Refinitiv Real-Time Distribution System

OPEN MESSAGE MODEL WHITE PAPER

Document Version: 3.5.1
Date of issue: August 2020
Document ID: OMM351WP.200

REFINITIV™

Contents

1 About this Document

1.1 Intended Audience

This white paper is directed towards market data systems managers, administrators, users, content modellers, and developers.

1.2 In this Guide

This white paper focuses on the ***Open Message Model*** and how it can be used, both by Refinitiv and other parties, to model many different types of content and/or workflows. The Open Message Model is a key component of the enterprise-wide data model architecture. This architecture defines a layered model which is used to describe/distribute all content and workflows. The data model architecture and Open Message Model have been specifically designed to overcome existing limitations and to provide a more flexible and efficient basis for exchanging financial data in the future.

Current interfaces and systems provided by Refinitiv support a range of different data models, most of which have been created around the popular Marketfeed record format. Nearly all applications understand the Marketfeed record format; unfortunately, it is not flexible enough to describe complex data efficiently. This led developers to force some data models into the simple Marketfeed record format, resulting in unwanted complexity and possible inefficiencies. This has also created a generation of applications that use proprietary data formats whose data models are not compatible with other applications.

This version of the white paper has been updated in relation to the Refinitiv Real-Time Distribution System 3.X infrastructure and Refinitiv Real-Time API 3.X releases.

1.3 Scope

This paper discusses the data model architecture in its entirety.

- Chapter 2 explains a high-level overview of the Open Message Model, the business needs that drove its development, and the benefits users can realize through its implementation.
- Chapter 3 discusses key terminology and concepts needed to understand Open Message Model implementation within the Refinitiv Real-Time Distribution System.
- Chapter 4 gives an overview of wire formats; the present range of formats will be replaced by one single format for all messaging layers.
- Chapter 5 looks in detail at the Open Message Model; including the message types and abstract data types that are the building blocks for defining workflows.
- Chapter 6 describes the Domain Message Model; where data types from the Open Message Model are used to model real content (such as market-by-price). Some pre-defined domain models are also examined in detail to show how the Open Message Model is used in practice.

The main purpose of this document is to define Open Message Model architecture. Robust Foundation API and Refinitiv Real-Time API representation of this architecture might use different terminology or expose slightly different attributes (e.g., use “service name” instead of “service identifier”).

1.4 Glossary of Terms

ACRONYM / TERM	MEANING
ADH	Refinitiv Real-Time Advanced Data Hub is the horizontally scalable service component within the Refinitiv Real-Time Distribution System providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	Refinitiv Real-Time Advanced Distribution Server is the horizontally scalable distribution component within the Refinitiv Real-Time Distribution System providing highly available services for tailored streaming and snapshot data, publication and contribution messaging with optional persistence, conflation and delay capabilities.
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
Backbone	The combination of Refinitiv Real-Time Advanced Data Hub and Refinitiv Real-Time Advanced Distribution Server communicating via multicast to deliver the services offered by providers to consumers. It represents the actual, real-time, streaming service-oriented-architecture (SOA) implementation of Refinitiv Real-Time Distribution System.
Consumer	Applications that utilize the services offered by providers. One of the primary roles of Refinitiv Real-Time Distribution System applications.
Decoding	The act of parsing and converting content from a wire format, or external representation, into a machine readable form (e.g., Refinitiv Wire Format parsing, Marketfeed parsing, XML parsing).
Domain Model	A Domain Model combines primitive data structures and relevant transport details to create data types that match market needs.
Enterprise Message API	Enterprise Message API is part of the Refinitiv Real-Time SDK.
Enterprise Transport API	Enterprise Transport API is a high performance, low latency, foundation of the Refinitiv Real-Time SDK. It consists of transport, buffer management, compression, fragmentation and packing over each transport and encoders and decoders that implement the Open Message Model. Applications written to this layer achieve the highest throughput, lowest latency, low memory utilization, and low CPU utilization using a binary Refinitiv Wire Format when publishing or consuming content to/from Refinitiv Real-Time Distribution Systems.
Event Stream	The result of a request/response with interest interaction. When an application requests streaming news headlines, an event stream is created that provides asynchronous headlines (updates) and state information.
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
Market-by-Order	Instrument market information sorted by order (i.e. full depth order book: level-2 content).
Market-by-Price	Top of book instrument market information sorted by best price (i.e. market depth: level-2 content).
Market-Price	Trades, quotes and inside top of book quotes (i.e. level-1 content).
OMM	Open Message Model
Provider	A provider is an application that implements, or makes available, a service to consumers; one of the primary roles of Refinitiv Real-Time Distribution System applications. Providers can be interactive or non-interactive.
Publisher	Applications that make data, or content, available. Publishers can be implemented as providers that offer services or ***consumers*** that push information into services.
QoS	Quality of Service

Table 1: Acronyms and Abbreviations

ACRONYM / TERM	MEANING
Refinitiv Real-Time Distribution System	Refinitiv Real-Time Distribution System is Refinitiv's financial market data distribution platform. It consists of the Refinitiv Real-Time Advanced Distribution Server and Refinitiv Real-Time Advanced Data Hub. Applications written to the Refinitiv Real-Time SDK can connect to this distribution system.
Refinitiv Real-Time Distribution System Infrastructure	The primary components that make up the Refinitiv Real-Time Distribution System infrastructure (i.e., Refinitiv Real-Time Advanced Distribution Server, Refinitiv Real-Time Advanced Data Hub).
Reactor	The Reactor is a low-level, open-source, easy-to-use layer above the Enterprise Transport API. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users.
RIC	Reuters Instrument Code.
RFA	Robust Foundation API: a thread-aware messaging API that applications can use to provide and consume Open Message Model.
RMTES	A multi-lingual text encoding standard
RSSL	Refinitiv Source Sink Library
RTSDK	The Refinitiv Real-Time SDK which includes APIs (i.e., Enterprise Message API and Enterprise Transport API) supported on Refinitiv Real-Time.
RTT	Round Trip Time, this definition is used for round trip latency monitoring feature.
Refinitiv Wire Format	A Refinitiv proprietary format for data representation
SSL	Sink Source Library
RDF-D	Refinitiv Data Feed Direct
Wire Format	An encoded, hardware neutral, external representation of a data model that provides for the transmission of the model between multiple components (usually over a LAN or WAN). Wire formats may be optimized using many different networking tricks (e.g. nibbles, bitmaps, integer chains, etc.).

