

No Response Message is required from the provider application for this Event.

NOTE: Listener connection events are not associated with the inbound consumer client sessions.

```
private void ProcessConnectionEvent(OMMConnectionEvent OmmConnectionEvent)
{
    ConnectionStatus connectionStatus = ommConnectionEvent.Status;
    RFA_String text = new RFA_String();
    RFA_String port = new RFA_String();
    if (GetPort(port, ommConnectionEvent.ConnectionName))
    {
        text.Append("\r\n    Listening on Port: ");
        text.Append(port);
    }

    if (connectionStatus.State == ConnectionStatus.StateEnum.Up)
    {
        text.Append("<- Received ConnectionEvent -----Connection Up!----- ");
    }
    else if (connectionStatus.State == ConnectionStatus.StateEnum.Down)
    {
        text.Append("<- Received ConnectionEvent -----Connection Down!----- ");
    }
    text.Append("\r\n    ConnectionStatusText: ");
    text.Append(connectionStatus.StatusText);
    text.Append("\r\n    ConnectionStatusCode: ");
    text.Append(connectionStatus.StatusCode.ToString());
    text.Append("\r\n    ");
}
```

```

text.Append(ommConnectionEvent.ConnectionName);
AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("{0}\r\n", text.ToString()));
}

```

Example 194: Handling Connection Events

12.2.4.8 Handling LoggerNotify Events

As with all of the Event Sources, logger Events can be received as well. For more information on handling logger events, refer to Chapter 10, "Logger Package."

No Response Message is required from the provider application for this Event.

12.2.4.9 Handling Completion Events

As with all of the Event Sources, completion Events can be received as well. See Section 5.2.1, Event Distribution Model for more information on handling completion Events. Because the outbound architecture of the OMM Provider is synchronous, only the unregistering of client ([UnregisterClient\(\)](#)) call can generate the completion Event.

No Response Message is required from the provider application for this Event.

12.2.4.10 Handling Connection Statistics Events

The [OMMConnectionStatsEvent](#) provides information on statistics of a connection. The statistics of a connection are bytes read and written on the wire provided by RFA, periodically over a time period specified by the application while registering to this event.

```

/ Register for connection statistics events
public void ProcessEvent( Reuters.RFA.Common.Event evnt )
{
    switch ( evnt.Type )
    {
        case Reuters.RFA.SessionLayer.SessionLayerEventTypeEnum.OMMConnectionStatsEvent:
        {
            OMMConnectionStatsEvent connectionStatsEvent = (OMMConnectionStatsEvent)evnt;
            System.Console.WriteLine("Received Connection statistics Event from {0} on Handle {1} with bytes Read = {2} and bytes Written = {3}",
                connectionStatsEvent.ConnectionName, connectionStatsEvent.Handle, connectionStatsEvent.BytesRead,
                connectionStatsEvent.BytesWritten);
        }
        break;
        // Process other event types...
    }
}

```

12.2.5 Sending Messages for Interactive Provider

Interactive provider needs to check for message types in order to client's request correctly. The following snippet activity diagram depicts the relation between processing `OMMSolicitedItemEvent` and sending messages for interactive provider.

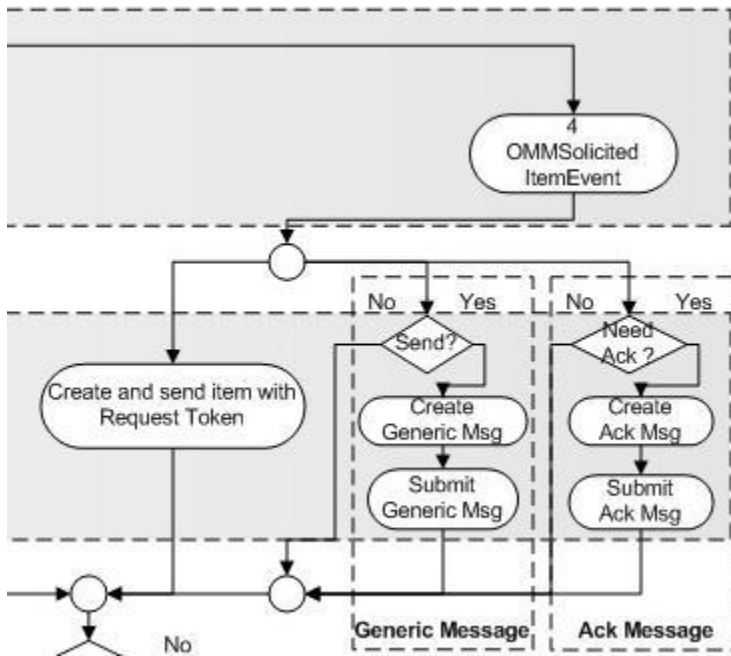


Figure 93: Relationship between processing `OMMSolicitedItemEvent` and sending messages for interactive provider

12.2.5.1 Response Message

Sending a Response Message to the consumer client session involves processing as follows:

1. Create a Response Message (`Reuters.RFA.Message.RespMsg`). The response message is subclassed from `Reuters.RFA.Common.Msg`.
2. Set the message model type of the response using the `MsgModelType` property. Possible values are defined in `Reuters.RFA.RDM.RDM.MESSAGE_MODEL_TYPES`. See the *RFA RDM Usage Guide .NET Edition* for details.
3. Set response type by using the `RespType` property. Possible values are defined in `Reuters.RFA.Message.RespMsg.RespTypeEnum`.
4. Create or re-use a request attribute object (`Reuters.RFA.Message.AttribInfo`). In many cases the `AttribInfo` from the `ReqMsg` can be used depending upon the contents of the `AttribInfo` and what the provider application needs to send. For details, refer to the *RFA RDM Usage Guide .NET Edition*.
5. Populate the response message with the `AttribInfo` object using the `AttribInfo` property.
6. For details on appropriate creation of the `RespMsg`, refer to Section 7.2.1.2.
7. Create the `cmd` and populate it with the response message using the `Msg` property. The `cmd` essentially acts as a wrapper around the response message.
8. Call the `Submit()` method of the OMM Provider Event Source, passing in the Event Queue to which responses should be sent, the Interest Specification described in the previous bullet, the client whose callback will receive the responses (refer to Section 9.1.3), and the closure. The returned handle has to be retained and will be used for closing the new Event Stream.

The `Cmd(OMMSolicitedItemCmd` or `OMMClientSessionCmd`), Response Message and response attribute objects may be reused for making subsequent responses.

NOTE: The concepts of "Update" and "Snapshot" has been changed somewhat to handle the OMM. This is because the initial refresh image might not be sent in a single `Submit()` call (i.e., when the refresh image is very large and the application does not want to consume the requisite CPU and other resources for extended amounts of time to send a single image). By splitting up the refresh, the providing application can continue to process other requests (inbound and outbound) in a timely manner.

```
private bool SubmitMsg()
{
    bool result = false;
    OMMSolicitedItemCmd itemCmd = new OMMSolicitedItemCmd();
    List<long> clientSessions = new List<long>();
    providerWatchList.GetClientSessions(clientSessions);

    for (int pos = 0; pos < clientSessions.Count; pos++)
    {
        long clientSessionHandle = clientSessions[pos];
        ClientWatchList cw1 = providerWatchList.GetClientWatchList(clientSessionHandle);

        // skip on error condition or if there are no tokens
        if ((cw1 == null) || (cw1.Size() == 0))
        {
            continue;
        }

        ClientWatchList.TokenInfo TS = null;
        RFA_String reason = new RFA_String();
        for (int i = 0; i < cw1.Size(); i++)
        {
            TS = cw1.GetTokenInfo(i);

            // need to skip login request token!
            if (TS.isItemRequest)
            {
                reason.Set("Solicited MMT_MARKET_PRICE ");
                RespStatus status = new RespStatus();
                if (!TS.isSubmitted)
                {
                    // send refresh if it was not submitted yet
                    status.StreamState = RespStatus.StreamStateEnum.Open;
                    status.DataState = RespStatus.DataStateEnum.Ok;
                    status.StatusCode = RespStatus.StatusCodeEnum.None;

                    status.StatusText = new RFA_String("Solicited Refresh Completed");

                    SetMarketPriceMsg(RespMsg.RespTypeEnum.Refresh, TS.attribInfo, status,
clientSessionHandle, true);

                    itemCmd.Msg = respMsg;
                    itemCmd.RequestToken = TS.requestToken;
                    reason.Append("Refresh");

                    result = SubmitCmd(itemCmd, null, reason);
                    TS.isSubmitted = true;

                    //Update information in structure after set the submit flag
                    cw1.UpdateToken(cw1.GetRequestToken(i), TS);
                }
            }
        }
    }
}
```

```

        else
        {
            // send update if it was not paused yet
            SetMarketPriceMsg(RespMsg.RespTypeEnum.Update, TS.attribInfo, status,
                             clientSessionHandle, TS.isAttribInfoInUpdates);
            itemCmd.Msg = respMsg;
            itemCmd.RequestToken = TS.requestToken;
            reason.Append("Update");

            result = SubmitCmd(itemCmd, null, reason);
        }
    }
}
return result;
}

```

Example 195: Submit Response Message

12.2.5.2 Generic Message

An interactive OMM Provider can send Generic Message(s) by using the currently available `OMMSolicitedItemCmd` and `Submit(..)` methods. To send a `GenericMsg` the interactive provider would encode the message, create an `OMMSolicitedItemCmd`, and pass corresponding stream's `RequestToken` on the `OMMSolicitedItemCmd`, and pass the `OMMSolicitedItemCmd` to the `Submit(..)` function.

```

private void ProcessGenericMsg(OMMSolicitedItemEvent evnt)
{
    GenericMsg genMsg = evnt.Msg as GenericMsg;
    RequestToken reqToken = evnt.RequestToken;
    long CSH = reqToken.Handle;

    OMMSolicitedItemCmd solItemCmd = new OMMSolicitedItemCmd();
    if (genericMsg == null)
    {
        genericMsg = new GenericMsg();
    }

    genericMsg.Clear();
    if ((genMsg.HintMask & GenericMsg.HintMaskFlag.AttribInfo) != 0)
    {
        RFA_String name = new RFA_String("\r\n    name    : ");
        RFA_String service = new RFA_String("\r\n    service : ");
        AttribInfo atInfo = genMsg.AttribInfo;
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.Name) != 0)
        {
            name.Append(atInfo.Name);
        }
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.ServiceName) != 0)
        {
            service.Append(atInfo.ServiceName);
        }
        AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received OMMSolicitedItemEvent with Generic Message {0}{1}{2}",
            OMMStrings.SolicitedItemEventToString(evnt).ToString(), name.ToString(), service.ToString()));
        if ((atInfo.HintMask & AttribInfo.HintMaskFlag.Attrib) != 0)
        {
            decoder.DecodeData(genMsg.AttribInfo.Attrib);
        }
    }
}

```

```

    }
    genericMsg.AttribInfo = (genMsg.AttribInfo);
}
else
{
    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("<- Received OMMSolicitedItemEvent with Generic Message
        {0}", OMMStrings.SolicitedItemEventToString(evt).ToString()));
}
if ((genMsg.HintMask & GenericMsg.HintMaskFlag.Payload) != 0)
{
    decoder.DecodeData(genericMsg.Payload);
}
else
{
    AppUtil.Log(AppUtil.LEVEL.WARN, "No payload in received Generic Msg");
}

ElementList payLoad = new ElementList();
payLoad.SetAssociatedMetaInfo(reqToken.Handle);
ElementListWriteIterator iter = new ElementListWriteIterator();
ElementEntry entry = new ElementEntry();
DataBuffer buffer = new DataBuffer();
RFA_String ename = new RFA_String();
RFA_String val = new RFA_String();

iter.Start(payLoad);
ename.Set("Data");
val.Set("Provider's <GENERIC> Message");
buffer.SetFromString(val, DataBuffer.DataBufferEnum.StringAscii);
entry.Name = ename;
entry.Data = buffer;
iter.Bind(entry);
iter.Complete();

genericMsg.Payload = payLoad;
genericMsg.MsgModelType = (byte)(cfgvariables.GenericMsgModelType);
genericMsg.IndicationMask = GenericMsg.IndicationMaskFlag.MessageComplete;

solItemCmd.Msg = genericMsg;
solItemCmd.RequestToken = reqToken;
SubmitCmd(solItemCmd, null, new RFA_String("GENERIC_MESSAGE"));
}

private bool SubmitCmd(OMMSolicitedItemCmd ommSolicitedCmd, object closure, RFA_String reason)
{
    uint cmdId = 0;
    RFA_String infoStr = OMMStrings.SolicitedItemCmdToString(ommSolicitedCmd);
    cmdId = ommProvider.Submit(ommSolicitedCmd, closure);
    AppUtil.Log(AppUtil.LEVEL.TRACE, string.Format("-> Submit {0} {1} cmdId: {2}", reason, infoStr.ToString(),
        cmdId));
    return true;
}

```

Example 196: Submit Generic Message

12.2.5.3 Ack Message

Ack Message must be sent on the same stream as the one on which the post message it is acknowledging was received. To send the ack message, a providing application must:

- Create an Ack Message (`Reuters.RFA.Message.AckMsg`). The ack message is subclassed from `Reuters.RFA.Common.Msg`.
- Set the message's model type using the `MsgModelType` property. These can be RDM or a user-defined type.
- Create, if needed, an attribute object (`Reuters.RFA.Message.AttribInfo`).
- Populate the `AttribInfo` object with attributes. For details regarding the meaning and applicability of each attribute for the ack message, refer to the *RFA RDM Usage Guide .NET Edition*.
- Populate the ack message with the `AttribInfo` object using the `AttribInfo` property.
- Create, if needed, the ack message's payload.
- Populate the ack message with its payload using `Payload` property.
- Populate the ack message with `AckID` which should match the `PostID`, `SeqNum`.
- Create the `OMMSolicitedItemCmd itemCmd` and populate it with the ack message using the `Msg` property.
- Populate `itemCmd` with the item's request token.
- Submit `itemCmd` using `OMMProvider.Submit()` method.

```
void SubmitAckMsg(RequestToken requestToken, long handle, bool postIdSet, UInt32 ackID, bool sequenceNumSet,
    uint sequenceNum, bool nackCodeSet, byte nackCode, RFA_String text, byte msgModelType, AttribInfo
    postAttribInfo)
{
    AckMsg ackMsg = new AckMsg();
    ackMsg.AckID = ackID;

    // Client app is required to set msg Model type of the posted message.
    ackMsg.MsgModelType = msgModelType;

    if (sequenceNumSet)
    {
        ackMsg.SeqNum = sequenceNum;
    }

    if (nackCodeSet)
    {
        ackMsg.NackCode = nackCode;
    }

    if (!text.Empty())
    {
        ackMsg.Text = text;
    }

    // Set serviceId need to be set in case the AckMsg could have been from
    // offstream posting
    AttribInfo ackAttribInfo = new AttribInfo();

    if ((postAttribInfo.HintMask & AttribInfo.HintMaskFlag.ServiceID) != 0)
        ackAttribInfo.ServiceID = postAttribInfo.ServiceID;
    else if ((postAttribInfo.HintMask & AttribInfo.HintMaskFlag.ServiceName) != 0)
        ackAttribInfo.ServiceName = postAttribInfo.ServiceName;

    ackMsg.AttribInfo = ackAttribInfo;

    ackMsg.SetAssociatedMetaInfo(handle);
}
```

```

// Create a OMMSolicitedItemCmd instance
OMMSolicitedItemCmd itemCmd = new OMMSolicitedItemCmd();
itemCmd.Msg = ackMsg;
itemCmd.RequestToken = requestToken;
ommProvider.Submit(itemCmd);
}

```

Example 197: Submit Ack Message

12.2.6 Unregistering Events for Interactive Provider

Cleanup of Entities is generally done in the opposite order in which they were created/initialized. The following sequence should be done sequentially for each client session so that any submitting of responses to that client session is not sent in-between. The following snippet activity digram depicts the unregister events for interactive provider.

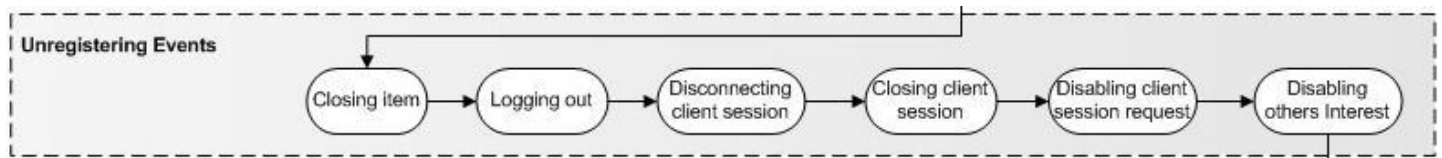


Figure 94: Unregistering processes for interactive provider

12.2.6.1 Closing items and Handling Closes

12.2.6.1.1 Closing Items

In case that the Interactive Provider application needs to close an item, the application needs to send an `OMMSolicitedItemCmd` with closed status for a specific request token. The provider application then needs to make sure that no further data specifying that request token is being sent. The particular request token needs to be discarded.

The following is an example of closing an item.

```

OMMSolicitedItemCmd cmd = new OMMSolicitedItemCmd();
RespMsg msg = new RespMsg();

// Create Status
RespStatus rStatus = new RespStatus();
rStatus.StreamState = RespStatus.StreamStateEnum.Closed;
rStatus.DataState = RespStatus.DataStateEnum.Unspecified;
rStatus.StatusCode = RespStatus.StatusCodeEnum.None;
rStatus.StatusText = new RFA_String("Closing item...");
msg.RespStatus = rStatus;

msg.MsgModelType = messageModelType; // message model type of this item
msg.RespType = RespMsg.RespTypeEnum.Status;
cmd.Msg = msg;
cmd.RequestToken = requestToken; // request token of this item
ommProvider.Submit(cmd);

```

Example 198: Closing items

12.2.6.1.2 Handling a Login Close

If the connected consumer issues a login close, the providing application will receive an `OMMSolicitedItemEvent` for login with no interactionType flags (close). The provider application will then receive an `OMMInactiveClientSessionEvent` to signal that the consumer has disconnected.

After receiving the `OMMSolicitedItemEvent` for the login (with no `interactionType` flags (close)), the provider application needs to make sure that no further data specifying any of the request tokens associated with that login is sent. All associated request tokens need to be discarded. The provider application needs to discard the associated request tokens on the first event, either login close event or inactive client session event.

See section **Error! Reference source not found., Error! Reference source not found.** for how to handle the `OMMInactiveClientSessionEvent`.

12.2.6.1.3 Handling Item Close

In the case where the connected consumer closes a particular item, the providing application will then receive an `OMMSolicitedItemEvent` with no `interactionType` flags (close). The provider application then needs to make sure that no further data specifying that request token is sent. That particular request token needs to be discarded.

12.2.6.2 Logging out a Client Session

At any time after the client session has been accepted, the provider application has the ability to logout any client session. This is done in the same manner as shown in in Section Rejecting Login.

In the case where the application is shutting down this should be done for each client session.

12.2.6.3 Disconnecting a Client Session

In the event that the provider application wants to disconnect a client session, the application needs to `Submit()` an `OMMClientSessionCmd` specifying the client session handle to be disconnected and appropriate `ClientSessionStatus`.

NOTE: A provider application must not unregister a client session after closing down or rejecting the client session as the session handle is no longer active. Doing so results in an invalid usage exception.

```
// Reject the client session

OMMClientSessionCmd mCliSessRejectCmd = new OMMClientSessionCmd();
mCliSessRejectCmd.ClientSessionHandle = cliSessHandle;    // client session handle

// set the status appropriately
ClientSessionStatus mCliSessRejectStatus = new ClientSessionStatus();
mCliSessRejectStatus.State = ClientSessionStatus.StateEnum.Inactive;
mCliSessRejectStatus.StatusCode = ClientSessionStatus.StatusCodeEnum.Reject;
mCliSessRejectCmd.Status = mCliSessRejectStatus;

// submit to OMM Provider
ommProvider.Submit(mCliSessRejectCmd);
```

Example 199: Disconnecting a client session

12.2.6.4 Closing a Client Session

The process of unregistering the client session will disable the provider application from receiving anymore Request Message for that particular client session. This is done by calling `UnregisterClient()` on the OMM Provider specifying the client session handle.

NOTE: Do not call this process after closing a client session. Doing so results in an invalid usage exception.

Warning! All request tokens associated with this client session handle must no longer be referenced by OMM Provider. Additionally, calling `UnregisterClient(null)` is not supported.