Table 1: Acronyms and Abbreviations

1.5 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.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to Refinitiv by clicking **Send File** in the **File** menu. Use the apidocumentation@refinitiv.com address.

2 Introduction

2.1 Executive Summary

Refinitiv has created an architecture for modelling workflows (and data content) within the Refinitiv Real-Time Distribution System. This architecture enables applications to send and receive data more efficiently, while providing developers the flexibility to access new data types and model additional data types to meet changing market requirements.

Care has been taken to ensure continued support for the thousands of previously-deployed custom applications and third-party products. The current architecture maintains all key interfaces necessary to run legacy applications while simultaneously taking advantage of new data models and workflows.

2.1.1 Driving Change

Recent years have seen a strong increase in the demand for market data. Trading applications, particularly the fast-increasing number of algorithmic-trading applications, are driving this trend. These applications need to scan and analyze huge volumes of data with ultra-low latencies to spot trading opportunities, determine risk, and trigger orders or execution.

In addition to the sheer volume of data, these applications require programmatic access to an even broader array of capabilities than ever before; such as level 2 data (e.g. market depth, order books), deep tick history, news, portfolios, and transactional workflows. As this trend for new content to feed applications accelerates, in addition to access to many different workflows, there will be a constant requirement for programmatic access for these new services.

Existing data structures have served markets well for more than a decade now, but it has become clear that simple data structures are not going to be able to supply the levels of flexibility and performance (regarding both processing speed and throughput) that will become the norm in the future. Consequently, Refinitiv has re-architected its data model from top to bottom, to create structures that will meet these current needs, and anticipate future needs, while providing continued support for existing data structures.

To address these requirements, Refinitiv developed the Open Message Model data model architecture. For an overview of this model's 'layers', refer to the Open Message and Open Domain Models Summary in Section 2.2.

2.1.2 Unleashing Innovation

The ***Open Message Model*** defines a specification that allows for truly open and extensible data models. Developers can extend existing data models to suit their needs or create completely new ones. This degree of flexibility is a key requirement because applications need access to new capabilities; including increasing depth of data and access to transactional systems.

Additionally, the definition of data structures is no longer dependent on infrastructure or API upgrades. With legacy data formats, you have to wait for a new software release or an updated management policy vis-a-vis configuration of the system before you can exploit any changes to the basic data structures. With the Open Message Model you can rapidly design a new message model, code it into your application using a Refinitiv Real-Time API (e.g., the Enterprise Message API), and deploy it in your environments using the Refinitiv Real-Time Distribution System. With modern development and operational procedures, development changes to an application can occur on timescales measured in minutes instead of weeks. The only limitation is the imagination of the application developer. As a result, business requirements can be delivered in progressively faster time to market adjustments to keep your organization focused on delivering high returns on investments.

It has always been central to the success of the Refinitiv Real-Time Distribution System that development partners and individual institutions were able to leverage the system in different ways to deliver extra dimensions of functionality and value. The flexibility and underlying performance of the new architecture opens up new opportunities for creating innovative financial applications. For example, in algorithmic trading, the Open Message Model can represent new information, such as full order books, complex yield curves, machine readable news, tick histories, corporate actions, and reference data. The Open Message Model also supports new functionality for Refinitiv Real-Time Distribution System applications such as enabling transactions and settlements, leveraging standards like FIX, and modelling SWIFT in the Open Message Model. Optimized and effective messaging is the foundation of Refinitiv Real-Time Distribution System.

Extending the concept of openness, the Refinitiv Developer Community portal shares innovations, offering collaborative forums and the opportunity to interact directly with Refinitiv developers and other members of the development community. Customer-defined data models will be available for access across organizations. Via the portal, common interest groups supporting these data models can collaboratively expose new use cases, above and beyond what traditional systems process today (e.g., new transaction types or trade blotters).

2.1.3 Built-in Benefits

Market data managers will appreciate the efficiencies that are inherent in Open Message Model structures, which result in lower costs, lower risk, and higher performance:

- The underlying Refinitiv Real-Time Distribution System architecture can be structured to deliver both high throughput and low latency. Using the Open Message Model, sub-millisecond latency can be achieved at higher update rates than ever before.
- The binary efficiency of the Refinitiv Wire Format means that message sizes are reduced. Encoded content is half the size of the same data in Marketfeed format, and about one-sixth of the size that of TibMsg.
- Using the Open Message Model, you can achieve higher throughput rates using Refinitiv Real-Time Distribution System components when compared to older components using Marketfeed (i.e., Refinitiv Real-Time Distribution System). The Refinitiv Real-Time Distribution System and Open Message Model can deliver anywhere from 45% to 100% more throughput.
- The binary format of the Refinitiv Wire Format ensures the faster processing speeds required by algorithmic applications. Using binary data, there is no need for costly conversion between strings and some other data type strings.
- This is an architecture that enables more effective sharing of data between the front, middle and back offices; meaning tighter infrastructure control, a wider audience, and greater breadth and depth of data access for applications.