Calling this method by itself leaves RFA in an incomplete state.

```
// Unregister for client session
ommProvider.UnregisterClient(ommClientSessionHandle);
```

Example 200: Closing a client session

12.2.6.5 Disabling a Client Session Request

When the application no longer has interest in receiving any more client session requests, it may relinquish interest via the `UnregisterClient()` methods specifying the handle returned from the `RegisterClient()` with the `OMMClientSessionListenerIntSpec` (see section 12.2.2.1, Initial Steps for Registering for Events).

```
// Unregister for ClientSessionListener connection events (inbound Consumer Client Session Requests)
ommProvider.UnregisterClient(ommClientSessionListenerIntSpecHandle);
```

Example 201: Disabling a client session request

12.2.6.6 Disabling Other Intest Spec

Unregistering for connection, logger and `CmdError` Events are done similarly to the above except that the corresponding handle should be passed in.

12.2.7 Cleaning up

The remaining cleanup code is shown below. The only part related to the OMM Provider is the destruction of the Event Source using the `Destroy()` method. The following snippet activity diagram depicts the sequence for cleaning up an OMM provider application.

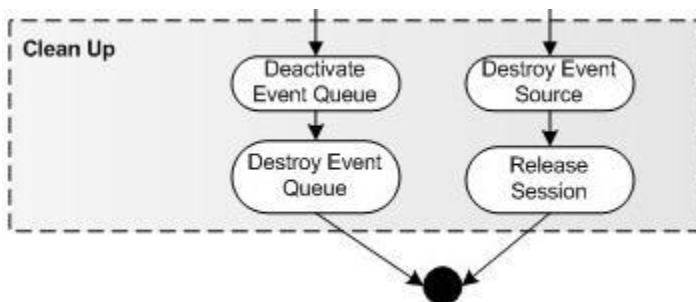


Figure 95: Cleaning up OMM Provider

NOTE: 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.

```
// cleanup provider
if (ommProvider != null)
{
    ommProvider.Destroy();
    ommProvider = null;
}

// Release Configuration Database
if (appStgCfgDataBase != null)
{
    appStgCfgDataBase.Destroy();
    appStgCfgDataBase = null;
}
if (appCfgDataBase != null)
{
    appCfgDataBase.Release();
}
```

```

    appCfgDataBase = null;
}

// force the event queue not send out any events
if (eventQueue != null)
{
    eventQueue.Deactivate();
}

// Release Session
if (session != null)
{
    session.Release();
    session = null;
}

// Unregister Logger Client and release application logger
if ((appLoggerMonitor != null) && (loggerHandle != 0))
{
    appLoggerMonitor.UnregisterLoggerClient(loggerHandle);
}
if (appLoggerMonitor != null)
{
    appLoggerMonitor.Destroy();
    appLoggerMonitor = null;
}
if (appLogger != null)
{
    appLogger.Release();
    appLogger = null;
}

// Destroy the EventQueue
if (eventQueue != null)
{
    eventQueue.Destroy();
    eventQueue = null;
}

// Release Configuration Database
if (stgCfgDataBase != null)
{
    stgCfgDataBase.Destroy();
    stgCfgDataBase = null;
}

if (cfgDataBase != null)
{
    cfgDataBase.Release();
    cfgDataBase = null;
}

if (!Context.Uninitialize() && Context.InitializedCount == 0)
{
    // Report problem if and ONLY if Context.Uninitialize() returned false
    // AND Context.InitializedCount returns zero;

    // If Context.InitializedCount returns more than zero,
    // RFA did not attempt to uninitialize its context
    AppUtil.Log(AppUtil.LEVEL.ERR, "RFA Context fails to uninitialize.");
}

```

```
}
```

Example 202: Cleaning up RFA resources

12.3 Non-Interactive Provider

The following activity diagram depicts high-level activities for implementing Non-Interactive Provider involved in initialization OMM Provider, creating Request Message, registering events, event processing, sending messages, unregistering events, and clean up.

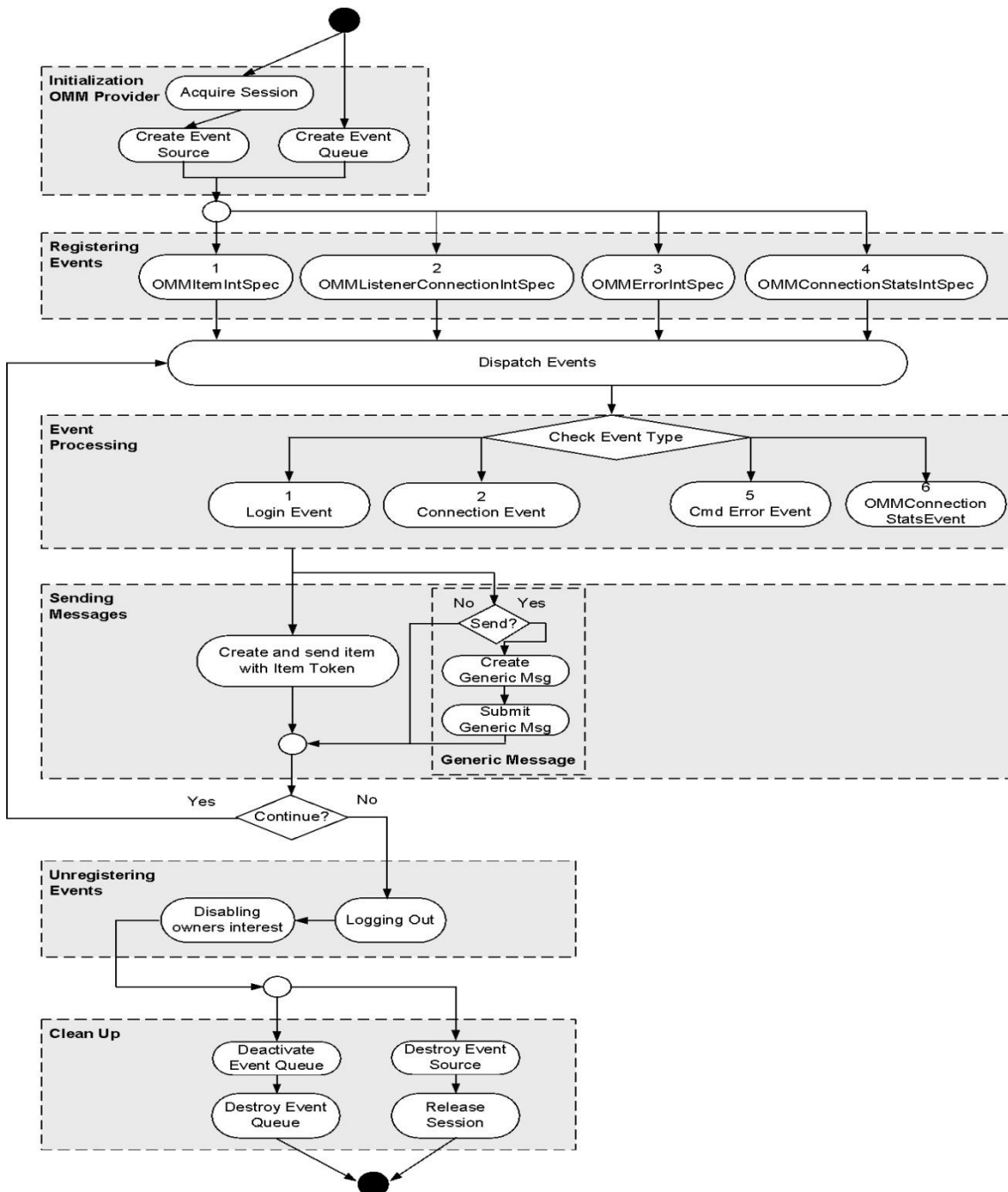


Figure 96: High level activity diagram for implementing Non-Interactive Provider

12.3.1 Initialization OMM Provider

This is the same process as Interactive Provider in section

12.3.2 Registering Events for Non-Interactive Provider

The following snippet activity diagram depicts the relationship between calling `RegisterClient()` for Interest Specifications of Non-Interactive Provider and the event which will receive from these registrations.

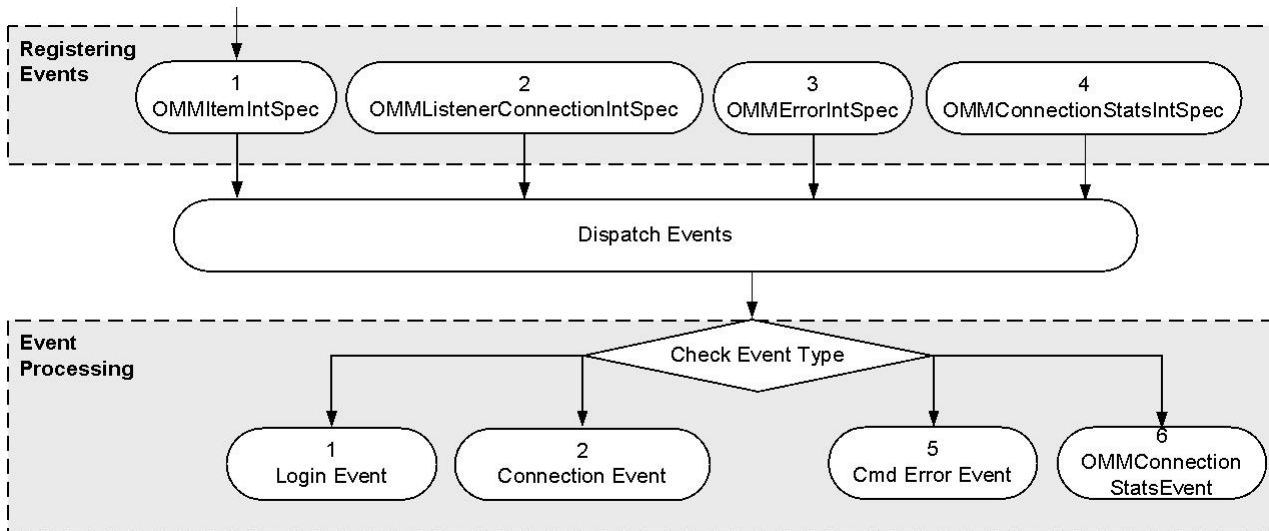


Figure 97: Registering Events for Non-Interactive Provider

- OMMItemIntSpec:** This interest specification is required by a provider application. After the OMM Non-Interactive Provider has been created, it will need to send a login request and receive login permission before it can start to publish data. In order to receive login Response Messages, the application needs to register with the OMM Non-Interactive Provider to send login request by calling `RegisterClient()` using the `OMMItemIntSpec`. Once registered, the application will be able to receive all login related events. The `RegisterClient()` method will build up a connection to the back-end infrastructure, and send login request to the infrastructure, returning a login handle to the application. Note that RFA will return a Cmd error message if the provider application submits data before receiving a login success message. See Section 11.3.1, Login for how to create a login Request Message. Once the provider application has registered for this Interest Specification, the application will then be able to receive `OMMItemEvent` coming from the back-end infrastructure.
- Instanceld** is one of Login request's attributes which is used to differentiate applications running on the same client host. If there are more than one Non-Interactive Provider instance running on the same host, they need to be set as different value by provider application. Otherwise infrastructure component which providers connect to will reject the login request of same Instanceld value, and then cut the connection.
- OMMConnectionIntSpec:** This interest specification is optional for a provider application. This is used for accepting connection status events. Once the provider application has registered for this Interest Specification, the application will then be able to receive `OMMConnectionEvent` coming from RFA adapter.
- OMMErrorIntSpec:** This interest specification is optional for a provider application. This is used to register so that the provider application can receive error events (`OMMCmdErrorEvent`) related to the call to `Submit()` of the OMM Non-Interactive Provider.
- OMMConnectionStatsIntSpec:** This interest specification is optional for a provider application. This is used to receive connection statistics events for a connection or a list of connections. Once the provider application registers for this Interest Specification, the application is able to receive `OMMConnectionStatsEvent`.
- When calling the `RegisterClient()` method, the method accepts up to four parameters. The first parameter is a reference to the Event Queue to which the connection Event will be sent. The second parameter is the Interest Specification. The third parameter is the Client, which will receive the Event callback.

- Finally, the fourth parameter is the Closure as described in section 5.1.1.6, Closures. The return value of `RegisterClient()` is a Handle to the newly opened Event Stream.
- If an application allocates any of the Interest Specification on the heap, it should release the Interest Specification when it is no longer needed. The application can release the Interest Specification at any time after the return from the `RegisterClient()`. RFA retains a copy in the `RegisterClient()` call.

12.3.2.1 Initial Steps for Registering for Events

Typically after the creating of the OMM Non-Interactive Provider, the providing application should register for `OMMItemIntSpec`, `OMMConnectionIntSpec`, and `OMMErrorIntSpec`. This sets up the provider application so that it can receive all event types from the OMM Non-Interactive Provider.

```
// Register for Connection events (these are connection events)
OMMConnectionIntSpec connectionIntSpec = new OMMConnectionIntSpec();
ommConnIntSpecHandle = ommProvider.RegisterClient(eventQueue, connectionIntSpec, this);

// Register for Cmd Error events (These events are sent back if the Submit() call fails)
OMMErrorIntSpec ommErrorIntSpec = new OMMErrorIntSpec();
ommErrIntSpecHandle = ommProvider.RegisterClient(eventQueue, ommErrorIntSpec, this, null);

// Register to receive login response events from the back-end infrastructure
OMMItemIntSpec ommItemIntSpec = new OMMItemIntSpec();
ReqMsg reqMsg = new ReqMsg();
AttribInfo attribInfo = new AttribInfo();

reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

attribInfo.NameType = RDM.Login.USER_ID_TYPES.USER_NAME;
attribInfo.Name = cfgVariables.UserName;

ElementList elementList = new ElementList();
ElementEntry element = new ElementEntry();
DataBuffer elementData = new DataBuffer();
ElementListWriteIterator elwiter = new ElementListWriteIterator();
elwiter.Start(elementList);

element.Name = RDM.Login.ENAME_APP_ID;
elementData.SetFromString(cfgVariables.AppId, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);

element.Name = RDM.Login.ENAME_POSITION;
elementData.SetFromString(cfgVariables.Position, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);

if (cfgVariables.InstanceId.Length != 0)
{
    element.Name = RDM.Login.ENAME_INST_ID;
    elementData.SetFromString(cfgVariables.InstanceId, DataBuffer.DataBufferEnum.StringAscii);
    element.Data = elementData;
    elwiter.Bind(element);
}

elwiter.Complete();
attribInfo.Attrib = elementList;
reqMsg.AttribInfo = attribInfo;
```

```
ommItemIntSpec.Msg = reqMsg;
loginHandle = ommProvider.RegisterClient(eventQueue, ommItemIntSpec, this, null);
```

Example 203: Register for OMMConnectionIntSpec, OMMErrorIntSpec and OMMItemIntSpec

12.3.3 Dispatching Event Queue

This is the same process as Interactive Provider in section

12.3.4 Event Processing for Non-Interactive Provider

In addition to registering for Events, the application must define the code to process these Events when RFA invokes the Client call-out method (i.e., `ProcessEvent()`). The application defines the code to process these Events by deriving from the `Client` interface and implementing the `ProcessEvent()` method.

The following snippet activity diagram depicts the relationship between registering and processing events for non-interactive provider.

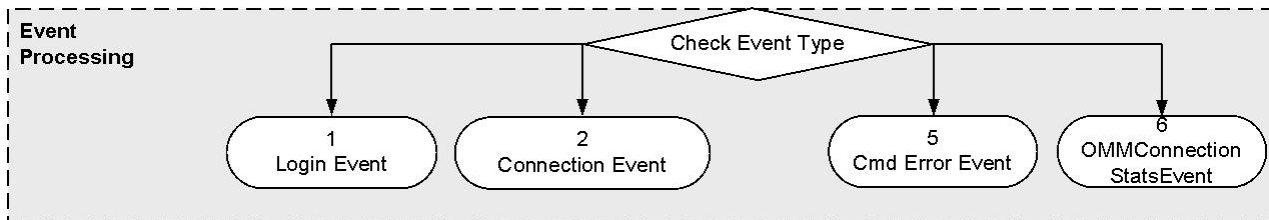


Figure 98: Processing Events for Non-Interactive Provider

The responsibility of Non-Interactive Provider is to handle events which are registered to receive and submit `cmds` to consumer clients whenever it wants to. Moreover, the item token object is used to represent a unique identifier for every publishing item.

12.3.4.1 Handling ProcessEvent() in the application

The best practise for processing events is to separate each event handling into a function so the application can handle them separately by switching on the Event type into the application's implementation of `ProcessEvent()`. If the application uses the Client to process other types of Events, it would define additional case statements.

The implementation example of `ProcessEvent()` is shown as follows:

```
public void ProcessEvent(Event evt)
{
    switch (evt.Type)
    {
        case SessionLayerEventTypeEnum.ConnectionEvent:
            ProcessConnectionEvent(evt as OMMConnectionEvent);
            break;
        case SessionLayerEventTypeEnum.OMMItemEvent:
            ProcessOMMItemEvent(evt as OMMItemEvent);
            break;
        case SessionLayerEventTypeEnum.OMMCmdErrorEvent:
            ProcessOMMCmdErrorEvent(evt as OMMCmdErrorEvent);
            break;
        case LoggerEventTypeEnum.LoggerNotifyEvent:
            ProcessLoggerNotifyEvent(evt as LoggerNotifyEvent);
            break;
        case SessionLayerEventTypeEnum.OMMConnectionStatsEvent:
            Process OMMConnectionStatsEvent (evt as OMMConnectionStatsEvent);
            break;
        default:
    }
}
```

```

        AppUtil.Log(AppUtil.LEVEL.WARN, string.Format("<- Received unknown Event type: {0} handle: {1}",
            evnt.Type, evnt.Handle));
        break;
    }
    if (evnt.IsEventStreamClosed)
    {
        AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("<- Received Stream Closed Event type: {0} handle:
            {1}", OMMStrings.EventTypeToString(evnt.Type).ToString(), evnt.Handle));
    }
}

```

Example 204: Handling ProcessEvent()

All OMM Events 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 Event (or its contained message) using the `Clone()` method.

12.3.4.2 Handling Item Events (Login Events)

The `OMMItemEvent` in the OMM Non-Interactive Provider represents a Response Message of `MMT_LOGIN` model type. The response message will tell the provider application the state of the user login. Once this event is received, the call `RespStatus` property on the `RespMsg` will return the status of the login. The state of login response will determine what to do next.

The `RespStatus` property returns `RespStatus` which will tell you the stream state and data state of the user login through `StreamState` and `DataState` properties. If the stream state from the first login response message is Open, it means that the user got login permission from the back-end infrastructure and the provider application can start to publish data. Otherwise the provider application cannot start to publish data.

12.3.4.3 Login Events: Login Success

Once the login succeeds, the provider application can start to publish data including source directory, dictionary and other Response Messages of different message model types. The provider application will need to create a [RespMsg](#) populated according to the type. Specific handling of all message model types is defined in the *RFA RDM Usage Guide .NET Edition*. This document covers all parameters and encoding details for properly responding to each of the message model types.

To encode the payload for the [RespMsg](#), see section 7.2.1.2, Encoding Response Message for details on appropriate creation of the [RespMsg](#).

Once the [RespMsg](#) is properly populated, it then needs to call [submit\(\)](#) on the OMM Non-Interactive Provider to send data out. See section 12.2.5.1, Response Message.

Note that it is a good idea for the provider application to initialize items to be published after login succeeds. The initialization includes getting an item token for each item through the [GenerateItemToken\(\)](#) method on OMMProvider interface.

12.3.4.4 Login Events: Other Login States

If the stream state of the first login response message is Closed, it means the login failed, and the provider application failed to get permission from the back-end infrastructure. In this case, the provider application cannot start to publish data.