2.1.4 Upgrade with Ease

Refinitiv Real-Time Distribution System is designed to facilitate easy upgrades, allowing you to meet the needs of new programmatic applications for more content by simply extending your Refinitiv Real-Time Distribution System. Components work together seamlessly across point releases to facilitate operational upgrades as/when these are needed. Thus new applications can access all of your valuable internally-published data. There is no need for wholesale system-changes or upgrades.

Applications using current APIs and data structures can connect to the upgraded infrastructure without any changes and will continue receiving the information they receive today. Refinitiv Real-Time Distribution System components provide automatic conversion of Open Message Model field lists (used in the market-price/level-1 model) to formats required by existing APIs like SSL, and System Foundation Classes.

2.2 Open Message and Open Domain Models Summary

In the Refinitiv Real-Time Distribution System, content is viewed in a more layered/structured manner as depicted in Figure 1. This structure enables each layer to build on the facilities provided by those beneath and offers a vocabulary for discussing the capabilities. The Refinitiv Real-Time Distribution System and Refinitiv Real-Time Distribution System strictly follow this data model architecture.

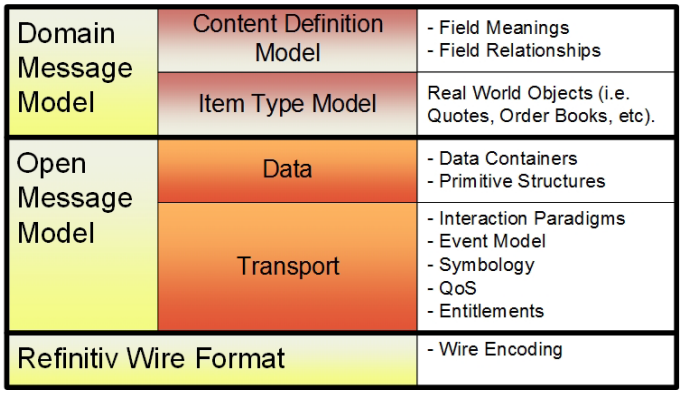

Figure 1. Data Model Architecture

The three key layers are shown in yellow. From the bottom upward, these are:

- The foundation layer of the model is the ***Refinitiv Wire Format***. This is the binary format used to distribute all data between Refinitiv Real-Time Distribution System and Refinitiv Real-Time Distribution System's core infrastructure (i.e., Refinitiv-Real-Time, Refinitiv Real-Time Distribution System, EED, and Refinitiv Data Feed Direct). In the past, there have been a number of different formats. Now all of these will be replaced by one single format that is more efficient in the way it handles data. This is a key quality, leading directly to shorter message lengths, higher throughput and lower latency.
- The middle layer is the ***Open Message Model***, which is implemented in all messaging APIs and core infrastructure components. Open Message Model contains two important components: (1) a transport protocol (i.e., Refinitiv Source Sink Library) which includes request types (such as those for handling generic event stream concepts), and (2) a set of primitive data structures (or abstractions) that are the building blocks for more complex types of data and workflows. The flexibility provided by the Open Message Model means that new workflows can be modeled in the future without any need for new versions of the messaging APIs (i.e., Enterprise Message API) and/or infrastructure components (i.e., Refinitiv Real-Time Distribution System).
- The top layer contains the ***Domain Message Model***. This is where primitive data structures are combined with relevant transport details to create data types that match market needs. Refinitiv has already created standard message models to meet existing needs; such as login, dictionary, market-price, symbol list, and market-by-order.

  Domain model types have been created by composition from basic Open Message Model message structures. Domain Message Model types are used in conjunction with one another, just like variable data types are in a programming language. Some Domain Message Model types include:
  - ***Market Price*** which combines Open Message Model constructs to represent Best Bid Offer content.
  - ***Market by Order*** which combines Open Message Model constructs to represent full order books.
  - ***Yield Curve*** which combines Open Message Model constructs to represent curve calculation details along with yield curve output points.

This data model architecture will be used to define how all Refinitiv content is described, structured, accessed and produced through client facing interfaces. Applications supporting these standards will inter-operate independent of the vendor/exchange providing the data/ workflow. Any interested parties can define their own domain models for item types, and these can be used without software enhancements to Refinitiv Real-Time Distribution System infrastructure or the Refinitiv Real-Time APIs.

Domain Message Models use the capabilities provided by the Open Message Model to define real business objects (e.g. market-price, news headlines) that are familiar to the industry. Domain model concepts are not understood by Refinitiv Real-Time Distribution System, thus allowing for new ones to be created without costly software upgrades. There are a few exceptions within the administrative domain (e.g. login, directory) where Refinitiv Real-Time Distribution System components have to perform some internal processing.

Developers define domain models through comprehensive documentation and examples. Message semantics and data representation (e.g. nested data formats) will be fully documented to provide maximum interoperability. Consumer and provider applications must conform to these definitions in order to properly work together. Domain models can be defined by Refinitiv or any interested party in order to satisfy particular business needs. Where messaging standards already exist (e.g. FIX), Refinitiv will use these standards modelled on top of the Open Message Model to define the domain models.

The Domain Message Model is divided into two layers as shown in light red in Figure 1.

- The item type model makes up the first layer in the domain model. It defines the actual object types, their corresponding transport behavior and data representation (i.e. data formats) using Open Message Model abstractions/concepts. Key attributes (see Section 5.2.1.2) are chosen and given meaning within the domain. Utilized messages are defined and given full semantic meaning (request/ response and any possible event streams). The use of any quality-of-service parameters (see Section 5.2.1.4) is defined and given concrete meaning. Any attributes that make up the extended header will also be defined.
- The content definition model builds upon the item type model in order to complete the model. This important and often not completely specified layer defines any field meanings and relationships on top of the item type models. A single item type model can have multiple content definition models (e.g. different field identifiers or symbology). Content definition models can include data dictionaries, enumerations information and any required/optional field definitions and relationships. Some item type models (e.g. transactions) require a more strict content definition model than currently defined for the price dissemination models.

3 Fundamentals

3.1 Overview

This section describes the key terminology and concepts needed to understand the Open Message Model implementation within Refinitiv Real-Time Distribution System.

3.2 Provider/Consumer Model

The service-oriented-architecture[1] paradigm has always been employed within Refinitiv Real-Time Distribution System. In fact, Refinitiv Real-Time Distribution System and its predecessors implemented a service oriented architecture before the term became popular. This is due to the fact that market data systems historically have had to solve the problem of integrating multiple “services” in an interoperable way. The solution to the problem continues to include a loose coupling between the offered services and their users. Refinitiv Real-Time Distribution System accomplishes this through comprehensive and compatible APIs (Refinitiv Real-Time APIs, collectively referred to as the RTSDK) combined with a realtime streaming service-oriented architecture implementation.

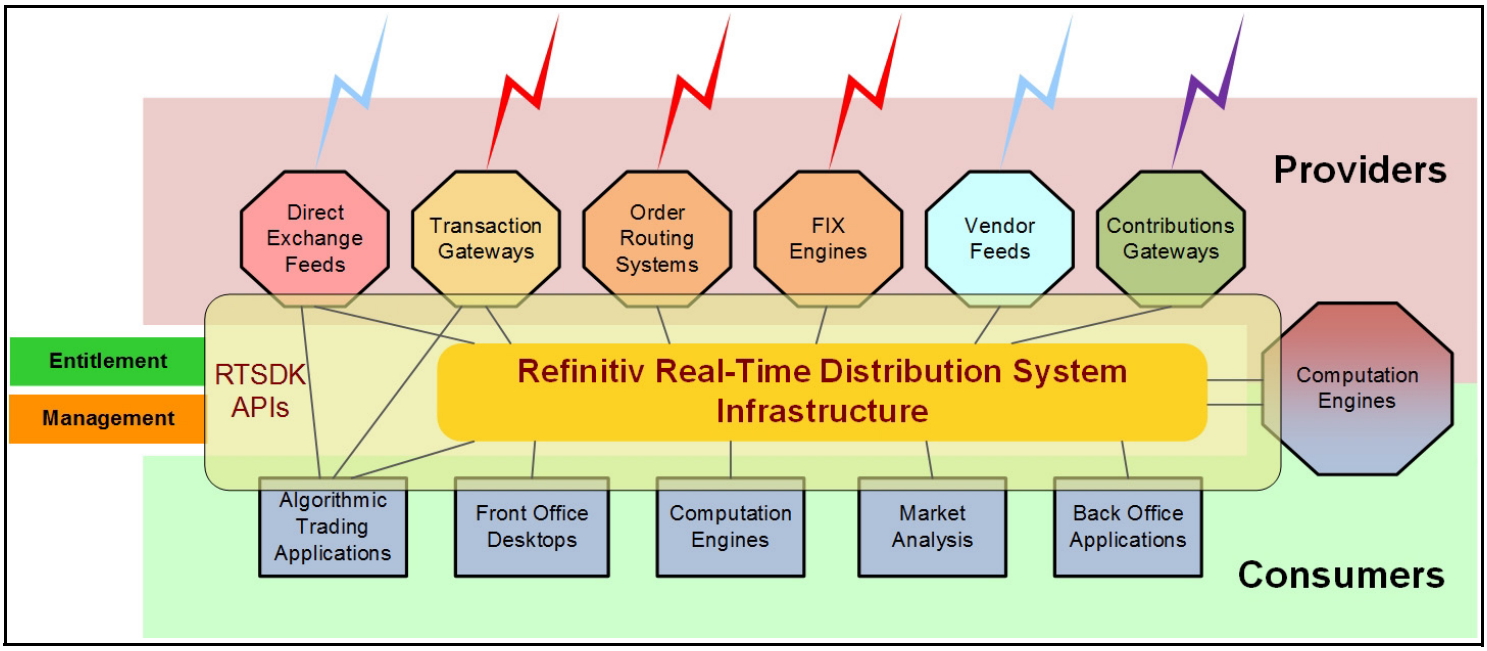

Figure 2. Provider/Consumer Model

The Refinitiv Real-Time Distribution System infrastructure and its associated APIs are designed to integrate the capabilities (data and workflows) of packaged interoperable services for generic utilization. All applications that interact with Refinitiv Real-Time Distribution System infrastructure (other than applications handling entitlement and management) can be classified as either a consumer or a provider:

- Consumers interact with providers, either directly or via the Refinitiv Real-Time Distribution System, to access data, services, and various capabilities. Consumers must have proper permissions (i.e., entitlements) to access data and services.
- Providers implement services and provide data. The term ‘provider’ is used instead of publisher, because this type of application can both publish data (e.g. direct exchange feed, vendor feed) and also provide access to a workflow (e.g., exchange gateway for transactions, vendor contributions).

Publishers are now defined within their respective roles: some publishers are implemented as full blown providers while others post content into the system but act primarily as service consumers.

1. For a description of Service Oriented Architecture, refer to the Medium article, “What Is Service-Oriented Architecture?”.

3.3 Services

Services offer a unique identification scheme, or namespace, for the packaging of different types of providers. They also give a convenient mechanism to manage large sets of data and/or capabilities. All request/response traffic is directed to, and received from, a service provider. Services can be categorized by many different criteria including business classification, vendor name, exchange content and transactions gateway.