The provider application can also receive other login states during publishing. If the stream state is OK, but the data state is [DataStateEnum.Suspect](#), it means all connections are down. The application can stop publishing and resume again once the data state becomes [DataStateEnum.Ok](#).

```
private void ProcessOMMItemEvent(OMMItemEvent evnt)
{
    Msg msg = evnt.Msg;
    switch (msg.MsgType)
    {
        case MsgTypeEnum.GenericMsg:
            ProcessGenericMsg(evnt);
            break;
        case MsgTypeEnum.RespMsg:
            ProcessRespMsg(evnt, (RespMsg)(msg));
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, String.Format("<- Received event with unknown message type: {0}",
                msg.MsgType));
            break;
    }
}

void ProcessRespMsg(OMMItemEvent evnt, RespMsg respMsg)
{
    switch (respMsg.MsgModelType)
    {
        case RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN:
            ProcessLoginResponse(evnt, respMsg);
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.WARN, String.Format("<- Received unhandled OMMItemEvent msgModelType:
                {0}", respMsg.MsgModelType));
            break;
    }
}

void ProcessLoginResponse(OMMItemEvent evnt, RespMsg respMsg)
{
}
```

```

RespStatus status = respMsg.RespStatus;
RFA_String text = OMMStrings.RespStatusToString(status);

//For a Login Response examine the stream state and data state:
//If stream state is open and data state is OK then we have a successful Login
//If stream state is open and data state is NOT OK then we have a pending Login switch (respMsg.RespType)
{
    case RespMsg.RespTypeEnum.Refresh:
        if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
        {
            if (status.StreamState == RespStatus.StreamStateEnum.Open)
            {
                if (status.DataState == RespStatus.DataStateEnum.Ok)
                {
                    AppUtil.Log(AppUtil.LEVEL.INFO, String.Format("<- Received MMT_LOGIN Refresh - Login Accepted {0}", text.ToString()));
                    ProcessLoginSuccess(evt);
                }
                else if (status.DataState == RespStatus.DataStateEnum.Suspect)
                {
                    AppUtil.Log(AppUtil.LEVEL.WARN, String.Format("<- Received MMT_LOGIN Refresh - Login Pending {0}", text.ToString()));
                    ResetItems();
                    isPublishData = false;
                }
            }
            else if (status.StreamState == RespStatus.StreamStateEnum.Closed)
            {
                AppUtil.Log(AppUtil.LEVEL.ERR, String.Format("<- Received MMT_LOGIN Refresh - Login Denied {0}", text.ToString()));
                isPublishData = false;
            }
        }
        break;
    case RespMsg.RespTypeEnum.Status:
        if ((respMsg.HintMask & RespMsg.HintMaskFlag.RespStatus) != 0)
        {
            if (status.StreamState == RespStatus.StreamStateEnum.Open)
            {
                if (status.DataState == RespStatus.DataStateEnum.Ok)
                {
                    AppUtil.Log(AppUtil.LEVEL.INFO, String.Format("<- Received MMT_LOGIN Status - Login Accepted {0}", text.ToString()));
                    ProcessLoginSuccess(evt);
                }
                else if (status.DataState == RespStatus.DataStateEnum.Suspect)
                {
                    AppUtil.Log(AppUtil.LEVEL.WARN, String.Format("<- Received MMT_LOGIN Status - Login Pending {0}", text.ToString()));
                    ResetItems();
                    isPublishData = false;
                }
            }
            else if (status.StreamState == RespStatus.StreamStateEnum.Closed)
            {
                AppUtil.Log(AppUtil.LEVEL.ERR, String.Format("<- Received MMT_LOGIN Status - Login Denied {0}", text.ToString()));
                isPublishData = false;
            }
        }
    }
}

```

```

        else
            AppUtil.Log(AppUtil.LEVEL.ERR, "<- Received MMT_LOGIN Status - No RespStatus");
            break;
        case RespMsg.RespTypeEnum.Update:
            //In the future we could receive an update with a new permission profile
            AppUtil.Log(AppUtil.LEVEL.ERR, "<- Received MMT_LOGIN Update");
            break;
        default:
            AppUtil.Log(AppUtil.LEVEL.ERR, String.Format("<- Received a non-supported MMT_LOGIN RespMsg
                type: {0}", respMsg.RespType));
            break;
    }
}

```

Example 205: Handling Login Events

12.3.4.5 Handling CmdError Events

The `OMMCmdErrorEvent` represents an error Event that is generated during the `Submit()` call on the OMM Non-Interactive Provider.

This Event gives the provider application access to the `Cmd`, `CmdID`, `closure` and `OMMErrorStatus` for the `Cmd` that failed. These can be accessed via `Cmd`, `CmdID`, `SubmitClosure` and `Status` respectively.

No Response Message is required from the provider application for this Event.

```

private void ProcessOMMCmdErrorEvent(OMMCmdErrorEvent evnt)
{
    AppUtil.Log(AppUtil.LEVEL.ERR,
        string.Format("<- Received OMMCmdErrorEvent:\r\n    Cmd ID: {0}\r\n    State: {1}\r\n    StatusCode:
            {2}\r\n    StatusText: {3}\r\n",
            evnt.CmdID, evnt.Status.State, evnt.Status.StatusCode, evnt.Status.StatusText.ToString()));
}

```

Example 206: Handling CmdError Events

12.3.4.6 Handling Connection Events

For the OMM Non-Interactive Provider, the `OMMConnectionEvent` provides information with respect to the connection. This will contain connection name and type information along with the `ConnectionStatus` of the event. This information will tell if the connection is up or down.

The `OMMConnectionEvent` interface also allows the application to receive more information related to the connection, such as active host name, port, and version of the connected component.

No Response Message is required from the provider application for this Event.

```
private void ProcessConnectionEvent(OMMConnectionEvent ommConnectionEvent)
{
    ConnectionStatus connectionStatus = ommConnectionEvent.Status;
    RFA_String text = new RFA_String();
    RFA_String port = new RFA_String();
    if (GetPort(port, ommConnectionEvent.ConnectionName))
    {
        text.Append("\r\n    Listening on Port: ");
        text.Append(port);
    }

    if (connectionStatus.State == ConnectionStatus.StateEnum.Up)
    {
        text.Append("<- Received ConnectionEvent -----Connection Up!----- ");
    }
    else if (connectionStatus.State == ConnectionStatus.StateEnum.Down)
    {
        text.Append("<- Received OMMConnectionEvent -----Connection Down!----- ");
    }
    AppUtil.Log(AppUtil.LEVEL.INFO, string.Format("{0}\r\n", text.ToString()));
}
```

Example 207: Handling Connection Events

12.3.4.7 Handling LoggerNotify Events

As with all of the Event Sources, logger Events can be received as well. See section 10.2, Logger Package Usage for more on handling logger Events.

No Response Message is required from the provider application for this Event.

12.3.4.8 Handling Connection Statistics Events

The `OMMConnectionStatsEvent` provides information on statistics of a connection. The statistics of a connection are bytes read and written on the wire provided by RFA, periodically over a time period specified by the application while registering to this event.

```
/ Register for connection statistics events
public void ProcessEvent( Reuters.RFA.Common.Event evnt )
{
    switch ( evnt.Type )
    {
        case Reuters.RFA.SessionLayer.SessionLayerEventTypeEnum.OMMConnectionStatsEvent:
            {
                OMMConnectionStatsEvent connectionStatsEvent = (OMMConnectionStatsEvent)evnt;
                System.Console.WriteLine("Received Connection statistics Event from {0} on Handle {1} with bytes Read = {2} and bytes written = {3}",
                    connectionStatsEvent.ConnectionName, connectionStatsEvent.Handle, connectionStatsEvent.BytesRead,
                    connectionStatsEvent.BytesWritten);
            }
        break;
        // Process other event types...
    }
}
```

12.3.5 Sending Messages for Non-Interactive Provider

Non-Interactive provider needs to login into a consumer application before sending a message. After that it can send messages to the consumer application without any request. The following snippet activity diagram depicts the relation between processing the login event before sending a message for non-interactive provider.

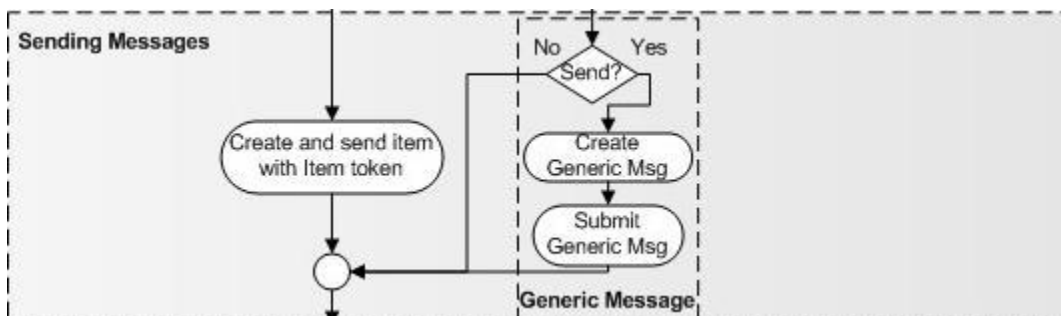


Figure 99: Sending messages for Non-Interactive Provider

12.3.5.1 Response Message

Sending Response Message to the connection involves processing as follows:

1. Create a Response Message (`Reuters.RFA.Message.RespMsg`). The response message is subclassed from `Reuters.RFA.Common.Msg`.
2. Set the message model type of the response using the `MsgModelType` property. Possible values are defined in `Reuters.RFA.RDM.RDM.MESSAGE_MODEL_TYPES`. See the *RFA RDM Usage Guide .NET Edition* for details.
3. Set response type by using the `RespType` property. Possible values are defined in `Reuters.RFA.Message.RespMsg.RespTypeEnum`.
4. If the response type is `RespTypeEnum.Refresh`, the provider application needs to set response type enumeration to `REFRESH_UNSOLICITED` by using `RespTypeEnum` property. Note that all refresh messages to be published should be unsolicited refresh messages.

5. Create or re-use a request attribute object (`Reuter.RFA.Message.AttribInfo`). Refer to the *RFA RDM Usage Guide .NET Edition* for details.
6. Populate the response message with the `AttribInfo` object using the `AttribInfo` property. This is optional if an item token is used, and the message is an update or a status message.
7. For details on how to create `RespMsg`, refer to Section 7.2.1.2.
8. Create the `OMMItemCmd` object and populate it with the response message using the `Msg` property. The `Cmd` essentially acts as a wrapper around the response message.
9. Call the `ItemToken` property to set item identifier if the provider application decides to use an item token to identify an item. Note that if an item token is used, all messages including refresh, updates and other types of messages need be submitted with item tokens.
10. Call the `Submit()` method of the OMM Non-Interactive Provider Event Source to write the response message directly out to the network through the connection.
11. The `Cmd(OMMItemCmd)`, Response Message and response attribute objects may be reused for making subsequent responses.

```
private bool SubmitMsg()
{
    bool result = false;
    OMMSolicitedItemCmd itemCmd = new OMMSolicitedItemCmd();
    List<long> clientSessions = new List<long>();
    providerWatchList.GetClientSessions(clientSessions);

    for (int pos = 0; pos < clientSessions.Count; pos++)
    {
        long clientSessionHandle = clientSessions[pos];
        ClientWatchList cw1 = providerWatchList.GetClientWatchList(clientSessionHandle);

        // skip on error condition or if there are no tokens
        if ((cw1 == null) || (cw1.Size() == 0))
        {
            continue;
        }

        ClientWatchList.TokenInfo TS = null;
        RFA_String reason = new RFA_String();
        for (int i = 0; i < cw1.Size(); i++)
        {
            TS = cw1.GetTokenInfo(i);

            // need to skip login request token!
            if (TS.isItemRequest)
            {
                reason.Set("Solicited MMT_MARKET_PRICE ");
                RespStatus status = new RespStatus();
                if (!TS.isSubmitted)
                {
                    // send refresh if it was not submitted yet
                    status.StreamState = RespStatus.StreamStateEnum.Open;
                    status.DataState = RespStatus.DataStateEnum.Ok;
                    status.StatusCode = RespStatus.StatusCodeEnum.None;
                    status.StatusText = new RFA_String("Solicited Refresh Completed");
                    SetMarketPriceMsg(RespMsg.RespTypeEnum.Refresh, TS.attribInfo, status,
                        clientSessionHandle, true);

                    itemCmd.Msg = respMsg;
                }
            }
        }
    }
}
```

```

        itemCmd.RequestToken = TS.requestToken;
        reason.Append("Refresh");

        result = SubmitCmd(itemCmd, null, reason);
        TS.isSubmitted = true;

        //Update information in structure after set the submit flag
        cw1.UpdateToken(cw1.GetRequestToken(i), TS);
    }
    else
    {
        // send update if it was not paused yet
        SetMarketPriceMsg(RespMsg.RespTypeEnum.Update, TS.attribInfo, status, clientSessionHandle,
            TS.isAttribInfoInUpdates);
        itemCmd.Msg = respMsg;
        itemCmd.RequestToken = TS.requestToken;
        reason.Append("Update");

        result = SubmitCmd(itemCmd, null, reason);
    }
}
}
}
return result;
}

```

Example 208: Submit Response Message

12.3.5.2 Generic Message

A non-interactive provider can send Generic Message(s) by using the currently available `OMMItemCmd` and `Submit(..)` methods. To send a `GenericMsg` the non-interactive provider would encode the message, create an `OMMItemCmd`, and pass corresponding stream's `ItemToken` on the `OMMItemCmd`, and pass the `OMMItemCmd` to the `Submit(..)` function.

```
private void SubmitGenericMsg(long itemHandle, ItemToken itemToken, RFA_String value)
{
    ElementList payLoad = new ElementList();
    payLoad.SetAssociatedMetaInfo(itemHandle);
    ElementListWriteIterator iter = new ElementListWriteIterator();
    ElementEntry entry = new ElementEntry();
    DataBuffer buffer = new DataBuffer();
    RFA_String ename = new RFA_String();
    RFA_String val = new RFA_String();

    iter.Start(payLoad);

    ename.Set("Data");
    val.Set(value.ToString());
    buffer.SetFromString(val, DataBuffer.DataBufferEnum.StringAscii);
    entry.Name = ename;
    entry.Data = buffer;
    iter.Bind(entry);
    iter.Complete();

    GenericMsg genericMsg = new GenericMsg();
    genericMsg.MsgModelType = 200;

    AttribInfo attribInfo = new AttribInfo();
    attribInfo.NameType = RDM.RDM.INSTRUMENT_NAME_TYPES.INSTRUMENT_NAME_RIC;

    attribInfo.Name = new RFA_String("SPEAK");

    genericMsg.AttribInfo = attribInfo;
    genericMsg.Payload = payLoad;
    genericMsg.IndicationMask = GenericMsg.IndicationMaskFlag.MessageComplete;

    OMMItemCmd itemCmd = new OMMItemCmd();
    itemCmd.ItemToken = itemToken;
    itemCmd.Msg = genericMsg;

    ommProvider.Submit(itemCmd);
}
```

Example 209: Submit Generic Message

12.3.6 Unregisters Events for Non-Interactive Provider

The following sequence should be done sequentially to unregister any submitting of responses. The following snippet activity diagram depicts the unregister events for non-interactive provider.

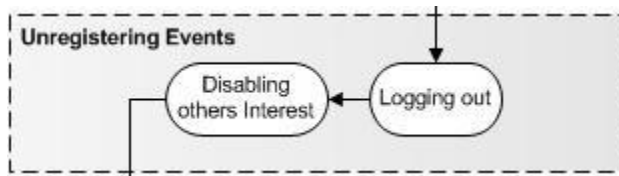


Figure 100: Unregistering processes for Non-Interactive Provider

12.3.6.1 Logging out

At any time after the user login has been accepted, the provider application has the ability to log out the user through the `UnregisterClient()` method by passing in the login handle which is received through `RegisterClient()` method. The process of unregistering will disconnect all existing connections associated with the login handle, and clean up all item tokens and their memory allocated by RFA internally.

All item tokens associated with this login handle MUST NO longer be referenced by the Non-Interactive provider application. Calling `unregisterClient(null)` is NOT supported.

```
// Unregister login handle
ommProvider.UnregisterClient(loginHandle);
```

Example 210: Logging out

12.3.6.2 Disabling others Interest Spec

Unregistering for connection, logger and `CmdError` Events is done similarly to the above except that the corresponding handle should be passed in.

12.3.7 Cleaning up

This is the same process as Interactive Provider in section 3.2.2.1.

Chapter 13 RFA Feature in Details

13.1 Batch

In addition to registering for interest in a single item, RFA consumer applications can specify interest in multiple items via a list of items. In response to the batch request, an RFA consumer will receive multiple, fully-functional, independent item streams, one for each item specified in the itemname list of the batch Request Message.

Batch functionality also allows clients to close multiple items in a single close request message. The batch reissue feature also allows clients 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.

Applications should check the [SupportBatchRequests](#) element of the login response to determine whether a provider supports batch requests, batch reissues, and batch closes. If a provider does not support batching, RFA will break apart the batch request on the application's behalf and send individual item requests to the provider instead.

In the example below, the client makes a single batch request, reissue and close.

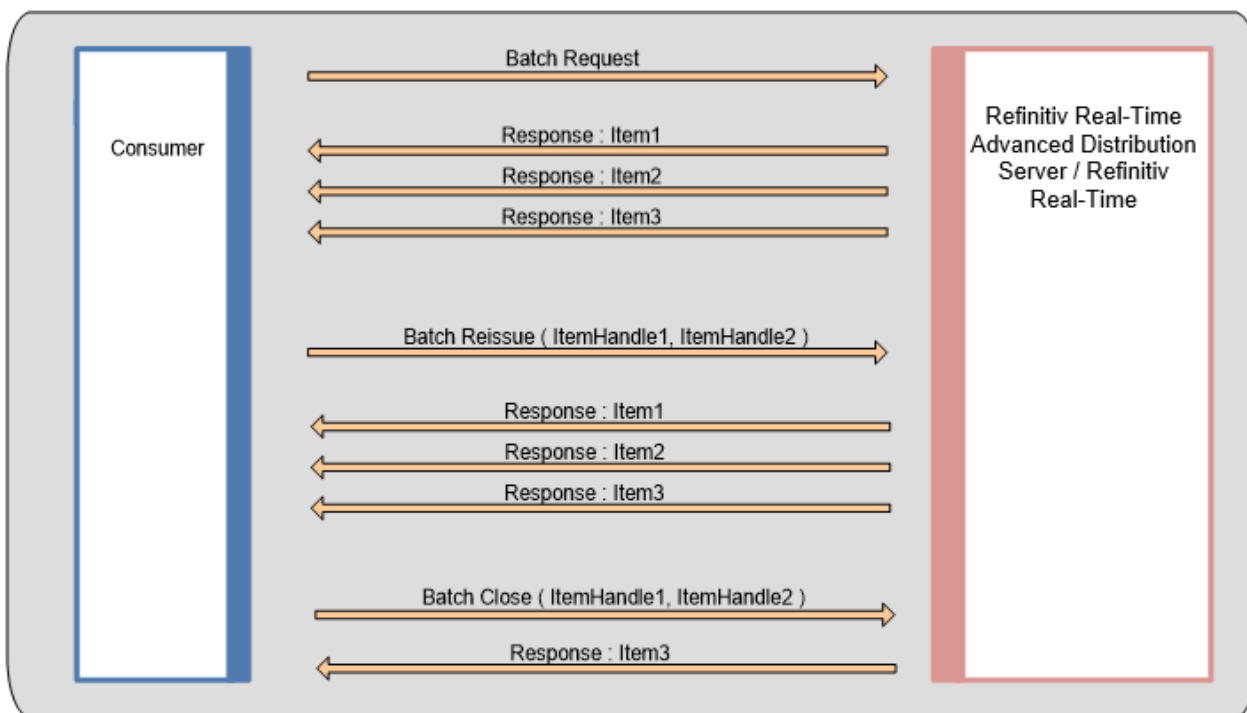


Figure 101: Batch Request, Reissue and Close

An OMM Consumer application can specify interest in a list of items by using a single batch request message. To send a Batch Request, the application sets the `ReqMsg.IndicationMaskFlag.Batch` and encodes an `ItemList` element entry in an element list contained in the payload of the request message. The `ReqMsg.IndicationMaskFlag.Batch` indicates that the request message contains a batch of items and that the payload includes an `ItemList`. See section 7.2.1.1.3, Encoding Request Message for Batching Items for usage examples.

An application should expect to get both a handle for the initial batch request and a handle for each item that was requested. The handle that is associated with the batch request is different from a normal item handle. It cannot be used to reissue the batch request because by the time the application has received the response the stream associated with the batch handle was considered closed.

The first response for each item specified in a batch request always includes the item name and the item handle. Each item stream is also completely independent of the original batch request. Clients can use the item handle to identify subsequent Response Message for the item or perform a reissue on the item stream. All RFA recovery mechanisms can be performed on the stream as usual. For usage examples, refer to Section 1.2.2.

All interest specifications specified in a batch Request Message must have the same request attributes. A batch request can be used for streaming or non-streaming item requests. Batch is available for all non-administrative RDM domains and any user-defined domain message models as long as all the items specified in the batch request have the same Message Model Type.

An OMM Consumer application can change the request attributes on a list of items by using a single batch request message. To send a Batch Reissue, the application must call `ReissueClient()` by passing a list of item handles. If a list passed in is empty, the attribute changes will apply to all open streams of non-administrative domain types.

```
List<long> itemHandleToBeReissued = new List<long>();

itemHandleToBeReissued.Add( itemHandle1 );
itemHandleToBeReissued.Add( itemHandle2 );

ommConsumer.ReissueClient( itemHandleToBeReissued, ommItemIntSpec );
```

Example 211: Sending a Batch Reissue

An OMM Consumer application can also close a list of items by using a single batch request message. To send a Batch Close, the application must call `UnregisterClient()` by passing a list of item handles. If a list passed in is empty, RFA will close all open streams except the login stream.

```
List<long> itemHandleToBeClosed = new List<long>();

itemHandleToBeClosed.Add( itemHandle1 );
itemHandleToBeClosed.Add( itemHandle2 );

ommConsumer.UnregisterClient( itemHandleToBeClosed );
```

Example 212: Sending a Batch Close

The Batching capability works together with RFA's request throttling. If throttling is enabled, RFA may split a batch request into multiple smaller requests. If throttling is disabled, a batch request will be sent upstream as it was from the client application. For example, If the `throttleType` is count, and the `throttleBatchCount` is set to 5, RFA will split a batch request of 9 items into one request of 5 items and one request of 4 items when sent to the provider.

Proper throttle configuration is required to make sure that configuration value is not set too high, which will flood the network and overload the connection, or set too low, which will disrupt the outbound speed of the batch request. More detail about Consumer Request Throttling refer to Section 14.9.

More information about Batch messages is listed in the table below.

DESCRIPTION	SECTION
Batch concept	3.2.1.3 Batch
Batching Items in Request Message	7.1.9 Batching Items in a Request Message
Batch request example from a consumer	7.2.1.1.3 Encoding Request Message for Batching Items
Batch response example from a provider	7.2.2.4.1 Decoding Batch Item Response Message

Table 76: Further Reading: Batch

13.2 Dynamic View

Consumer applications can reduce bandwidth by requesting a specific subset of [FieldEntry](#)s and [ElementEntry](#)s for a particular item. This reduces both the bandwidth needed on the wire and also time needed to decode the responses. Views can be used for streaming requests, non-streaming requests, and batch requests.³⁴ Views apply to any RDM or user-defined model that has pairs of [FID/FieldValue](#).

The Consumer defines a View by setting the [ReqMsg.IndicationMaskFlag.View](#) of the [ReqMsg](#)'s [IndicationMask](#) and encoding [ViewType](#) and [ViewData](#) element entries in an element list contained in the payload of the [ReqMsg](#). The [ReqMsg.IndicationMaskFlag.View](#) indicates that a View request exists within the request message, and that a View definition is in the payload.

Applications should check the [SupportViewRequests](#) element of the login response to determine whether a provider supports views. If a provider does not support views, RFA will internally remove the [ReqMsg.IndicationMaskFlag.View](#) on the application's behalf and send a request to the provider asking for a full image instead.

Provider applications can choose to supply only the requested subset of content across all Response Messages. Consumers should be prepared to receive more or fewer fields than just the subset of fields that they requested. It is the consumer's responsibility to maintain a view definition that it can use to filter subsequent Response Messages.

Views can also be reissued. A consumer application can dynamically change a view on an open item stream by reissuing a request with a new view. Although fields or elements in a view can be modified through a reissue, the [ViewType](#) cannot be changed after the stream is opened.

More information about Dynamic Views is listed in the table below.

DESCRIPTION	SECTION
Views concept	3.2.1.4 View
Dynamic views and views in RDM	7.1.10 Dynamic View
Dynamic view request example from a consumer	7.2.1.1.2 Encoding Request Message for View
Dynamic view response example from a provider	7.2.1.1.2 Encoding Request Message for View

Table 77: Further Reading: Dynamic Views

³⁴ In a batch request, a single view is specified is applied to all items contained in the batch.

13.3 Generic Message

Consumer and Provider applications can send Generic Message that can contain any payload data as needed. One advantage of using Generic Message is its freedom from the traditional Request/Response data flow. Generic Message can be sent from both consumers to providers and from providers to consumers, and are available on administrative RDMS³⁵ and user-defined domain models.

To send [GenericMsgs](#), a Consumer and a Provider should establish a stream through the request/response process. Once a stream is opened, bi-directional [GenericMsgs](#) from a Consumer to a Provider (or vice versa) can be sent over the stream.

Applications send a Generic Message by first creating a [GenericMsg](#) object and set the message's model type to a domain model. The application then populates the [AttribInfo](#) and encodes and sets the payload of the message. Finally, the application populates a [OMMHandleItemCmd](#) with the message and calls [Submit\(\)](#) with the [OMMHandleItemCmd](#) using the item handle returned from [RegisterClient\(\)](#) when the initial request was made.

More information about Generic Messages is listed in the table below.

DESCRIPTION	SECTION
Generic Message concept	3.2.1.9 Generic Message
Generic Message in the Consumer	11.7.1 Generic Message 11.6.5 Processing Generic Message
Generic Message in the Provider	12.3.5.2 Generic Message

Table 78: Further Reading: Generic Message

³⁵ Generic Message used by RFA on the login domain is limited to sending the ConsumerConnectionStatus information request for Warm Standby. Use by RFA on the directory domain is limited to the ConsumerStatus and SourceMirroringMode messages. Use by RFA on the dictionary domain is not supported.

13.4 Posting

OMM applications can use the Posting feature to can easily push content into any cache within the RTDS.³⁶ 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.

A Post Message contains information that an application wants to publish. Post Message is an OMM message type that can contain any OMM data container type (e.g., [Map](#), [ElementList](#)), opaque data, or even another OMM message as their payload. Posting enables end-to-end posting with no restrictions on data, message size, message type, or domain. Post Message can also be sent with an acknowledgement requested from the Posting provider. Posting also supports multi-part messages for large messages. Applications can use a post ID³⁷ and a sequence number to distinguish between messages.

When content is being posted, the permission data included in the [PostMsg](#) is used to permission the user that is posting the information. Posting also supports setting and receiving the user's DACS attributes, known as Post User Rights. Applications can use the [PostUserRightsMask](#) on the [PostMsg](#) for setting and receiving the user rights with that post.

When posting, consumer applications use their existing sessions to publish content to any cache residing within the RTDS. Applications can send a Post using the item handle, known as an on-stream post, or the login handle, known as an off-stream post.

Consumer and Provider applications should use the **SupportOMMPost** login attribute to indicate whether they can use and support the OMM Post feature.

To publish a [PostMsg](#), the application creates a Post Message and populates it with the necessary message and payload information. The consumer application encapsulates the Post Message in an [OMMHandleItemCmd](#) object by setting [Msg](#) property and also sets the item handle³⁸ to [Handle](#) property. The Post Message is then sent to the network using the [Submit\(\)](#) method.

Calling [Submit\(\)](#) can fail in some cases. For example, an application might attempt to send a multi-part Post Message without specifying a post ID or sequence number. The application will receive an [OMMCmdErrorEvent](#) in these cases. It should register to listen for [OMMErrorIntSpec](#) to receive these messages.

Processing a [PostMsg](#) is similar to other messages. After determining that the message type is [MsgTypeEnum.PostMsg](#), the application can retrieve values like the post ID, sequence number, attributes, and decode the payload.

More information about Posting is listed in the table below.

DESCRIPTION	SECTION
Posting concept in consumer	3.2.1.8 Posting
Posting concept in provider	3.2.2.1.2 Posts
Steps to send Post Message and process Ack Message in a Consumer	11.7.2 Post Message 11.6.6 Processing Ack Message
Steps to process Post Message and send Ack Message in a Provider	12.2.4.5 Handling Consumer SolicitedItemEvents: Post Message 12.2.5.3 Ack Message

Table 79: Further Reading: Posting

³⁶ Posting replaces the way MarketData used contributions/inserts for publishing into a cache.

³⁷ Post IDs are also used to match Posts and their corresponding Acks.

³⁸ The application obtained this handle when it made the initial request for the item or login using [RegisterClient\(\)](#).

13.5 Private Stream

In contrast to 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. Private streams may be established for streaming Level 1 and Level 2 data.

To make a request for an item on a Private Stream, a Consumer sets the `IndicationMask` of the `ReqMsg` to `Reuters.RFA.PrivateStream.IndicationMaskFlag.PrivateStream`. The Provider should recognize the `PrivateStream` and honor the request or send a status message with appropriate text and close the stream. Once established, RFA is aware that the stream state is private and guards against state changes between Consumer and Provider.

13.5.1 Private Streams Rules and Constraints

- If an OMM Provider receives a request with the `PrivateStream` set, it may accept it by providing a Refresh or Status message with the `PrivateStream` set and a stream state of `Open`, or it may reject it by providing a refresh or status message with the `PrivateStream` set and a stream state of `Closed` with the appropriate status text.
- An OMM Consumer is not required to set the `PrivateStream` on messages other than the initial request message.
- The `PrivateStream` must be set on Refresh or Status messages by the provider application before being sent to the consumer.
- To ensure no data sharing on the network and application, RFA enforces the `PrivateStream` settings between request and response messages.
 - If an OMM Consumer receives a response to a private stream request and no `PrivateStream` is set, it implies that the stream is not private. RFA will intercept this message and close this stream to the provider.
 - If an OMM Consumer receives a response to a standard stream request and `PrivateStream` is set, it implies that the stream is not standard. RFA will intercept this message and close this stream to the provider.
 - If a `PrivateStream` mismatch between response and request messages is detected, RFA will overwrite incoming message status code and text to indicate this condition.
- An OMM Consumer should request private streams using concrete service names even if service groups are used. This ensures that the OMM Consumer connects with a private stream to the OMM Provider of its choice.
- Private streams will not be recovered regardless of the cause of recovery and of the `SingleOpen` value used in the login request.
- Depending on the infrastructure configuration, private streams may be cached. The cached values, however, will be used for item reissues only. When the connection is closed (or if it is lost), the cache will be removed.
- When using private streams with Warm Standby, RFA recovers only regular streams from the standby server and sends an internally-generated status message with the stream state `ClosedRecover` for all private stream items during recovery. Items requested as private streams are not sent to the standby servers.

More information about Private Streams is listed in the table below.

DESCRIPTION	SECTION
Private Streams concept	3.2.1.13 Private Stream
Private Streams overview	7.1.12 Private Streams

Table 80: Further Reading: Private Streams

13.6 Service Group

Service Group is a collection of services which appears like a single service to a Consumer application³⁹. For example, price information regarding TRI can be provided by multiple providers, each using a unique service name and ID. When a consumer requests price information for TRI, the consumer will need to specify the service (and therefore the vendor) from which it 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 typically grouped together to provide redundancy in case one of the services goes down, or to provide ease of use for combining services having different qualities of service. Services in the group do not need to be similar; Service Groups are a union of all offered services.

Services in a group are not hidden from the Consumer, which will still see information in the directory messages about all services, including the service group. Services in a group are routed within Session Layer, which ranks its potential routes according to a set of criteria.

Consumers can specify a service group in item request which is similar to specifying a concrete service with `AttribInfo.ServiceName` property. The difference is that the local configuration for an application will be set up two or more services into a group that RFA is aware of.

For example, two service combined in a group would be configured as shown below:

```
# 2-service grouped session
\ServiceGroups\SG2\serviceList = "SERVICE1,SERVICE2"
\Sessions\SvcGroupSession2\serviceGroupList = "SG2"
\Sessions\SvcGroupSession2\connectionList = "Connection_RSSL"
\Sessions\SvcGroupSession2\groupStatusFanoutEnabled = false
```

Example 213: Two Service Group Configuration Example

More information about Service Groups is listed in the table below.

DESCRIPTION	SECTION
Service Groups concept	3.2.1.13 Private Stream
Service Group renaming/aliasing	14.10 Using Service Groups: Service Renaming / Aliasing

Table 81: Further Reading: Service Groups

³⁹ Service Groups are a Consumer concept. A service provider does not know whether its service has been grouped by any Consumer.

13.7 Connection and Item Recovery

RFA automatically performs connection recovery on behalf of Consumers and Non-interactive Providers. If a connection is lost, RFA periodically attempts⁴⁰ to reconnect until the connection is reestablished. Once a connection is reestablished, RFA will re-login on behalf of the application using the same credentials used during the initial login. If the application is a Consumer and requested **SingleOpen** in the login request, RFA will also perform item recovery.

SingleOpen is a login attribute sent by the Consumer specifying that it would like automatic item recovery performed for all streaming item requests that it makes. For example, assume that the connection to a provider was lost. If the client requested **SingleOpen**, RFA will re-request all streaming items from the provider as soon as the connection is reestablished. However, if **SingleOpen** was not request by the client, the client is responsible for re-requesting all items when the connection is reestablished.

RFA provides another configuration parameter, **maxRetryCount**, that determines the number of times that RFA will retry an unsuccessful connection. By default RFA will retry the connection indefinitely; however, if **maxRetryCount** is set to some positive number, RFA will only retry the connection that number of times.

More information about this topic is listed in the table below.

DESCRIPTION	SECTION
Connection and Item Recovery concepts	3.2.1.10 Connection Recovery 3.2.1.11 Item Recovery and SingleOpen
Login and connection overview	9.1.2 OMM Concepts
Making a consumer login request	11.3.1 Login
maxRetryCount , serverList configuration parameters	<i>RFA Configuration Guide .NET Edition</i>

Table 82: Further Reading: Connection and Item Recovery

⁴⁰ As specified by the configuration parameter **connectRetryInterval**, which defaults to 15 seconds.

13.8 Item Group Management

Item groups can be used 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.

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 [ServiceId](#) is also considered the values should be unique. A Consumer application should track the [ServiceId/GroupId](#) 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 keep in mind that each item group must be 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 suspected while all items from any other upstream services remain in their prior state.

State change indications and item group merges occur via group status messages. A group status message is communicated to the RFA via the Source Directory domain message model. If a consumer subscribes to the Source Directory, it may receive the group status messages. The subscription is optional, since the RFA processes the information and always sends a status event to the consumer application when a group merges or a group status changes.

State change indications and item group merges occur via group status messages. A group status message is communicated to the RFA via the Source Directory domain message model. If a consumer subscribes to the Source Directory, it may receive the group status messages. The subscription is optional, since the RFA processes the information and always sends a status event to the consumer application when a group merges or a group status changes.

- For more information on Item Groups, refer to Section 7.1.11.
- For an Item Group scenario, refer to Section 7.1.11.3.

13.9 Connection Redirection (Load Balancing)

Connection Redirection addresses the need for a dynamic and balanced provider discovery. Instead of manually configuring an RFA application to use a particular provider, an RFA application has the ability to redirect itself based on the server information it receives from a provider.

The redirection process itself is transparent to the application. If a redirection occurs, the initial login response is not forwarded through RFA and only the login response of the target server is used for matching the login response parameters.

An OMM provider application should provide a list of known providers in its network (possibly including itself).

The `RSSL_Cons_Adapter` provides the capability for downloading the configured list of servers. Unless the adapter configuration parameter **disableDownloadConnectionConfig** is set to true, RFA automatically requests the list of servers by specifying the **DownloadConnectionConfig** login attribute. RFA will redirect to the first server on the list. The redirection process is deemed complete when one of the following conditions occur:

- 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 (**maxNumRedirection**) is reached.
- When the server does not return **ConnectionConfig** in the payload or sends a response without a payload.

For more details on login attributes and the login response payload, refer to **Section 3.2.1.14** and the *RFA RDM Usage Guide .NET Edition*.

13.10 Tunneling

Consumer and provider applications can also establish connections via tunneling over the Internet. When tunneling using HTTP, standard RWF messages leverage HTTP headers to tunnel through proxy servers and Internet-based routers. When tunneling using encryption, standard RWF messages are encrypted using Transport Layer Security (TLS). Such tunneling provides a level of security for transmitted data by encrypting the streams that flow over the established connection. These extensions to the connection allow for an OMM Consumer to connect to a provider that is not necessarily deployed at the same site.

When using Encrypted RWF data, some type of SSL Accelerator component is required on the provider site. The SSL Accelerator is responsible for decrypting any data from the Consumer prior to passing it to the Provider or ADS, and it is responsible for encrypting any data from the Provider or ADS prior to passing it to the Consumer. This component should be configured based on its own specific guidelines.

Due to the overhead associated with encrypting and decrypting data and addition and removal of additional HTTP and SSL headers, performance is impacted while using tunneling. In addition, connection throughput and latency will be directly related to the sizing of the connection through the Internet.

RFA tunneling leverages the Microsoft WinInet Internet communication library. As a result, the tunneling functionality is available only on Windows platforms. For the specific RFA tunneling configuration parameters, `tunnelingObjectName`, `tunnelingReconnectionTime`, and `tunnelingType`, see the *RFA Configuration Guide .NET Edition*.

For more Information about Tunneling refer to **Section 4.2.7**.

13.11 Pause and Resume

A common scenario in RFA display applications is to subscribe to a large set of data but display only a small portion of that data at any given time. The rest of the data might be hidden in other windows or tabs, or scrolled out of view. RFA allows applications to make a Pause request to pause data streaming and a Resume request at a later time. Applications can also send Pause All and Resume All requests in a single message to pause or resume all items associated with the application's login handle. This reduces unnecessary data and bandwidth usage, as well as the overhead of unsubscribing and resubscribing to entire sets of items.

Applications should check the [SupportOptimizedPauseResume](#) element of the login response to determine whether a provider supports Pause and Resume. If a provider does not support Pause and Resume, RFA will internally drop the pause or resume request.

A pause request is only a request and not a guarantee that the request will be fulfilled. In various cases, the infrastructure does not support pause functionality (e.g., multicast distribution). Also, because some updates might still be in transit on the network or queued in the client when a pause request is sent, the application must still be prepared to process updates after submitting the pause request. Also, because multiple updates are conflated on a just-in-time basis, some messages might be sent even when the stream is paused. For example, a pause might be sent after the item request but before the image is received.

Consumers issue a pause request by calling [ReissueClient\(\)](#) with the handle of the item to be paused and with the [InteractionType](#) set to [ReqMsg.InteractionTypeFlag.Pause](#). Conversely, a resume request sets the [InteractionType](#) (back) to [ReqMsg.InteractionTypeFlag.InterestAfterRefresh](#). To request a Pause All or Resume All, Consumers do the same thing but use the login handle instead of the item handle. More information about Pause and Resume, refer to 0.

13.11.1 Optimized Pause and Resume

Pause and resume is available in two flavors: the original Pause and Resume (PAR), and Optimized Pause and Resume (OPAR). OPAR has superseded PAR in most cases. Further discussion of PAR assumes OPAR, which extends and refines PAR in several key areas:

- OPAR is not limited to MarketPrice and Login domains. OPAR extends the PAR rules to any domain, whether a Refinitiv domain or a customer-defined domain model.
- OPAR is designed to work in conjunction with Warm Standby.
- OPAR supports a single-message “pause all” or “resume all.” RFA can pause or resume all subscribed items with a single message instead of internally fanning out the pause request to all subscribed items. Providers that support OPAR return [SupportOptimizedPauseResume](#) = 1 in the login response; this is handled internally by RFA and is transparent to the application.

The RFA interface between PAR and OPAR is the same from a user perspective.

13.11.2 Pause and Resume Message Flow

The following diagram shows message flow in a typical pause and resume scenario (when supported by the server).