Services are dynamic in nature and thus may be created or removed on the fly. Their existence and characteristics are detected dynamically by the system. Consumers rely upon these events for access to new capabilities as they are added or removed from Refinitiv Real-Time Distribution System.

3.4 Access Point

Consumers utilize the content and capabilities from providers through access points. Consumer access points currently manifest themselves in two different forms:

- Direct: A service provider that supports direct connections implements a concrete consumer access point. Multiple consumers can directly interact with the provider in order to access the content and capabilities offered by the service. Datafeeds (e.g. EED, Refinitiv Data Feed Direct) or exchange gateways typically implement this style of access point.
- Proxy: When certain capabilities are required (e.g. service provider integration, large scale distribution, local content management, resiliency, etc.), Refinitiv Real-Time Distribution System components can be placed between the concrete service providers and the consumers. In this case the consumers interact with Refinitiv Real-Time Distribution System components acting as proxies to the concrete providers. The Advanced Distribution Server (ads) implements a proxy access point.

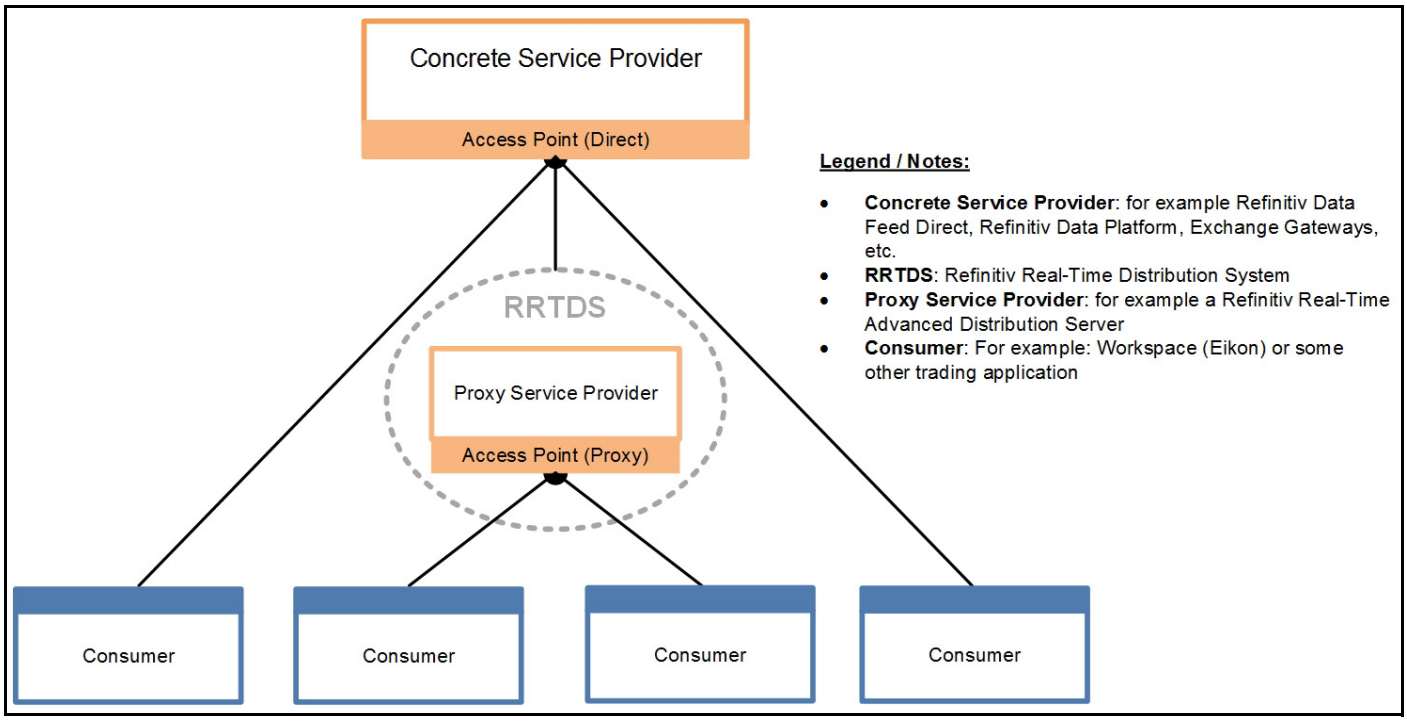

Figure 3. Consumer Access Points

4 Wire Formats

4.1 Overview

A wire format defines a hardware-neutral, external data encoding for use in transmitting data between components which may exist on multiple machines. Most network-based communications happen over a local area network (LAN) or a wide area network (WAN). Various incompatible wire formats exist that allow the transferring of simple data types (e.g. integers, floats, time) between different parties.

Previously, Refinitiv Real-Time Distribution System and Refinitiv Real-Time APIs and systems defined and utilized several different wire formats. In fact, the wire format was usually coupled with the type of data transported. Though many wire formats utilized common techniques, each had its own definition. Legacy examples of wire formats include:

- ***Marketfeed***: A delimited ASCII based data format that is used to represent logical records. Field identifiers are used to identify particular fields and require a dictionary to make sense of the data. The Marketfeed format is documented and encoded/decoded within applications.
- ***QForms***: A binary based wire data format that is used to represent logical records. Field identifiers are used to identify particular fields and require a dictionary to make sense of, and parse, the data. QForms encoding and decoding is implemented within the TIBMsg API. QForms is no longer supported as of Refinitiv Real-Time Distribution System version 3.0.
- ***TibMsg***: A self-describing binary and ASCII based wire format that is used to represent logical records. TibMsg encoded messages are typically much larger than QForms or Marketfeed due to their self-describing nature. TibMsg encoding and decoding is implemented within the TIBMsg API. TibMsg is no longer supported as of Refinitiv Real-Time Distribution System version 3.0.
- ***ANSI Page***: A page format that uses ANSI sequences to express display oriented data.
- ***Sink Source Library***: A point-to-point binary-based wire format that was used to transport protocol messages between components. It is was implemented in Sink Source Library 4.x APIs.
- ***Refinitiv Reliable Management Protocol***: A multicast binary-based wire format used to transport protocol messages between Refinitiv Real-Time Distribution System components (such as the Refinitiv Real-Time Advanced Data Hub and Refinitiv Real-Time Advanced Distribution Server).