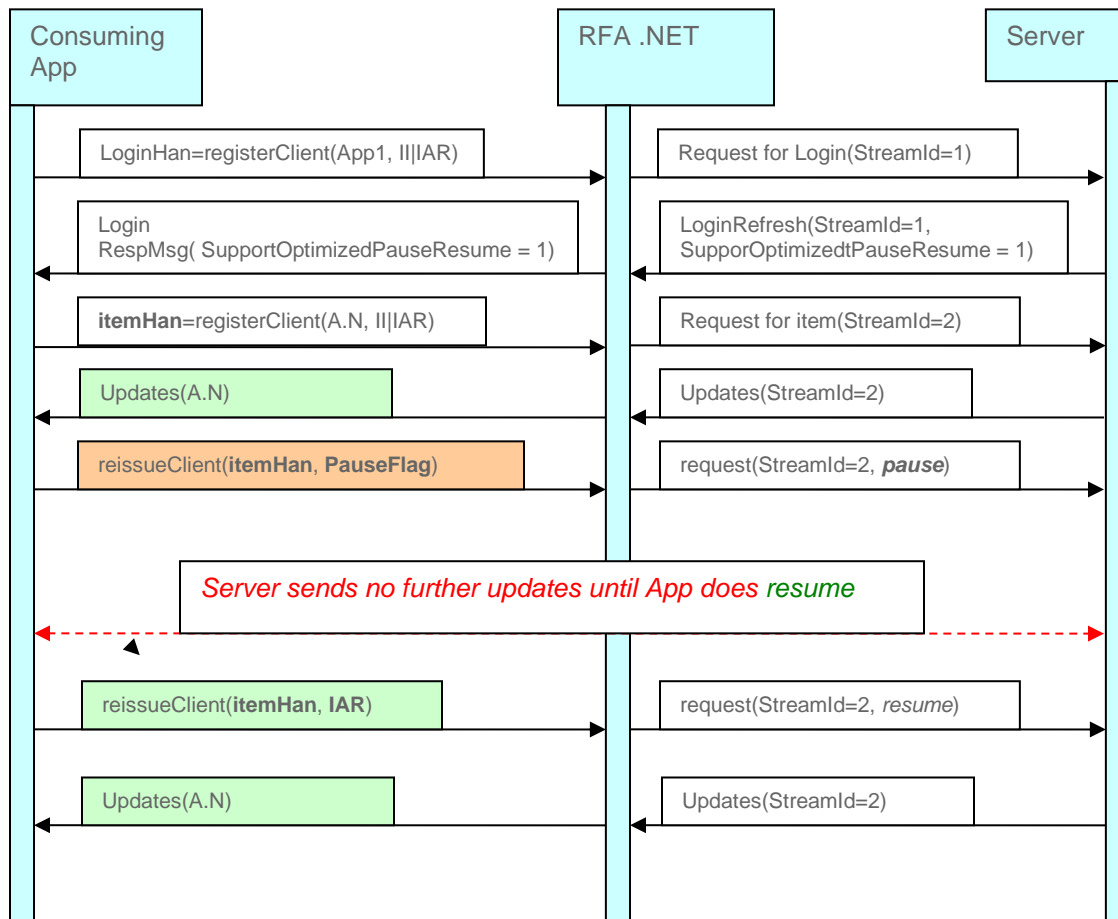


Figure 102: Pause and Resume Message Flow (II = InitialImageFlag, IAR= InterestAfterRefreshFlag)

13.11.3 Handling Handle

When an application makes multiple requests for the same item that is routed to the same connection, RFA manages a single network stream and fans out the network messages as appropriate. Consequently, RFA can only send a pause request to the network if all handles for that item have been paused.

For example if an application requests TRI.N twice it will have two separate handles. If one handle is paused, RFA will remember the paused state for that handle but will not send a pause request to the server. As a result, both application clients for both handles will continue to receive all updates. The paused client must be prepared to process these updates. If the other handle is paused at a later time, RFA will then send the pause request to the server and both handles will stop receiving updates at the same time. If either one of the handles resume, RFA sends a resume request to the server and both handles will receive updates again.

Applications that do not want this behavior must ensure only one request per item is made to RFA. They must also manage the message fanout themselves.

13.11.4 Pause All and Resume All

When using non-optimized pause and resume, sending “pause all” and “resume all” messages result in RFA internally fanning out the message to the set of subscribed items. A single-message “pause all” and “resume all” is only supported in optimized pause and resume.

The following code snippet illustrates sending “pause all” and “resume all” requests using the login handle:

```
// Login request
long loginHandle = ommConsumer.RegisterClient( eventQueue, ommItIntSpecLogin, this, null );

// subscribe 2 items, A.N and B.N
long itemAhandle = ommConsumer.RegisterClient( eventQueue, ommItIntSpec, this, null );

// ...

long itemBhandle = ommConsumer.RegisterClient( eventQueue, ommItIntSpec, this, null );

// Send pause all using the login handle
reqMsgLogin.InteractionType = ReqMsg.InteractionTypeFlag.Pause;
ommItIntSpecLogin.Msg = reqMsgLogin;

ommConsumer.ReissueClient( loginHandle, ommItIntSpecLogin );

// Send resume all using the login handle
reqMsgLogin.InteractionType = ReqMsg.InteractionTypeFlag.InterestAfterRefresh;
ommItIntSpecLogin.Msg = reqMsgLogin;

ommConsumer.ReissueClient( loginHandle, ommItIntSpecLogin );
```

Example 214: Pause All and Resume All

13.12 Warm Standby

RFA can be configured to failover to a standby connection in the event the primary connection fails. This feature is known as Warm Standby. Because the standby connection is already aware of items an application has registered interest for, during a failover RFA does not need to re-request open items between an OMM provider and consumer. Warm Standby not only reduces overall recovery time, but also network traffic by not inducing a “packet storm” with a flurry of re-requests.

The consumer application receives updates only on the item streams opened on the active server; it does not get updates, status, unsolicited refreshes, or Generic Message from the standby server(s). If the active server fails, RFA notifies the next server in the standby list that it is the new active server. That server then begins sending data without the consumer application needing to re-request the items.⁴¹

This process is transparent to the Consumer application. Standby Provider servers must be configured so that they provide similar login responses and provide similar directories, but the only necessary Consumer set up is to configure RFA to use a server list via the **serverList** configuration parameter, as shown in the Windows registry configuration example below.

```
\Connections\Connection_RSSL\connectionType      = "RSSL"
\Connections\Connection_RSSL\hostName             = "dataserver1"
\Connections\Connection_RSSL\rsslPort             = "14002"

\Connections\Connection_RSSL\serverList           = "warmStandbyGroup1"
\Connections\Connection_RSSL\disableDownloadConnectionConfig = "false"
\Connections\Connection_RSSL\maxNumRedirection    = 1

\StandbyLists\warmStandbyGroup1\startingActiveServer = "dataserver1:14002"
\StandbyLists\warmStandbyGroup1\serverList        = "dataserver2:14003, dataserver3:14004"
```

Example 215: Warm Standby Configuration Example

More information about Warm Standby, refer to section 3.2.1.12, Warm Standby. Warm Standby can provide a secondary, standby message stream between an RFA provider and an RFA consumer. In the event of a failure of the primary provider, an RFA-driven failover to the second stream occurs. This is a faster type of failover because it does not re-request all open items and instead just resumes processing with the back-up stream.

The following diagram illustrates the order of events when using the Warm Standby feature:

⁴¹ Items on Private Streams are by design not recovered.

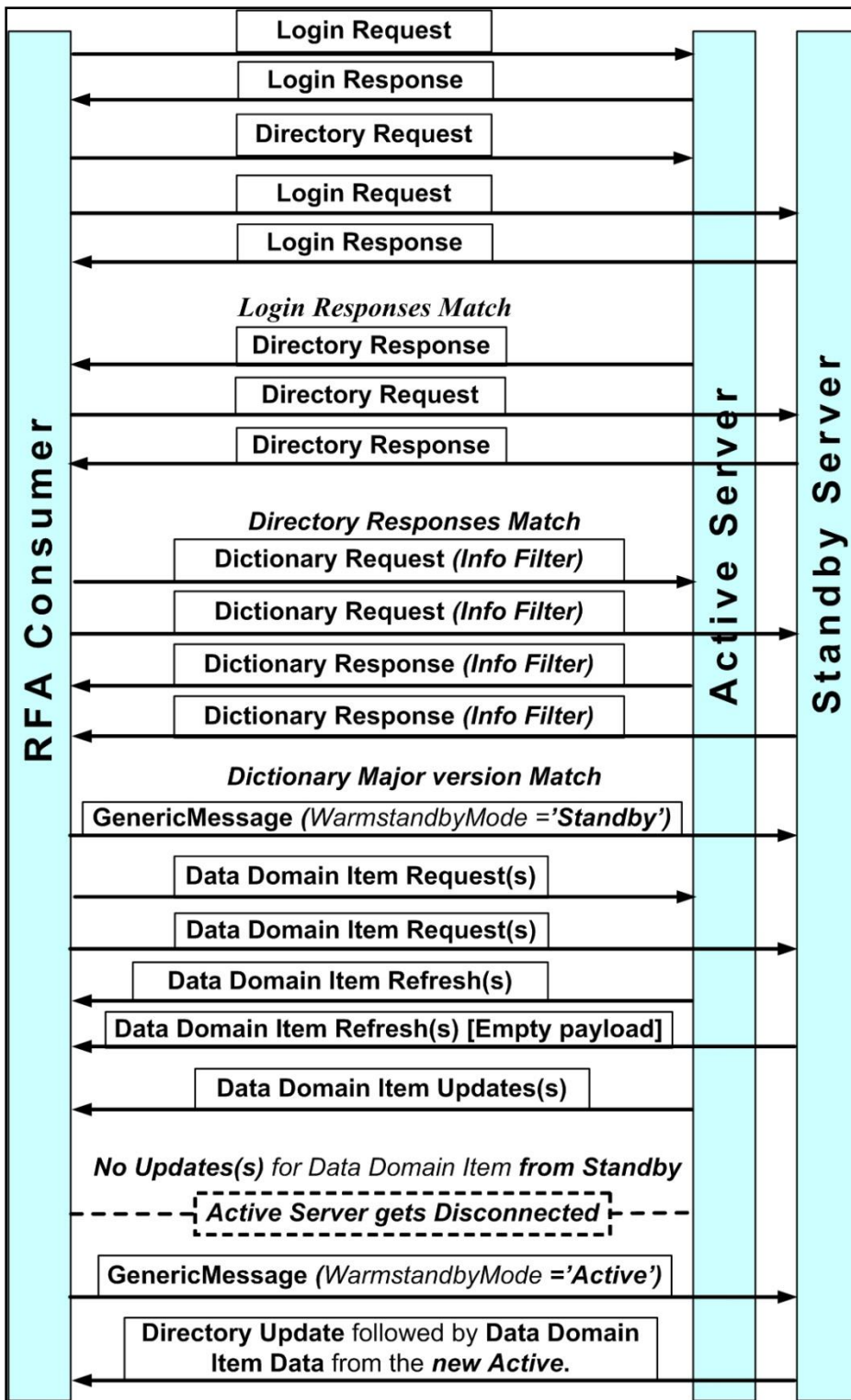


Figure 103: Order of events when warm standby is active

13.12.1 Effects in the Consumer

RSSL-type adapters (e.g., RSSL_Cons_Adapter) can be configured to provide warm standby capability. The consumer application must be configured to connect to two or more servers, out of which one is designated as an active server and

the rest are standby servers. With this configuration RFA will internally request the same items from both active and standby servers.

The consumer application receives updates only on the item streams opened on the active server; it does not get updates, status, unsolicited refreshes, and Generic Message from the standby server(s). If the active server fails, RFA notifies one of the standby servers that it is the new active server. That server then begins sending data without the consumer application needing to re-request the items. The server that becomes active after the failover should begin sending any conflated updates containing all changed data for conflatable domains and send unsolicited refresh messages followed by updates for any non-conflated domains. This process brings active data streams back to their prefailover state.

If the failed server comes back online, RFA does not switch back to it. It becomes one of the servers in the standby list. For more information on warm standby configurable parameters see the *RFA Configuration Guide .NET Edition*.

Internally, the RFA consumer-side sends a ConsumerConnectionStatus [GenericMsg](#) on an [MMT_LOGIN](#) domain to indicate to the provider applications whether they should operate as an active or standby server. Refer to the *RFA RDM Usage Guide .NET Edition* in the [MMT_LOGIN](#) section for the details on this message.

After establishing a connection to the standby server, RFA internally obeys the following rules for Warm Standby:

- The Standby server's [MMT_LOGIN](#) response must match the following login attributes:
 - **ApplicationId**
 - **PositionId**
 - **ProvidePermissionProfile**: If requested by the user, the values received from all servers must match what was requested
 - **ProvidePermissionExpressions**: If requested by the user, the values received from all servers must match what was requested
 - **SingleOpen**
 - **AllowSuspectData**
 - **SupportPost**
 - **SupportBatchRequests**
 - **SupportOMMPost**
 - **SupportViewRequests**

If any of these elements does not match then RFA disconnects the standby server.

- The standby server's [MMT_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 [MMT_DICTIONARY](#) response must match the major version; otherwise, RFA disconnects the standby server.

- RFA routes Generic Message only on streams in administrative domains (e.g., ConsumerStatus Generic Message, ConsumerConnectionStatus Generic Message). Generic Message on data domain streams are not routed to the standby server(s).
- Post Message is routed to the active and all standby servers to ensure data on all servers is synchronized.

If the active server fails, when switching to the new active server, RFA does the following:

- Logs a message stating that it switched to a new active server. RFA also sends a `ConnectionStatus.StateEnum.Up` event with a new status code `ConnectionStatus.StatusCodeEnum.ServersSwitched` to the consumer application.
- For all services that were offered by the old active server but are not offered by the new active server, RFA sends a directory update message stating that the service is down.
- For an item open on the failed server but no longer open on the newly active server, RFA sends an internally-generated status message with stream state `RespStatus.StreamStateEnum.ClosedRecover` to the consumer application.

13.12.2 Effects in the Provider

An RFA Provider application must notify the client if the server supports warm standby via the SupportStandby login attribute. 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 the login stream. A providing application operating in standby mode must send a refresh with an empty payload in response to an item request for data domains.

NOTE: This applies only to providers that are directly connected to consumers. Providers connected to RTDS components such as an ADH/ADS don't need to be concerned with supporting warm standby.

For more details refer to the *RFA RDM Usage Guide .NET Edition*.

When a providing application operating in a standby mode receives a ConsumerConnectionStatus Generic Message on a login stream indicating that it should now operate in active mode, the providing application must do the following:

- Send a full update on the directory.
- If data is being conflated, send an update 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 data state back to `RespStatus.DataStateEnum.OK`. After this, the providing application should send update messages as normal.

13.13 Horizontal Scaling

Horizontal Scaling is a feature of RFA that enables applications to use multiple instances of consumer connections on multi-core processors and dynamically scale the number of instances that applications use. Applications can use this feature to increase their total throughput for both Response Message and Request Message (measured in number of messages per second).

13.13.1 Effects in the Consumer

The benefit of Horizontal Scaling in the consumer comes from the fact that each instance of the `RSSL_Cons_Adapter` processes messages on its own unique connection independently of other `RSSL_Cons_Adapter` instances. To enable this feature, the application must set the **singleton**⁴² configuration variable to false as follows:

```
\Adapters\RSSL_Cons_Adapter\singleton = False
```

Example 216: Configuration for enable Horizontal Scaling in `RSSL_Cons_Adapter`

With Horizontal Scaling, RFA maintains a one-to-one relationship between Session instance and `RSSL_Cons_Adapter` instance, and each individual session does not share its resources. The application can then scale by acquiring and releasing Sessions as needed. Each instance of the Session/`RSSL_Cons_Adapter` pair is independent of the others. Applications can configure any Sessions for either single or dual-threaded.

⁴² **singleton** defaults to true: the Horizontal Scaling feature is disabled and RFA uses a single shared instance of the `RSSL_Cons_Adapter`.

The following diagram is a high-level overview of an application using the Horizontal Scaling feature and the flexibility of the configuration. The application processes messages on three connections using three [OMMConsumers](#) and three Sessions. One Session is configured for throughput (dual-threaded) while the other two are configured for latency (single-threaded). The first two [OMMConsumers](#) use [EventQueues](#) while the third uses the callback model. Additionally, the application has at least two worker threads that make requests and/or process messages from the [EventQueues](#). Thus, the first part of this multi-threaded application follows the Client Model with throughput configuration, the second part follows the Client Model with latency configuration, and the third follows the Callback Model with latency configuration. For more information, refer to Sections 14.3 and 14.4. The configuration is illustrated in the following diagram.

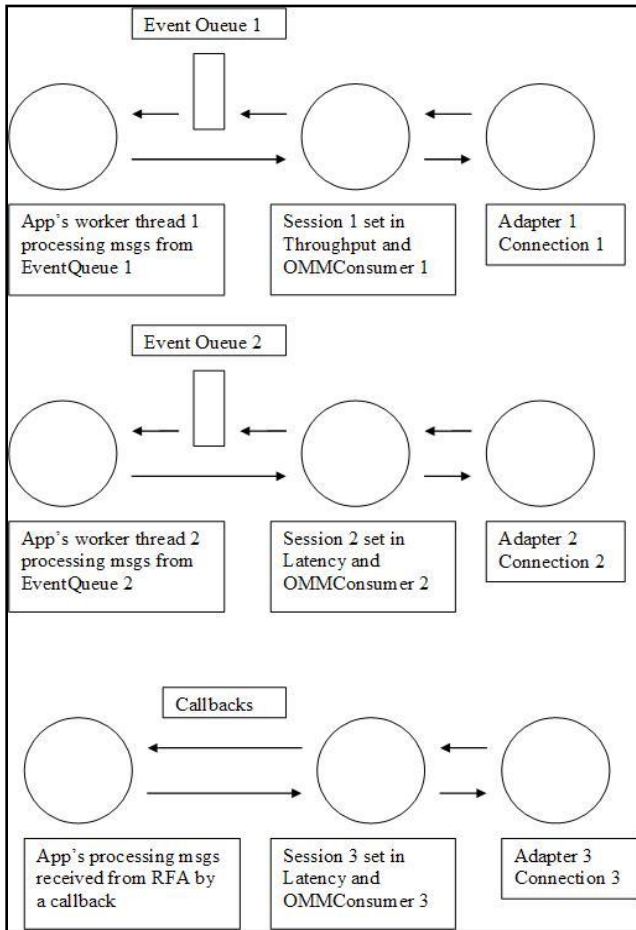


Figure 104: Horizontal Scaling Application

The following set of steps initializes this application:

1. Application initializes [ConfigDatabase](#) and merges its configuration with RFA's configuration.
2. Application acquires three separate instances of [Session](#) using unique session names as configured.
3. Application creates two [EventQueues](#).
4. Application creates three [OMMConsumers](#), one on each [Session](#).
5. Application creates two worker threads.
6. Application issues login and item requests for each [OMMConsumer](#).
7. Using two worker threads, application dispatches events / messages from the two [EventQueues](#).

13.13.2 Effects in the Provider

Horizontal Scaling can also be used to enable multiple instances of RSSL_Prov Adapters and dynamically scale the number of instances that applications use as needed. By utilizing this feature on multi-core processors, applications can increase their total throughput for both Response Message and Request Message. This benefit comes from the fact that each instance of RSSL_Prov_Adapter processes messages on its own unique connection independently of other RSSL_Prov_Adapter instances. To enable this feature, the application must set the **singleton** configuration variable to **false** as follows:

```
\Adapters\RSSL_Prov_Adapter\singleton = False
```

Example 217: Configuration for enable Horizontal Scaling in RSSL_Prov_Adapter

When enabled, RFA maintains a one-to-one relationship the between RSSL_Prov_Adapter and Session instances and each individual session does not share its resources. The application can then scale by acquiring and releasing Sessions as needed. Each instance of the Session/RSSL_Prov_Adapter pair is independent of the others.

13.14 Service ID vs. Service Name

There are two ways for an RFA provider to generate a service ID for a service name. The first is that the Provider encodes only a service name in the directory domain. RFA on the providing side will internally assign a service ID for that service name as a map entry key when sending the directory message. The other way is that the Provider assigns the service ID in the Directory domain Info Filter. RFA then validates the service ID and sets a valid ID as a map entry key when sending the directory message.

Allowing the provider to assign the ID benefits provider applications that want to use the same service ID which they assigned to create DACS locks for content they publish. If the service ID assigned by provider client is invalid, the Directory message published will be dropped and a command error will be sent to provider.

RFA internally maps service name to service ID. RFA usually uses service names for requests like item specification, but there are some cases where an application must use a service ID instead. For example, Generic Message must use a service ID to send a message on a service that is not known by RFA⁴³.

Applications can set the service ID on the [AttribInfo](#) of a message using the [AttribInfo.ServiceID](#) property. Applications should set either service ID or service name on the [AttribInfo](#) of a [Msg](#), but not both. An invalid API usage exception will be thrown by RFA if an application tries to set both. Provider applications can retrieve the service ID of the service to which the item was requested by using the [AttribInfo.ServiceID](#) property.

NOTE: The RFA consuming side internally sets the Service ID of a service in the Directory Domain INFO filter. If the consuming application is configured to use service groups, RFA sets an internally generated service ID in the Directory domain message. Refer to the [under the directory domain's INFO filter section](#).

RFA conforms to the following rules when validating service IDs on messages:

- If a consumer application sets a service ID on the [AttribInfo](#) for a [ReqMsg](#), RFA validates the service ID set on the request against the list of services mapped to their corresponding service IDs. If the service ID is not valid then the RFA consuming side sends a status message with a stream state **Open** and data state **Suspect**, and the request is not routed to the network.
- If a provider application sets a service ID on the [AttribInfo](#) for a [RespMsg](#) on an item, then RFA validates the Service ID set on the request against the list of services mapped to their corresponding Service IDs. If the Service ID is not valid then RFA routes the message with a service ID of 0 on the message.
- If a consumer application sets a service ID on the [AttribInfo](#) for a [PostMsg](#) on a stream other than a login stream (i.e., on-stream), then RFA replaces the service ID on the post message with the service ID value of the initial request when the item stream was opened.
- If a consumer application sets a service ID on the [AttribInfo](#) for a [PostMsg](#) on a login stream (i.e., off-stream), then RFA validates the service ID set on the post message against the list of published services. If the service ID is not found in the list of corresponding service IDs, then RFA sends a command error to the application and does not route the post message to the network. (This is similar to how a service name is validated when an application sets a service name).
- Applications can set only the service ID on the [AttribInfo](#) of a [GenericMsg](#). When a consumer application sets a service ID on the [AttribInfo](#) for a [GenericMsg](#), RFA routes the message to the network without validating the service ID. The service name on the [AttribInfo](#) for a [GenericMsg](#) is not used. If the application tries to set the service name then the message is routed to the network with a service ID set to 0 on the message.

⁴³ An RFA consumer can also discover the service ID of a service from the source directory response message.

13.15 RDMFieldDictionary Utility

Some data types require the use of a dictionary for encoding or decoding. This dictionary typically defines type or formatting information and directs the application on how to encode or decode specific pieces of information. Content that uses the FieldList type requires the use of a field dictionary. Usually this dictionary is the RDMFieldDictionary supplied by Refinitiv, although it could also be a user-defined or modified field dictionary.

The Data Dictionary can be either downloaded from the infrastructure or loaded from the local filesystem. For more information, refer to Section 4.1.4.

13.15.1 Consumer Handling

The Source Directory message should inform the consumer of any dictionaries required to decode the content provided on a service, as well as the dictionaries available for download. A Consumer application can determine whether to load necessary dictionary information from a local file or download the information from the provider if the provider makes this information available.

- If loading from a file, RFA offers several utility functions to load and manage a properly-formatted field dictionary.

```
public bool LoadDictionaryFromFile(RFA_String rdm_field_dict_path, RFA_String enumtype_def_path)
{
    if (dictionaryLoaded)
    {
        return true;
    }

    try
    {
        rdmFieldDict.ReadRDMFieldDictionary(rdm_field_dict_path);
        rdmFieldDict.ReadRDMDictEnumTypeDef(enumtype_def_path);
        rdmFieldDict.Version = new RFA_String("1.1");
        rdmFieldDict.DictId = 1;
        dictionaryLoaded = true;
    }
    catch (InvalidUsageException)
    {
        AppUtil.Log(AppUtil.LEVEL.ERR, "Decoder.LoadDictionaryFromFile() failed to load dictionary\r\n");
    }

    return dictionaryLoaded;
}
```

Example 218: Loading dictionary from file

- Downloading the dictionary, the application issues a request using the Dictionary domain model. The provider application should respond with a dictionary response, typically broken into a multi-part message. RFA offers several utility functions for encoding and decoding Dictionary domain content.

```
public void SendRequest()
{
    reqMsg.Clear();
    attribInfo.Clear();

    // Constructed according to the RDM Usage Guide
    reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DICTIONARY;
```

```

    reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
    ReqMsg.InteractionTypeFlag.InterestAfterRefresh;

    attribInfo.DataMask = RDM.Dictionary.DICTIONARY_VERBOSITY_VALUES.DICTIONARY_NORMAL;
    attribInfo.ServiceName = appConfig.ServiceName;
    attribInfo.Name = networkDictName1;

    reqMsg.AttribInfo = attribInfo;
    ommItemIntSpec.Msg = reqMsg;
    dictHandle1 = parentConsumer.AppRegisterClient(ommItemIntSpec, this, null);

    attribInfo.Name = networkDictName2;
    reqMsg.AttribInfo = attribInfo;
    dictHandle2 = parentConsumer.AppRegisterClient(ommItemIntSpec, this, null);
}

```

Example 219: Loading dictionary from file

The utility functions used in both instances and the information about Dictionary domain and expected content formatting are described in more detail in the .

13.15.2 Provider Handling

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

```

private void EncodeDirectoryArray(Reuters.RFA.Data.Array array)
{
    ArrayWriteIterator arrWit = new ArrayWriteIterator();
    arrWit.Start(array);

    // Specify Dictionary as a capability
    DataBuffer dataBuffer = new DataBuffer();
    ArrayEntry arrayEntry = new ArrayEntry();
    uint mType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_DICTIONARY;
    dataBuffer.UInt = mType;
    arrayEntry.Data = dataBuffer;
    arrWit.Bind(arrayEntry);

    // ...

    arrWit.Complete();
}

```

Example 220: Dictionary for notifying consumer

When the provider receives a dictionary request it should create a `RespMsg` with the dictionary as the payload of the message. The following example shows process to handle dictionary request and sending dictionary to the consumer.

```
private void ProcessDictionaryReq(OMMSolicitedItemEvent evnt)
{
    ReqMsg reqMsg = evnt.Msg as ReqMsg;
    RequestToken reqToken = evnt.RequestToken;
    long clientSessionHandle = reqToken.Handle;

    if (isDictionaryDataRead == false)
    {
        rdmFieldDictionary.ReadRDMFieldDictionary(cfgVariables.RDMFieldDictFilePath);
        rdmFieldDictionary.ReadRDMDenumTypeDef(cfgVariables.EnumTypeDefFilePath);
        rdmFieldDictionary.DictId = 1;
        isDictionaryDataRead = true;
    }

    RespMsg respMsg = new RespMsg();
    Series dictionaryBody = new Series();
    RespStatus respStatus = new RespStatus();
    respStatus.StreamState = RespStatus.StreamStateEnum.Open;
    respStatus.DataState = RespStatus.DataStateEnum.Ok;
    respStatus.StatusCode = RespStatus.StatusCodeEnum.None;
    RFA_String tmpStr = new RFA_String("RequestCompleted");
    respStatus.StatusText = tmpStr;

    long handle = evnt.RequestToken.Handle;
    if (handle != 0)
    {
        dictionaryBody.SetAssociatedMetaInfo(handle);
    }
    if (reqMsg.AttribInfo.Name == "RWFFld")
    {
        AppUtil.Log(AppUtil.LEVEL.TRACE, "<- Received MMT_DICTIONARY Field dict request");
        encoder.EncodeDictionaryMsg(respMsg, reqMsg.AttribInfo, respStatus);
        rdmFieldDictionary.EncodeRDMFieldDictionary(dictionaryBody);
    }
    else if (reqMsg.AttribInfo.Name == "RWFEnum")
    {
        AppUtil.Log(AppUtil.LEVEL.TRACE, "<- Received MMT_DICTIONARY Enum dict request");
        tmpStr.Clear();
        tmpStr.Append(reqMsg.AttribInfo.Name);
        encoder.EncodeDictionaryMsg(respMsg, reqMsg.AttribInfo, respStatus);
        rdmFieldDictionary.EncodeRDMDenumDictionary(dictionaryBody);
    }
    respMsg.Payload = dictionaryBody;

    OMMSolicitedItemCmd itemCmd = new OMMSolicitedItemCmd();
    itemCmd.Msg = respMsg;
    itemCmd.RequestToken = reqToken;

    SubmitCmd(itemCmd, null, new RFA_String("MMT_DICTIONARY Refresh"));
}
```

Example 221: Dictionary Handling in Provider

For more information, refer to the `StarterProvider_Interactive` example in the RFA package.

There are also utility functions provided to help the provider encode in an appropriate format for downloading.

The utility functions used in both instances and the information about Dictionary domain and expected content formatting are described in more detail in the *RFA RDM Usage Guide .NET Edition*.

13.16 News Headlines

The RTDS deliver news headlines, alerts and corrections via the N2_UBMS item.

To properly receive news headlines, application must subscribe to the N2_UBMS item based on the RTDS.

13.17 Using the DACSLOCK API

RFA distributes authorization lock information in a binary format known as a DACS Lock. A DACS Lock typically consists of a service ID, a list of Permission Expressions (PEs), and an operator enumeration that indicates whether authorization is required for any one or all of the PEs. An application may manipulate or combine DACS Locks (also known as a composite DACS Lock) through use of the DACSLOCK API.

The DACSLOCK API provides capabilities to create or change a DACS Lock. For example, an application can append PE codes, remove PE codes, or change a service ID.

The DACSLOCK API also provides the ability to extract the raw binary DACS Lock. A providing application may use this feature to publish a DACS Lock to the RFA Session Layer. Additionally, the DACSLOCK API provides the ability to input a raw binary DACS Lock. A consuming application may use this feature after having received a DACS Lock from the RFA Session Layer. A hybrid application may input and extract raw binary DACS Locks in order to provide some value-add information (e.g., creating a composite DACS Lock).

The DACSLOCK API consists of three interfaces:

- **Authorization Lock Data:** an interface that represents the actual data of the DACS Lock. It provides a constructor that accepts a character reference and length to the raw binary DACS Lock. It provides methods such as the ability to change a service ID ([ChangeLock\(\)](#)), compare DACS Locks ([CompareLock\(\)](#)), and combine DACS Locks ([CombineLock\(\)](#)). The interface also provides the [LockData](#) property used to extract the raw binary DACS Lock.
- **Authorization Lock:** an interface that provides methods to modify authorization information. It provides methods such as the ability to append PEs ([AppendPE\(\)](#)) or remove all PEs ([RemoveAllPes\(\)](#)).
- **Authorization Lock Status:** an interface that contains status information following operations on the Authorization Lock Data and Authorization Lock interfaces. The Authorization Lock Status interface extends the status interface in the RFA Common Package.

This section provides an overview of the DACSLOCK API; for a comprehensive description of the DACSLOCK API, see the *DACSLOCK API Developers Guide .NET Edition* and *DACSLOCK API Reference Manual .NET Edition*.

13.18 Parsing AnsiPage Data

- The AnsiPage API offers a programming interface that has the ability to interpret data received from AnsiPage sources, as well as the ability to generate page data suitable for publishing.
- An RFA Consumer application can identify AnsiPage data by checking the data format of an `OMMItemEvent` or `RespMsg`. If the response message's `Payload.DataType` is `DataEnum.DataBuffer` and the data buffer's `DataBufferType` is `DataBufferEnum.ANSI_Page`, the data is AnsiPage data. `DataBuffer.Buffer` can be used to extract the raw ANSI buffer that can then be parsed by the AnsiPage API.
- The AnsiPage API provides an in-memory, page-representation (namely, the `Page` interface) that supports decoding of a raw ANSI update buffer. The process of decoding a page produces a set of page updates (namely, the `PageUpdate` interface), which contain positioning information typically used to track changes to a page. A raw ANSI update buffer may represent either an entire page or specific changes to a page. To use the AnsiPage API, an application typically creates an instance of this `Page` interface to hold the page representation.
- Upon receiving a raw ANSI update buffer, a consuming application typically calls the `Decode()` method on the `Page` object to apply the ANSI update buffer to the `Page`. The consuming application can then operate directly on the `Page` or use the individual `PageUpdate` objects to track changes to the `Page`.
- An application may operate on a `Page` object directly to extract information for display or perform operations on a specific cell within the page. The AnsiPage API includes a `PageCell` interface used to perform operations on the specific cells within a page.
- For details on usage of the various page-related interfaces, see *AnsiPage Developers Guide .NET Edition*. This document includes an example that creates a `Page` object and set of `PageUpdate` objects. The example describes the procedure to encode a set of `PageUpdate` objects into a raw ANSI update buffer. It then creates a second `Page` object and uses the raw ANSI update buffer to decode the set of `PageUpdate` objects and formulate an identical in-memory page representation. A consuming application performs this decode operation. The example also illustrates handling a subsequent raw ANSI update buffer. The subsequent buffer represents a change to the existing `Page` object.

An application may use an OMM Provider to create an AnsiPage service. For OMM Providers, the MarketPrice RDM is used.

- As indicated above, the AnsiPage API provides an in-memory page representation that supports encoding the combination of a page and set of page updates into a raw ANSI update buffer. The `PageUpdate` objects contain positioning information typically used to track changes to a page. A raw ANSI update buffer may represent either an entire page or specific changes to a page.
- To use the AnsiPage API, an application typically creates an instance of the `Page` interface to hold the page representation along with a set of `PageUpdate` objects. Next, the application then creates a raw ANSI update buffer via the `Encode()` method. The raw ANSI update buffer represents specific data within a page, referenced by the set of `PageUpdate` objects.
- Once an OMM Provider application has created a raw ANSI buffer, it sets the buffers in a `DataBuffer` object with `SetBuffer()`. The `DataBuffer` can then be set in the `RespMsg` with `Payload` property. The `RespMsg` should be set to MarketPrice with `MsgModelType` property.

13.19 Application Signing

System administrators can configure RTDS systems such that they require applications to include an 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 RTDS 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:

```
ReqMsg reqMsg = new ReqMsg();
reqMsg.MsgModelType = RDM.RDM.MESSAGE_MODEL_TYPES.MMT_LOGIN;
reqMsg.InteractionType = ReqMsg.InteractionTypeFlag.InitialImage |
ReqMsg.InteractionTypeFlag.InterestAfterRefresh;
AttribInfo attribInfo = new AttribInfo();

attribInfo.NameType = RDM.Login.USER_ID_TYPES.USER_NAME;
attribInfo.Name = cfgVariables.UserName;

ElementList elementList = new ElementList();
ElementEntry element = new ElementEntry();
DataBuffer elementData = new DataBuffer();
ElementListWriteIterator elwiter = new ElementListWriteIterator();

elwiter.Start(elementList);

//Encode Application Id, position etc. (as shown in other example code)

//Encode Application Authorization Token.
RFA_String appAuthToken = new RFA_String("appToken1");
element.Name = RDM.Login.ENAME_APPAUTH_TOKEN;
elementData.SetFromString(appAuthToken, DataBuffer.DataBufferEnum.StringAscii);
element.Data = elementData;
elwiter.Bind(element);

elwiter.Complete();
attribInfo.Attrib = elementList;
reqMsg.AttribInfo = attribInfo;
```

Example 222: Authorization Token Encoding Example

Chapter 14 Performance Consideration

There are many performance tradeoffs that can be made when configuring RFA that allow users to configure and tune RFA applications to perform at maximum performance in their specific environment.

RFA applications are structured into three main components called the Adapter, Session Layer, and Client, which is the part of the application written by the user or customer. Depending on the configuration and the application needs, there can be one or more of each component.

14.1 Thread

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 adapter and Session Layer. Or, it can be configured to use a separate thread for each component, including the client.⁴⁴ Threads are used to handle both incoming and outgoing messages. For example, an adapter thread reads Response Message from the connection and writes Request Message to the connection.

The primary advantage of using multiple threads is the work performed by RFA and/or the client can be divided 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 sections.

14.2 RFA Consumer Queue

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 adapter and Session Layer is called the response queue and is used to queue Response Messages from the connection. The queue used between the Session Layer and the client is called the event queue. If an application uses RFA without an event queue then a single thread of execution passes messages all the way from the connection to the application via a callback function.

In a typical configuration, having an event queue implies that the Session Layer and client will be using separate threads (though this is not always the case). Setting the **threadModel** configuration parameter to **Dual** specifies that a response queue should be used by RFA and also implies that the adapter and Session Layer will operate on their own threads.

The principal 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 also requires additional machine overhead, as well as performing a copy of the message.

There are some restrictions on when queues may be used. For example, if there are multiple adapters sending Response Message to the same Session Layer component, a queue must be used. In this case both adapters will put messages on the same response queue for the Session Layer to process.

⁴⁴ Some restrictions to this apply: for example, if there are multiple components of a specific type, then each component may need to have a separate thread of execution.

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. For more information on event queue groups refer to Section 5.1.1.3.

Event queues can also be configured to keep statistics. These statistics can be useful for applications to track their usage of the event queue. For more information on event queue statistics refer to Section 5.2.1.6.

NOTE: RFA incurs additional overhead in maintaining and dispatching messages from an event queue group. RFA also incurs additional overhead when keeping event queue statistics.

14.2.1 Response Queue Parameter

RFA provides configuration parameters for the response queue to balance latency versus throughput. These are also described in the *RFA Configuration Guide .NET Edition*.

14.2.1.1 responseQueueMaxBatchSize

The **responseQueueMaxBatchSize** parameter controls a threshold for the Session Layer's response queue. When the Adapter has placed this amount of messages on the queue, the Session will be notified to immediately begin processing them. In general, larger numbers provide better throughput at the potential cost of increased latency, while smaller numbers may provide lower latency at the potential cost of decreased throughput.

The Session Layer thread services both the response queue and the request queue. In order for it to service the request queue in a timely fashion, it should not spend too much time processing the response queue before checking the request queue.

14.2.1.2 responseQueueBias

The **responseQueueBias** parameter controls a number of Response Message that the Session Layer will attempt to dispatch each time it is notified to dispatch. The Session Layer is notified to dispatch by either the **responseQueueMaxBatchSize** number of responses being put onto the queue or by the **responseQueueBatchInterval** amount of time passing.

For example, if the response queue contains 1,000 messages and this parameter is set to 500, the Session Layer will only dispatch 500 messages from the queue and then wait until the next notification (by either **responseQueueMaxBatchSize** or **responseQueueBatchInterval**) that it should begin processing the response queue again.

14.2.1.3 responseQueueBatchInterval

The **responseQueueBatchInterval** parameter controls a timer to wake up the Session Layer to check the response queue for incoming messages. The default value is 15 milliseconds. This parameter works together with **responseQueueBias**.

For example, assume **responseQueueBias** is set to 300 and 299 messages are received in the first 15 millisecond interval. The **responseQueueBias** will not be hit. However, after the 15 millisecond interval has expired, the timer will trigger the Session Layer thread to wake up and begin processing the messages on the response queue. The first few messages that were placed on the queue may have been waiting for up to 15 milliseconds.

Alternatively, if 300,000 messages/sec are being received, then 300 messages/millisecond on average are being received and the **responseQueueBias** setting would have caused the Session Layer to wake up and process messages once every millisecond. The **responseQueueBatchInterval** is not triggered.

14.3 Threading Model

RFA uses three threading models which are discussed in detail below.

14.3.1 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 be used by OMM Consumers and OMM Providers.

14.3.2 Client Model

The Client Model is used when an event queue is used. 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 is generally used for high throughput situations because multiple threads are used.

14.3.3 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 want to be 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. A 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 doing something else.

14.4 Configuring 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.

14.4.1 Single Threaded with No Event Queue (Callback Model)

The single threaded no event queue configuration does not use any queues and processes Response Messages with a single thread, as shown in the diagram below.

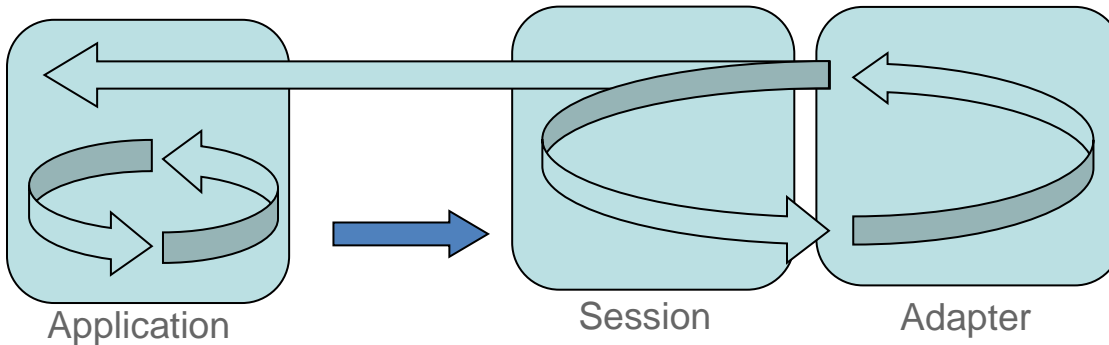


Figure 105: Single Threaded with no Event Queue (Callback Model)

There are two threads in this configuration. There is a main application thread that is started by the OS and a single RFA thread. The application thread is used to send requests. The RFA thread is used for the adapter, Session Layer, and for the application callbacks that process Response Message using `ProcessEvent()`.

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.