4.2 Refinitiv Wire Format

In Refinitiv Real-Time Distribution System, these different formats are replaced by a single wire format: the Refinitiv Wire Format. Regardless whether the layer performs transport behavior, or represents data formats, the Refinitiv Wire Format is always used for encoding.

The Refinitiv Wire Format uses binary encoding, bit-wise operations and reserved values in order to optimize all network distribution. The Refinitiv Wire Format is automatically encoded into network byte order (or big endian) for transmission and then decoded into the machine-specific byte order for access. When these features are properly combined, the binary nature of the Refinitiv Wire Format provides smaller messages that are easier and more efficient to encode/decode. This results in an increase in total throughput and a decrease in overall latency.

The base Refinitiv Wire Format defines primitive data types or building blocks that can be transmitted between multiple components. The Open Message Model uses these primitive types to represent more complex transport and data formats between components. In effect, the Open Message Model uses the Refinitiv Wire Format and extends it to build more complex messages semantics when needed.

Some data type encodings that make up the base Refinitiv Wire Format specification include:

- Fixed-sized signed and unsigned 8-bit, 16-bit, 32-bit and 64-bit integers. All values are encoded in network byte order and use the same number of bytes as defined by the value used (e.g. 16-bit is encoded in 2 bytes). The two's complement IEEE format is used to encode signed integer values.
- Special value variable sized unsigned 16-bit and 32-bit integers. These values must always contain a single byte. If the byte is less than a reserved value, then that byte is the actual value. Otherwise the value of the first byte is used to infer the number of bytes in which the value is encoded after this first byte.
- Reserved bit-variable-sized unsigned 15-bit, 30-bit and 62-bit integers. These values must also always contain a single byte. However, one, or two, high-order bits are reserved in the first byte to indicate the total length of the integer value.
- Real (decimal) values that contain a maximum 64-bit integer coefficient and a 6-bit integer exponent. The two high bits in the exponent byte can be used to identify the length of the coefficient for bandwidth optimizations.
- IEEE standard 754-1985 floating point numbers (4-byte floats and 8-byte doubles).
- Time (hour, minute, second, millisecond, microsecond, and nanosecond), date (day, month, year), and combined date/time.
- Multiple buffers that have different maximum lengths depending on the integer encoding used.
- ASCII, UTF-8 and RMTES strings.
- Array of any of the above data types.

5 The Open Message Model

5.1 Overview

The Open Message Model forms the basis for all messaging through re-usable transport and data abstractions. All current and future content (i.e. data and messages) will be described, structured, accessed, and produced using Open Message Model concepts. Also, most new data models can be realized without modifications to the API that you use or to Refinitiv Real-Time Distribution System components. Low latency and high throughput of messages was a main design goal of the Open Message Model. Abstractions have been carefully created in order to properly weigh performance with overall flexibility. When these goals came in direct conflict, multiple options have been employed to maintain performance where necessary.

When looking at the many different types of information and how that information is accessed, a large number of similarities can be identified. These commonalities can be found in both how you access the data and what form the data itself takes. The request/response with interest pattern that is used to access market price (Level 1) content can be used to access all event streams (i.e. open ended, or streaming, requests). The main difference is the meaning between the responses and any event stream updates. Some examples of this concept include:

- Market-by-order, or order book, information can be accessed using the request/response with interest pattern. The response to the request contains the actual order book information that is structured in a common data format. Updates represent modifications to the book received in the response and need to be applied appropriately.
- Streaming news headlines can also be accessed using the request/response with interest pattern. The response acknowledges the request and does not actually contain any information. Updates represent actual asynchronous headlines that are structured in a common format. Updates also contain an extra permissions expression that must be checked against the user's profile to verify access to the headline.

The Open Message Model design approach uses generic messages, attributes and data formats that defer specific semantics to the Domain Message Model definition. Data representations are also abstracted and separated from the behavior in order to offer the most flexibility. Applications can provide and consume any services using the transport and data abstractions defined in the Open Message Model.

The Open Message Model is divided into two different layers, each of which is described in detail in this section:

- The transport defines the interactions that can take place between a consumer and a provider, such as a request for information or a request to synchronize an existing set of information. The transport is described in Section 5.2.
- Data abstractions are the generic formats for data, such as the field list, vector, etc. These generic formats can be combined (e.g. a vector of field lists or a vector of vectors) to create formats that suit different types of real world content, such as an order book. The data abstractions are described in Section 5.3.

5.2 Transport

The Open Message Model transport layer encapsulates all messaging syntax and semantics. It defines generic messages, and attributes within these messages, that defer actual meanings to the domain models. Generic interaction paradigms are combined with an extensible identification mechanism (or key) in order to provide the most flexibility. Built-in state information exists for both the data and any event streams. A request priority scheme exists that can be used to prioritize request traffic (in terms of both initial request and any possible recovery). Event streams can be broken into different groups for efficient state transitions.