NOTE: Trying to overcome this limitation by creating another application thread and sending the messages to it makes this configuration very similar to the low latency with an event queue configuration.

- 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 Message. Conversely, Request Message may incur outbound latency if a large number of Response Message are being received.

The model is enabled by passing NULL for the `EventQueue` argument of `RegisterClient()` and by setting `threadModel=Single`⁴⁵ in the Session Layer configuration.

⁴⁵ For more information about ThreadModel (formerly OMMPerfMode), see Section Chapter 2,

14.4.2 Single Threaded with an Event Queue (Client Model)

The single threaded with an event queue configuration has an event queue, an application thread, and an RFA thread, as shown in the diagram below.

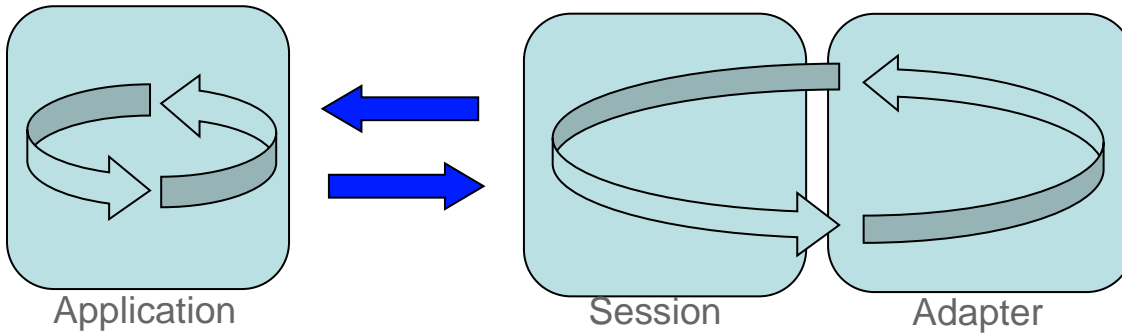


Figure 106: Single threaded with an Event Queue (Client Model)

There are a total of two threads in this configuration. The main application thread that is started by the OS is used to send requests and to process Response Messages. The second thread is used by RFA for adapter and Session Layer processing.

Advantages:

- Considered a low latency configuration.
- 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” single threaded configuration due to the use of an event queue.

The model is enabled by specifying an `EventQueue` argument in `RegisterClient()` and by setting `threadModel=Single` in the Session Layer configuration.

14.4.3 “Full” Dual-Threaded Configuration (Client Model)

The “full” dual-threaded configuration has an event queue, a response queue, an application thread, a Session Layer thread, and an adapter thread, as shown in the diagram below.

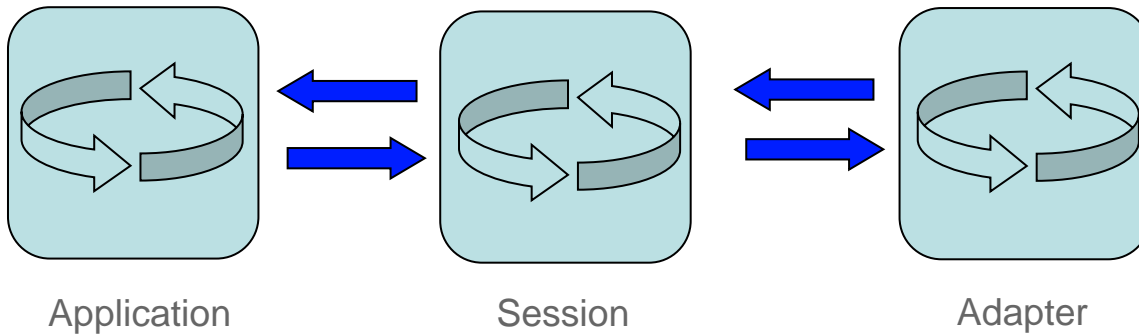


Figure 107: Full Dual-Threaded Configuration (Client Model)

There are a total of three threads in this configuration. The main application thread that is started by the OS 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 adapter to read and write messages to the connection.

Advantages:

- The three threads maximize throughput by allowing 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.

The model is enabled by specifying an `EventQueue` argument in `RegisterClient()` and by setting `threadModel=Dual` in the Session Layer configuration.

14.4.4 Dual-Threaded with No Event Queue (Callback Model)

The dual-threaded with no event queue configuration has a response queue and two threads to process response messages as shown in the diagram below.

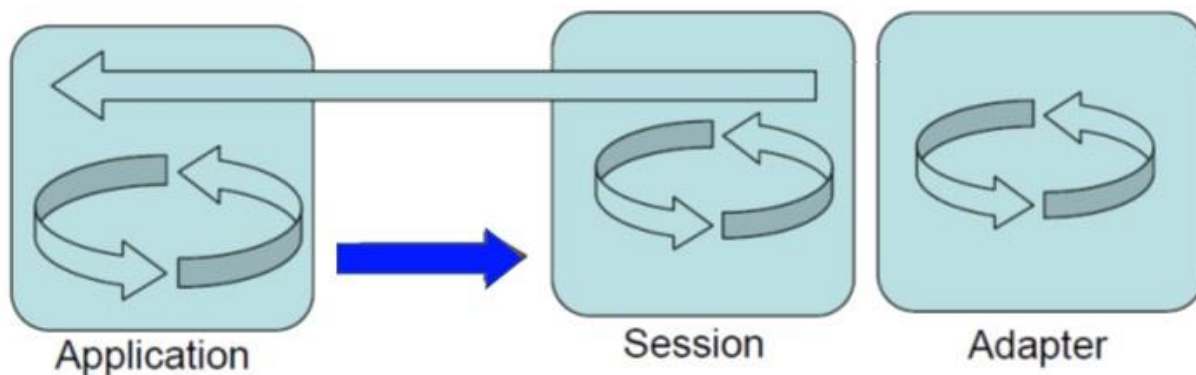


Figure 108: Dual-Threaded with No Event Queue (Callback Model)

There are a total of three threads in this configuration. The main application thread that is started by the OS 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 adapter to read and write messages to the connection.

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.

The model is enabled by specifying NULL as the `EventQueue` argument in `RegisterClient()` and by setting `threadModel=Dual` in the Session Layer configuration.

14.4.5 Multiple Event Queues (Client Model)

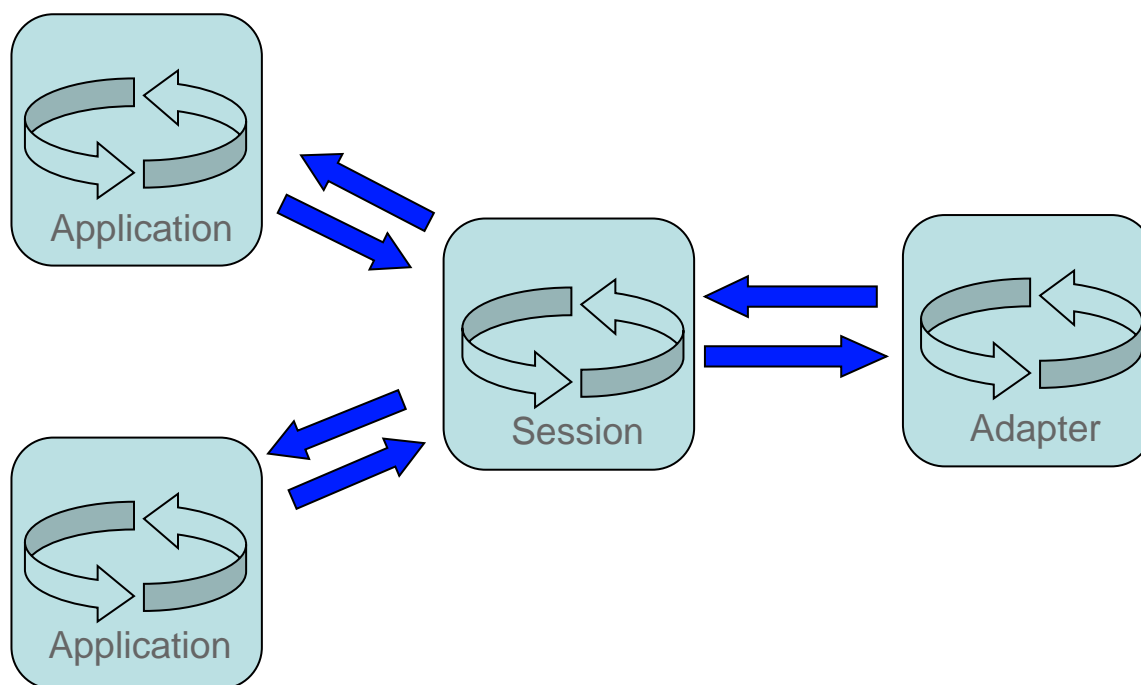


Figure 109: Multiple Event Queues (Client Model)

The above diagram shows a total of four threads: two application threads, a session thread, and an adapter 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.

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.

This model is enabled by each application thread specifying a different `EventQueue` argument in `RegisterClient()` and by setting `threadModel=Dual` in the Session Layer configuration. In this model, each application thread acquires (shares) the same session object.

14.4.6 Horizontal Scaling (Client Model)

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.

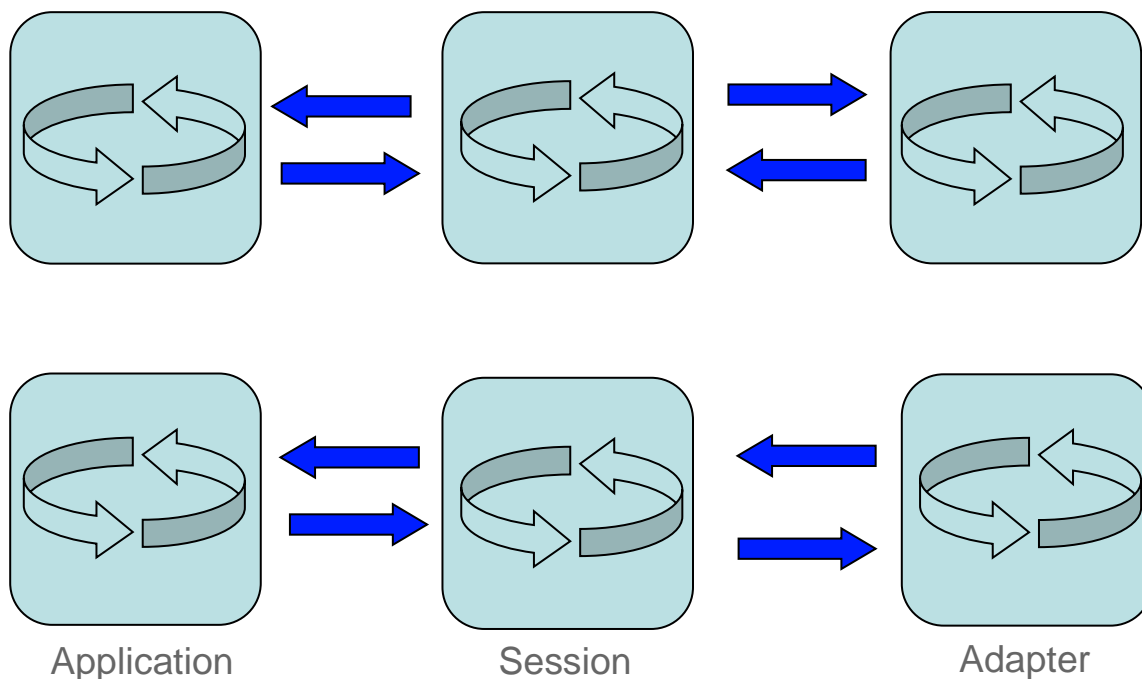


Figure 110: Horizontal Scaling (Client Model)

In the above diagram there are a total of six threads. There are two application threads, two session threads, and two adapter 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 adapters 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 adapter thread. However, in general, applications will find it more efficient to operate within a single process.

14.5 Configuring RFA Provider for Performance

14.5.1 Direct Write Model

The Direct Write thread model used for providers is shown in the diagram below. This model is similar to the full throughput consumer model in that inbound requests from the network are dispatched and processed in sequence by the adapter, Session Layer, and application threads using the event queue and the response queue. Unlike the full throughput model, this thread model has been optimized for outbound data. When the application submits a message to be sent to the network, it will be processed immediately by the Session Layer and adapter so it can be written to the network without a thread context switch. If the data cannot be flushed to the network immediately, the adapter thread will take over the responsibility of flushing the data so the application does not block while waiting on an I/O write.

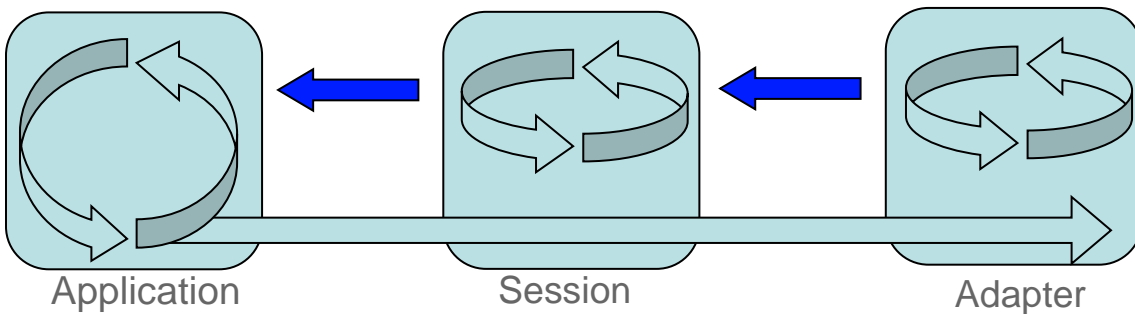


Figure 111: Direct Write Model

Since this model uses a response queue, the parameters **responseQueueBias**, **responseQueueMaxBatchSize**, and **responseQueueBatchInterval** described earlier can be used. Provider-specific parameters pertaining to RSSL provider connection performance are described below and can also be found in the *RFA Configuration Guide .NET Edition*:

NOTE: OMM Provider gives flexibility to application by supporting no event queue in the same way as the OMM Consumer. This can be specified on **OMMProvider.RegisterClient()** method. Benefits for this option are similar to the consumer using no event queue.

14.5.1.1 forceFlushOnWrite

Setting **forceFlushOnWrite** to **true** specifies that each outgoing message sent by the provider will be written to the connection immediately. Forcing a flush on write for every message will reduce latency, but it will also reduce maximum throughput. Setting this parameter to **false** will increase buffering, reduce the number of I/O writes done by RFA, and provide higher throughput.

14.5.1.2 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.

14.5.1.3 outboundMessagePacking

Setting the **outboundMessagePacking** parameter to **true** will enable message packing at the transport level. This will buffer messages until an internal threshold is reached, at which point the messages will be written to the transport as a single outbound packet. Setting this parameter to **false**, RFA will write individual packets. Its value is usually set to the opposite value of **forceFlushOnWrite** and **tcp_nodelay**.

14.6 Memory Management

RFA packages internally track several objects obtained by an application via the `Create()` or `Acquire()` methods. Although RFA internally tracks these objects, an application must explicitly indicate when it is no longer using these objects.

An application can explicitly control the sequence of releasing of RFA resource. Similarly, an application can call `Release()` for each call to `Acquire()` to avoid resource leaks. Whether a call to `Destroy()` or `Release()`, RFA may or may not immediately delete the actual object. Deletion of the actual object may be different for different types of interfaces. Although the application does not release RFA resource, the finalizer of each object will eventually release RFA resource for application.

In some cases, RFA provides the ability to destroy a `Container` instance without needing to destroy the contained instances. For example, an application may destroy an `EventQueueGroup` which implicitly destroys any contained `EventQueues`.

Once an application has called the `Destroy()` or `Release()` methods on a reference to a object, it should never use this reference again. If an application internally duplicates the reference returned by a `Create()` or an `Acquire()` method, it should take care not to `Destroy()` or `Release()` the object in one place of the application code while still using it elsewhere.

Typically, smaller sized interfaces expose constructors and destructors. This allows an application to create interfaces on the stack for ease of programming. If an application chooses to create instances of these interfaces on the heap, it must take care to free memory for these instances when no longer needed.

The application should not attempt to manage memory for objects provided in a Client callback method. Rather, RFA manages these objects (typically specific types of Events) provided in a Client callback method. Additionally, the application should not attempt to manage memory for any objects contained in objects provided in a Client callback method.

An application may create a copy of an Event by using the `Clone()` method while in the Client callback method. Unlike the Event provided in the Client callback method, the application is responsible for cleaning up a cloned Event via the Event's `Destroy()` method. A `Clone()` call makes a heap allocation and may affect performance.

14.7 Object Reuse

Because memory allocation and deletion impacts performance, applications should reuse objects whenever possible to improve performance. RFA makes copies of the information it needs when objects are passed in to the following functions:

- `RegisterClient()`
- `Submit()`
- `Bind()` in double-pass breadth encoding using a write iterator, `Complete()` for single-pass depth encoding

Applications can reuse these objects after these functions return. It is especially useful to reuse `Msg`, `Data`, `InterestSpec`, `Cmd`, and entry objects. If the class has a `Clear()` method, it should be called before reusing the object.

However, object reuse has two limitations:

1. When a nested data object is encoded in an entry with single-pass depth encoding (e.g., `SetData()` and `Bind()` are called before the nested data object is encoded), the nested data object should not be reused as a top-level data object or as a pre-encoded object (i.e., for double-pass breadth encoding).
2. When a data object is encoded with double-pass breadth encoding, it should not be reused in a single-pass depth encoding. For example, if a `FieldList` is pre-encoded for the summary data in a `Map`, it should not be reused in a single-pass depth encoding of the `Map`'s entries.

NOTE: Though the use of `Clear()` to reuse any Entry, Entry Definition, `Array`, `ElementList` or `FieldList` is optional (and hence skipped in a high-performance application), the use of `Clear()` is mandatory to reuse `ElementListDef`, `FieldListDef`, `FilterList`, `Map`, `Series`, or `Vector`.

14.8 OMM Message Pool

RFA maintains an internal pool of preallocated messages and reuses them as needed. For example, when an application reaches the end of a `processEvent()` callback, the event is released back to the pool for reuse.

Because RFA needs to store `OMMSolicitedItemEventMsg` and `OMMItemEventMsg` instances until after they are processed by the application, the `RSSL_Cons_Adapter` and `RSSL_Prov_Adapter` (RSSL-type adapters) provide the capability to control message pooling via `OMMItemEventPool` and `OMMSolicitedItemEventPool`. Applications can use configuration parameters to set the initial size of the pool, change how the pool size is increased, and set maximum pool sizes.

In addition to the `OMMItemEventPool` (used by both `RSSL_Cons_Adapter`) and `OMMSolicitedItemEventPool` (used by `RSSL_Prov_Adapter`), RFA instantiates other message pools used by applications:

- `ReqMsgPool` used by the providing application when processing Request Message
- `RespMsgPool` used by the consuming application when processing Response Messages (refreshes, statuses, and updates)
- `GenericMsgPool` used by the consuming and providing application when processing Generic Message
- `PostMsgPool` used by the providing application when processing Post Message
- `AckMsgPool` used by the consuming application when processing Ack Message

These pools work the same way as `OMMItemEventPool` and `OMMSolicitedItemEventPool`; but because applications typically process one message at a time, and RFA does not need to store their instances, their default pool sizes are low.

The **InitialSize** parameter sets the initial size of an appropriate message pool. The **increment** parameter dictates how many pre-allocated messages will be added to the message pool at a time, if needed. The **maximumSize** parameter sets a desired maximum size of an appropriate message pool. The **maximumSize** parameter does not limit the growth of the message pool, i.e, the message pool will always grow as much as it needs to accumulate all the messages it needs to store. However, the message pool will “shrink” to this size when conditions allow. If the **maximumSize** parameter is set to zero, then the appropriate message pool will never shrink. The message pool size will reflect the maximum number of messages that have been needed at one time.

Setting the **maximumSize** parameter value presents a tradeoff to applications. Consider the following information when configuring and testing an application:

- If **maximumSize** is set to **0**, then the message pool may grow very large and could use a large amount of memory for the life of the application. It will hold onto this memory even if it is never needed again. However, the pooled messages will be immediately available to the RFA, which in turn will provide quick responses without the overhead of memory allocation.
- If **maximumSize** is set to something greater than **0**, then the appropriate message pool will release memory in an effort to use no more messages than the specified value. However, every time there is a need to store more messages than the maximum, RFA will allocate memory from the operating system, which in turn will provide much slower responses.
- The **maximumSize** parameter specifies the size of the message pool in terms of number of messages and not in terms of used/allocated memory. The actual size of the allocated memory will vary.
- The number of messages stored by RFA depends on several factors such as update rate and its fluctuation, average depth of internal and application event queues, and the application’s speed. For example, the slower an application is, the more messages need to be stored by RFA.

For more details on the parameters names, defaults, and allowable values, refer to the *RFA Configuration Guide .NET Edition*.

14.9 OMM Consumer Request Throttling

If an OMM client application requests a large number of items at the same time, requests could overflow the outgoing channel. This could cause the channel to disconnect. At the same time, the requested item images could begin to arrive. If the consuming application cannot process the images fast enough, the network channel will overflow, causing the channel to be disconnected. When the consumer tries to recover, it will again request all of the items, thereby repeating the problem. This situation is often called thrashing. RFA uses request throttling to avoid thrashing and to ensure that the consuming application behaves in a reasonable manner.

RFA has two request throttle queue implementations. The first, a timer throttle, sends requests to the infrastructure at a fixed rate. The second, a count throttle, limits the maximum number of pending requests that have not yet received an image.

The throttle queues for RFA are configurable on a per connection basis. Most consuming applications can use the default values. 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.

By default, the RSSL connection uses a maximum pending count-based queue. The throttle queue implementation can be changed, configured, or even disabled. *For more details about these configuration parameters—**throttleEnable**, **throttleTimerInterval**, **throttleRequestPerInterval**, **throttleMaxCount**, and **throttleBatchCount**—refer to the *RFA Configuration Guide .NET Edition*.*

14.10 Using Service Groups: Service Renaming / Aliasing

The following is an example of a consuming application utilizing the service renaming capability of RFA to connect to two different service providers / feeds that use the same service name. The consuming application uses a single session – Session1 – and two connections – Connection1, and Connection2. Both connections provide the same service (RDF).

```
Sessions\Session1\ServiceGroupList = "SG"
ServiceGroups\SG\ServiceList = "RDF-0,RDF-1"
Connections\Connection1\ServiceList = "RDF-0"
Connections\Connection2\ServiceList = "RDF-1"
Services\RDF-0\FeeName = "RDF"
Services\RDF-1\FeeName = "RDF"
```

Example 223: Configuration for rename service

The application requests items from the Service Group SG. RFA translates / renames RDF-0 and RDF-1 service names to RDF to be used on Connection1 and Connection2 respectively.

Chapter 15 Deprecated Functionality

This section describes functionality that has been deprecated from RFA. Deprecated RFA classes are supported only for purposes of backward compatibility with existing applications. Deprecated classes should no longer be used for new applications. Refinitiv continues to support deprecated classes for use in existing applications through the lifetime of the particular release in which it was declared as deprecated. After that release, Refinitiv no longer supports the deprecated classes for any application and they can be removed from the RFA package. Deprecation can occur at the beginning of any point release (e.g., 8.0.0 RRG).

Please see the top level readme distributed with the RFA package for possible alternatives.

15.1 OMMPerfMode Session Configuration

The name of the **OMMPerfMode** configuration parameter has been changed to **threadModel** to better describe the impact that the parameter has on threading, not necessarily performance, within RFA. Its behavior within RFA is unchanged. **Latency** mode is now **Single**-threaded mode, while **Throughput** mode is now **Dual**-threaded mode.

For more details, refer to both *RFA Configuration Guide .NET Edition* and *RFA Migration Guide .NET Edition*

Appendix A Deployment

The RFA .NET edition software package is in the zip format. After the software package is properly downloaded, extract the software package to your specific location. It will take a few minutes for all files to be copied from the release package to the specified load directory. Upon completion, the parent directory will contain a directory named with the load name (*load_name_dir*) of the software (e.g., ***rfanet8.0.0.X.win.rrg*** where X is the load number). The use of a load name is designed to allow for easy identification and selection between current and future releases of the product.

The load-named directory contains the following subdirectories:

- <load_name_dir>/Docs
- <load_name_dir>/etc
- <load_name_dir>/Examples
- <load_name_dir>/Libs
- <load_name_dir>/MessageFiles

RFA.NET assemblies and native message file DLLs can be deployed by setting the PATH environment or using Global Assembly Cache (GAC).

To run the application, users must ensure that the following dll files exist in the current .exe file directory ("rfanet<major>.<minor>.<maintainance>.<load number>.win.rrg \Examples\YourExample\Release_WIN_xx_VSyy\") or /windows/system32: RFA8_NETyy.dll, RFA8_MsgFileyy.dll, DACS7_lock_NETyy.dll (if used) and AnsiPage_NETyy.dll (if used). If these dll files do not exist in these directories, users must specific the PATH environment variable to point to these files.

NOTE: xx is 32 or 64 bit platform and yy is version of Visual Studio (90 or 100 or 110).

To deploy the application, user must add RFA.NET assembly manually, or add RFA.NET assembly from the Global Assembly Cache. In order to use RFA.NET assembly from the Global Assembly Cache, user must install RFA.NET assembly as the following step.

1. Open Visual Studio Command Prompt, and install RFA.NET assembly with the gacutil command.

```
gacutil /i <load_name_dir>/Libs/WIN_32_VS90/RFA8_NET90.dll
```

or

```
gacutil /i <load_name_dir>/Libs/WIN_32_VS100/RFA8_NET100.dll
```

or

```
gacutil /i <load_name_dir>/Libs/WIN_32_VS110/RFA8_NET110.dll
```

2. To display RFA.NET assembly in Add Reference dialog box, user must add a registry key, such as the following, which points to the location of the assembly.

```
[HKEY_CURRENT_USER\SOFTWARE\Microsoft\ .NETFramework\AssemblyFolders\RFA8_NET90]@="<load_name_dir>\Libs\WIN_32_VS90"
```

or

```
[HKEY_CURRENT_USER\SOFTWARE\Microsoft\ .NETFramework\AssemblyFolders\RFA8_NET100]@="<load_name_dir>\Libs\WIN_32_VS100"
```

or

```
[HKEY_CURRENT_USER\SOFTWARE\Microsoft\ .NETFramework\AssemblyFolders\RFA8_NET110]@="<load_name_dir>\Libs\WIN_32_VS110"
```

3. To uninstall RFA .NET edition assembly, user must open Visual Studio Command Prompt and uninstall the assembly with the gacutil command as below.

```
gacutil /u <fully qualified assembly name>
```


Appendix B Implementation

B.1 Using IDisposable Interface

RFA .NET classes have the finalizer method so that the garbage collector automatically calls this method to release unmanage resources when that object is no longer used. However, it is not possible to predict when garbage collection will occur. By using this interface, clients can perform deterministic clean up unmanage resources of RFA .NET classes.

There are some scenarios that would result in expected error as below.

- Stack semantic in C++/CLI automatically delete unmanage resources when they are out of scope. If they are still reference by other object, unexpected error would occur.

B.2 Data Scope Handling

The object reference from event callback can be changed or deleted when the object reference are out of scope. RFA interface returns object reference to the application. The value in the object can be changed and deleted at any time. Therefore this object does not support stable value and reliable comparison. The application recommended to create a copy of the object by using `Clone()` method. After the copy object is created, the application can use the copy object with the original object's value safely.

```
DataBuffer^ dataBuffer = gcnew DataBuffer();
{
    RFA_String test("TEST");

    dataBuffer->SetFromString(%test, DataBuffer::DataBufferEnum::StringAscii);
}

// The unmanaged resource of RFA_String was released from the destructor of the test variable.
RFA_String^ temp = dataBuffer->GetAsString();
```

Example 224: Unexpected error in C++/CLI

- Clients might create multiple objects and set references to each other. Then they call `Dispose()` method of a object which is still needed by the others.
- Clients might create a object and call `Dispose()` method to release unmanage resource but they accidentally use it without setting null.

B.3 IEnumerable and IEnumerator

RFA .NET implements `IEnumerable` and `IEnumerator` interfaces in Config and Data package. `IEnumerable` interface provides `GetEnumerator()` method and return the `IEnumerator` interface. This `IEnumerator` interface allow the application to iterate through `ConfigTree` and all data containers by using foreach loop or the methods of `IEnumerator`.

For more information, refer to Sections 8.2.4.3 and 6.2.2.1.

B.4 Setting Reference Type

RFA .NET provides the interfaces for setting the reference of objects among each other. For instance, clients can create an [RFA_String](#) object and set it to an [AttribInfo](#) object. In this case, the [AttribInfo](#) object depends on the [RFA_String](#) object. Thus, if the value of the [RFA_String](#) object is changed, the [RFA_String](#) object must be set to the [AttribInfo](#) object again to ensure that the changed value is consistent between the two objects.

```
attribInfo.Clear();
RFA_String myString = new RFA_String("DIRECT_FEED");
attribInfo.ServiceName = myString;
myString.Set("IDN_RDF");
attribInfo.ServiceName = myString;
```

Example 225: Setting Reference Type

If the value of [RFA_String](#) object is changed but it has not set to the [AttribInfo](#) object, the service name property of [AttribInfo](#) may not be synchronized with the [RFA_String](#) object which leads to unexpected result.

```
attribInfo.Clear();

RFA_String myString = new RFA_String("DIRECT_FEED");
attribInfo.ServiceName = myString;
myString.Set("IDN_RDF"); // The value of attribInfo.ServiceName is unexpected result.
```

Example 226: Setting Reference Type which Leads Unexpected Result

Appendix C Application Design

RFA is designed to work in many different deployments. It provides thread safety and several threading models. Data used with RFA can be encoded and decoded with many different libraries. The flexibility provided by all of these features results in an application design that may, at first, appear unusual. This section describes how RFA-based applications fit into the collection of RFA libraries and components.

C.1 Typical Consumer Application

RFA is designed to connect using multiple protocols to different types of infrastructures, either independently or simultaneously. Some of the internal libraries used to implement these protocols have specific and different usage requirements for thread-safety. RFA hides these threading details in an internal Adapter. The Adapter is also responsible for some of the event normalization. RFA can have one Adapter for each connection type. However within the Adapter, RFA can support multiple connections of that Adapter's type. The session is responsible for bringing together one or more Adapters and presenting a unified interface. An application uses Session Layer interfaces to create event sources for subscription, contribution, and other tasks. The Application will receive events through an EventQueue. The Application can choose when events are dispatched from the EventQueue, giving the applications control of the threading behavior. For some Event Sources, the thread model can be changed. The EventQueue can be optionally skipped, resulting in a callback from the Session Layer's thread directly into the Application. This reduces threading flexibility but also improves latency. In either thread model, the Application is required to implement a simple Client interface to receive the events.

From the Application's Client implementation, it can decide what to do with the event and any associated data. Typically the Application passes the data to a Decoder library, such as the OMM decoder. While this puts the Application in the middle of the event processing flow, it provides a performance advantage. The Application can choose how to use the Decoder in the most efficient manner to satisfy its needs.

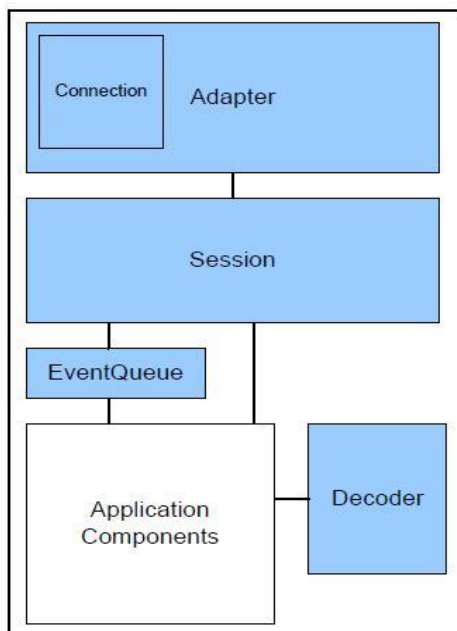


Figure 112: Typical Consumer Application Design

C.2 Complex Consumer Application

RFA's design allows Applications to use multiple Adapters, Sessions, Connections, Decoders and EventQueues at the same time. The following diagram shows some of this flexibility.

This application is using two sessions. The first session uses a single Adapter with two Connections. The second session uses two adapters. RFA has specific rules about which adapters can be merged into a single session. This application is using three EventQueues; this means it probably has three dispatch threads. When an application registers interest in an Event Stream, it can specify the EventQueue for that stream. Temporal order is not guaranteed between the EventQueues, but that may not be an issue for unrelated data. One of the EventQueues is being used for Event Streams from both Sessions.

This application is also using two decoders for different types of data. For example one decoder could be for OMM data and the other could be for ANSI Page data which retrieves from decoding OMM data.

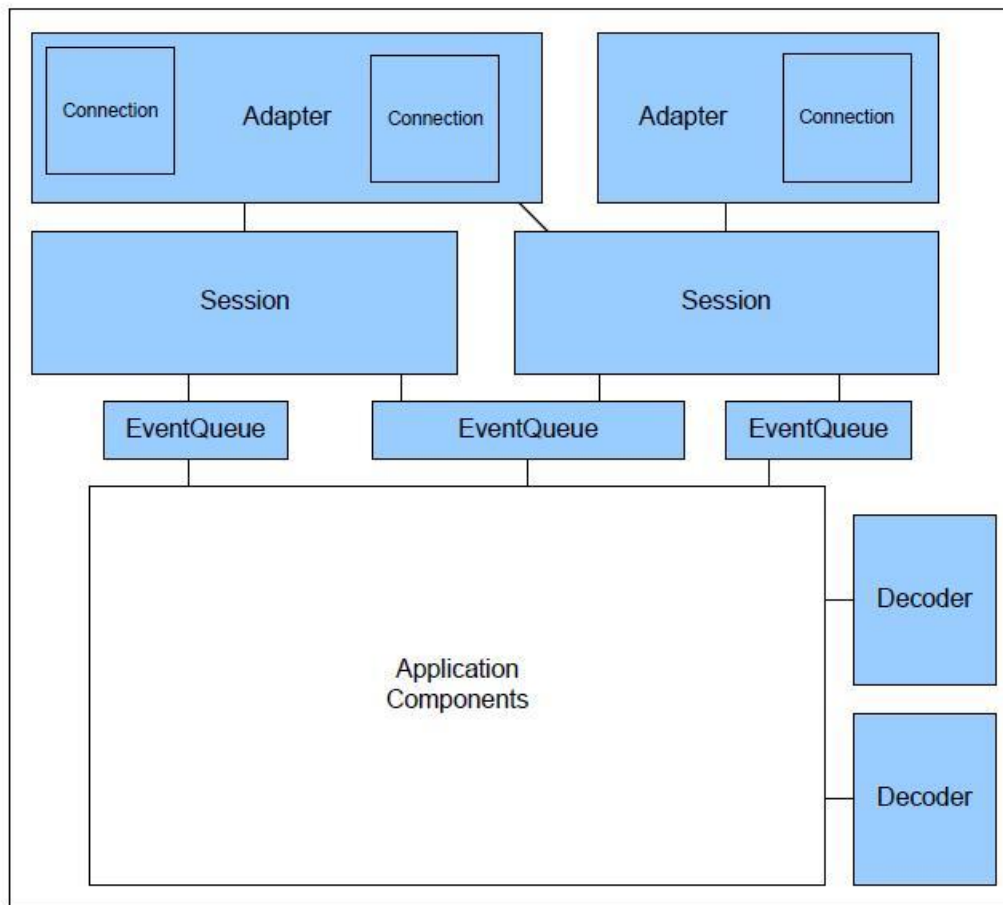


Figure 113: Complex Consumer Application Design

C.3 Typical Provider Application

Provider applications are similar to consumer applications except for the direction flow.

Like consumers, the Session is responsible for bringing together one or more Adapters and presenting a unified interface. However, most provider applications will only use one Adapter at a time. The Connection and the Adapter shown in the diagram may be a client connection to a server that republishes the data. For some Adapter types, it is actually a server listening port. When the Connection is a server, the Application will receive events about each inbound connection (i.e. client session) that connects to RFA.

An Application uses Session Layer interfaces to create Event Sources for publication. The Application can still receive events through an EventQueue. The Application chooses when events are dispatched from the EventQueue, giving the Application control of the threading behavior. The inbound EventQueue cannot be skipped. Instead, for some Event Sources any outbound thread context switches are optionally skipped. This improves the latency of outbound messages, while the inbound EventQueue maintains thread-safety and prevents reentrant calls from RFA into the Application from an Application thread. The Application is required to implement a simple Client interface to receive the events.

From the Application's Client implementation, it can decide what to do with the event and any associated data. Most events will not require much, if any, decoding. Instead the provider uses an Encoder to create new data to be published. The Application can choose how to use the Encoder in the most efficient manner to satisfy its needs. Depending on the data format, the Application may even be able to choose between different Encoders.

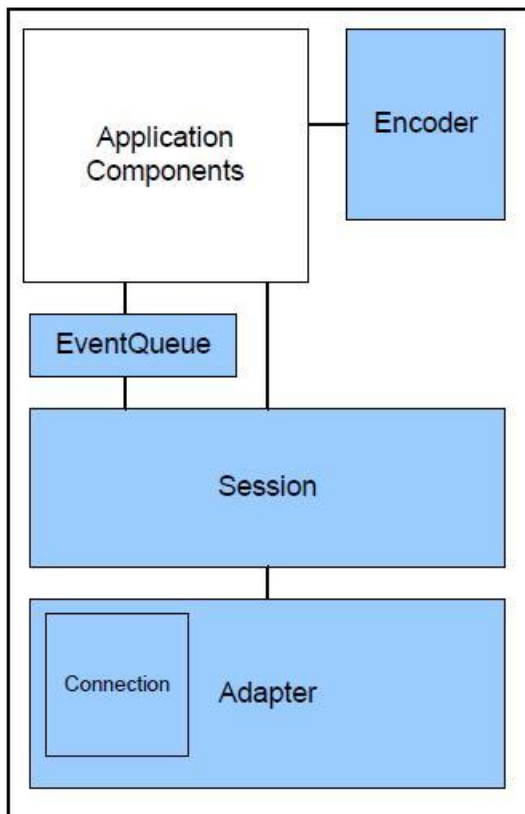


Figure 114: Typical Provider Application Design

C.4 Hybrid Application

RFA supports creating hybrid applications that both consume and publish data. Hybrid can receive a Request Message from a consumer and forward the same request message to another provider. Conversely, a hybrid application can receive a Response Message from a provider and forward the same response message to the originating consumer. While the data is passing through the hybrid, the contents of the messages can be decoded, changed, and re-encoded.

The diagram shows an hybrid application that has only one Event Source on the consumer Session no matter how many inbound client session are received from the provider Session. It forwards a stream from the consumer Session to provider Session for fanning the stream out to each interested client session.

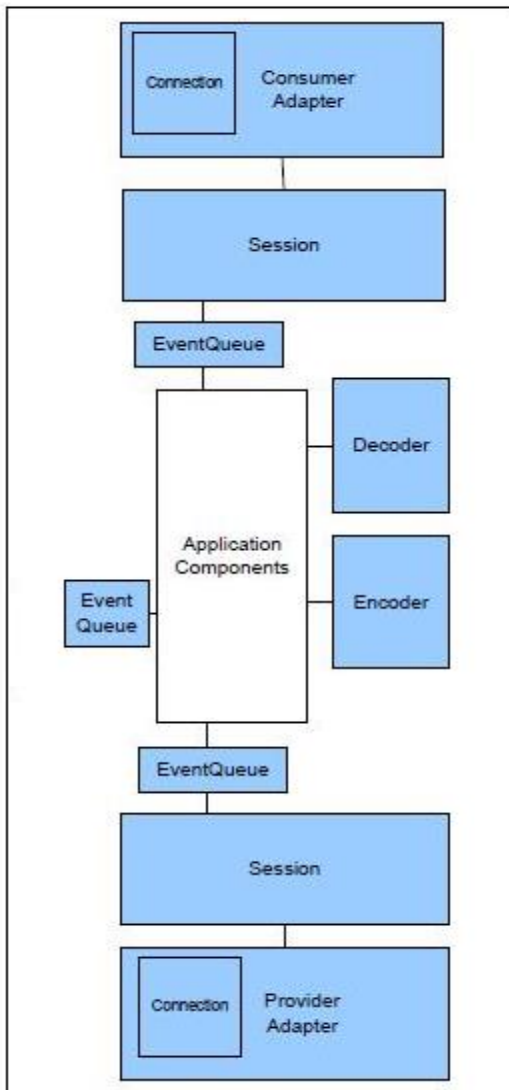


Figure 115: Hybrid Application Design

© Copyright 2015 - 2017, 2020, 2022 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.