Refresh and/or update messages can optionally contain permission expressions that define any extra requirements needed to receive the messages. These permissions expressions, tied with each user's permissions profile, allow for full control of user access to content. To properly permission users, you must use the login domain model with an access point to authenticate users and retrieve their permissions profiles.

The transport supports both message fragmentation/reassembly and multi-part refreshes for dealing with potentially large data sets. Message fragmentation and reassembly is automatically handled by the transport. Multi-part refreshes allow providers to interleave time-sensitive data with large response data (even within the same event stream). Both concepts need to be used together when properly implementing a provider with a large data set.

The transport layer has been designed for both point-to-point and multicast based delivery.

5.2.1 Concepts

This section describes the following important concepts used in the definition of the Open Message Model transport layer:

- Interaction Paradigms
- Key
- State
- Quality-of-Service
- Group Identifier

5.2.1.1 Interaction Paradigms

Consumers work with service providers in order to make use of their many different capabilities. The actual behavioral collaboration, or abstract access methods, between these two parties have been classified into three different ***interaction paradigms***.

- Request/Refresh: A consumer requests snapshot data, or a static capability, from a service provider. The provider typically acts on the request and responds appropriately via a refresh. Responses can optionally contain data and be broken into multiple parts.[1] However once the response is complete, the interaction is complete. Examples include snapshot news searches, snapshot market-prices (level 1) and time-series data.
- Request/Refresh with Interest: A consumer requests data or capabilities that can change over time from a service provider. Providers act on the request and respond appropriately via a refresh. However, the interaction remains active (an event stream is created) and asynchronous events are sent to the consumer. These streaming events can represent changes in received data or notifications of new information. Examples include market pricing information (like market-by-order, market-by-price or market-price), news headlines and symbol lists.
- Listen/Send: Also known as publish/subscribe. A provider sends data without knowing whether consumers exist to receive the data. Likewise, consumers listen for data without specific knowledge of the providers.

1. Single message responses can never receive update messages; however certain domains that use multi-part responses may interleave updates with the response messages.

5.2.1.2 Key

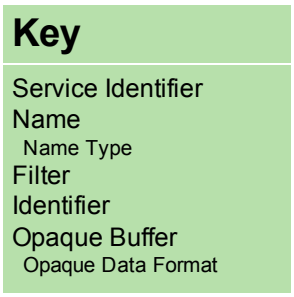

Figure 4. Key

Access to different types of data content and capabilities requires an extensible identification mechanism. The key implements the identification mechanism of the Open Message Model and replaces the old service name/item name (or subject) scheme. It is made up of many optional elements that are identified and given meaning by each upper-level domain. A key can include the following elements:

- ***Service Identifier***: An integer representing the identifier of the service provider. Not all domain models are directed to specific services (e.g. login).
- ***Name***: Name of the information requested (consists of a buffer and length). The maximum length of the name is 255 characters and it does not always have to contain printable characters. The name can be used to represent a symbol, a username, or any other named types of content.
- ***Name Type***: An enumeration representing the different forms the name can take. The name type can be used to represent different symbologies (e.g., RIC, ISIN, CUSIP), multiple user identification schemes (e.g. email, user identifier), or any other forms the name can take within that domain. The name type enumeration, for certain domain models, can be expanded to include third-party or client defined name types (e.g., internal symbologies).
- ***Filter***: A bitmap of optional data content/formats logically separated by the provider. The maximum number of filters (or bits) is 32. This high level filtering capability has been used to break up a large amount of source directory information into smaller request-able categories. It is not designed for field level requesting within a field or element list.
- ***Identifier***: A simple integer-based value for identifying different information. For example, you can use the identifier to represent a version number within some request. If a domain has multiple versions to allow it to evolve, the identifier can specify which version of the domain is being requested or provided.
- ***Attrib***: An opaque buffer, or extensibility mechanism, allowing for a complex identification mechanism (e.g. query, complex filters, etc.). The opaque element can be used to provide an SQL/XML query to a historical database or any other complex parameter lists to a provider. Attrib can house up to 32k bytes of content.
- ***Opaque Data Format***: The data format of the opaque data buffer. Data sent over the wire is limited to 32,767 bytes. Because particular applications might have their own limits, Refinitiv recommends that you also refer to the documentation for the specific applications and APIs that you use.

A generic request key can be used for generic identification, forwarding, and updating of messages. This allows items to be identified/ matched using a much more complex scheme than just a name or a subject. Domain models use the key by giving domain specific meaning to the different key elements.

5.2.1.3 State

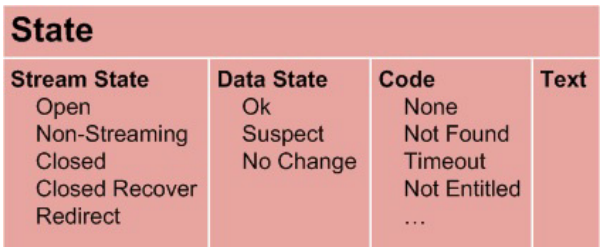

Figure 5. State

Generic request level status, or the request condition, is made available through a generic state model. This model represents a normalized view of status and separates stream state from data state. Response status, in terms of both requests and unsolicited responses, is not contained within the generic state model defined within this section. This allows for a clear separation of response state from event stream state. The elements that make up state include:

- ***Stream State***: the state of the event stream when using the request/response with interest paradigm. All non-streaming requests will contain a stream state of non-streaming.
  - Open: the event stream is actively open and asynchronous events can happen at any time.
  - Non-Streaming: the information/capability requested does not support streaming semantics or the request was for non-streaming content. The item will not incur interest after the final response. For example, a consumer can ask for a streaming news search, the provider can respond with the data and a stream state of non-streaming.
  - Closed: the event stream is closed and is not available from the provider at this time. It may be made available in the future.
  - Closed Recover: the event stream has been closed and should be re-opened by the consumer.
  - Redirected: the information, or capability, requested is available somewhere else as identified in the key (i.e. similar to an HTTP redirect). The consumer should re-request using the key provided in the response message. Any parameter in the key, including the service identifier, can be re-defined in a redirect.
- ***Data State***: represents the quality of the data in the response or in the event stream.
  - Ok: the data's state is ok.
  - Suspect: the data's of the data is unknown or stale.
  - No Change: the data's state has not changed from the previously communicated state.
- ***State Code***: additional status information for the event stream or data state. Not needed for generic state processing. Developers may define new state codes when necessary. Some include …
  - None: no additional information is available.
  - Not Found: the item is not available from the provider.
  - Timeout: the request has timed-out.
  - Not Entitled: the consumer cannot access the item.
  - Invalid Argument: an invalid argument was passed in the request.
  - Usage Error: illegal usage of messages or message data content.
  - Preempted: the event stream has been preempted in order to create room for another event stream.
  - JIT Conflation Started: just-in-time, or backpressure-based, conflation has started.
  - Realtime Resumed: just-in-time conflation has ended.
  - Failover Started: source mirroring failover has started on a service.
  - Failover Completed: source mirroring failover is complete for a service.
  - Gap Detected: A service has detected a message gap from data originator (e.g. exchange).
  - No Resources: No more resources exist in order to handle the request.
  - Too Many Items: The user has reached the limit for maximum number of event streams as defined by the system administrator.
  - Already Open: The event stream is already open for the consumer.
  - Service Unknown: The service identifier in the request key does not exist.
  - Not Open: The event stream is not open and cannot be closed.
- ***Text***: textual information about the stream and/or data state.

5.2.1.4 Quality-of-Service

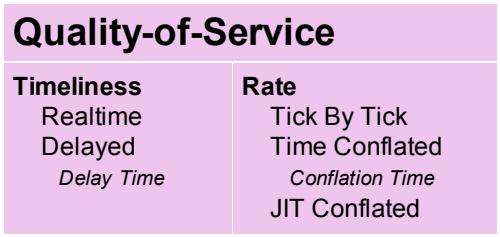

Figure 6. Quality-of-Service

Domains may classify data/events in order to provide differentiated tiers of service. ***Quality-of-service*** provides this classification and is divided into orthogonal sets of distinct properties. The two properties that make up QoS include:

- ***Timeliness***: age of the data
- ***Rate***: maximum period of change in data (for streaming events)

Timeliness of data has been broken into the concepts of ***realtime*** or ***delayed***. Realtime implies no delay is applied to the data (it is up-to-date and sent by the provider as soon as it happens). Delayed implies a view of the data in the past and usually includes a delay time.

Rate of change can be categorized as ***tick-by-tick***, ***time-conflated***, or ***just-in-time conflated***. Tick-by-tick implies the consumer receives every update, or change, in the data. Data is conflated when multiple events are combined in a way that preserves the final view of the content. Conflation can be based on time or may vary based on more complex parameters (e.g. channel capacity, congestion, etc.).

Quality-of-service always contains a single value in each dimension (e.g. realtime/tick-by-tick, realtime/time-conflated). Consumers can specify quality-of-service parameters on requests, and providers will identify quality-of-service attributes for all data and event streams. Note that not all domains utilize the quality-of-service concepts.

5.2.1.5 Group Identifier

Item groups are used to efficiently manage the state of many event streams that originate from a single service provider (e.g. they can be used to represent subservices). Using a single status to report that an entire item group has become stale is much more efficient than using a status for each of the separate event streams. A feed that maintains multiple backend connections could group items based on the route providing the item.

Each open event stream belongs to an item group as defined by the ***Group Identifier***. The item group association is set by the service in the initial response and can be modified with a status or another response message. Services can establish item groupings on any basis that makes sense to the provider's needs. For example, a service that maintains multiple backend communication channels might establish an item group for each distinct channel. This would allow the service to mark all of the items from a given channel as being suspect while not modifying the state of items provided by the other channels.

5.2.2 Messages

The transport layer defines the messages, and their semantics, that can flow between provider and consumer applications. These symmetric messages are used by both the consumer and provider in order to communicate. This not only avoids redundant messages and methods, but also reduces the learning curve when building either application type. Not all of the messages defined in this section are used by all domain models. Each domain model identifies which messages are used and defines their actual meaning.

Most messages contain data that is hierarchal and extensible. It may be sent from a provider to a consumer or from a consumer to a provider. Data can take the form of attribute/header or payload and is given meaning by the domain model definitions. Attribute data typically contains additional message knowledge (e.g. state, sequence number) while payload data is information that satisfies some business purpose (e.g. level 2 data, FIX messages). See the message definitions in this section for attribute data details (names of optional attributes are italicized) and see Section 5.3 for details on payload data.

5.2.2.1 Message Base

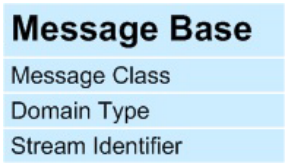

Figure 7. Message Base

The messages defined all contain the same base attributes.

- ***Message Class***: the message classification (e.g. request, response, update, etc.).
- ***Domain Type***: the domain model represented by this message (e.g. login, market-price, news headlines, etc.).
- ***Stream Identifier***: an optimization that allows applications to refer to event streams with a signed 32-bit value instead of the full key. This defined value can be sent in updates instead of the key in order to save bandwidth and simplify matching. Consumers that want the key placed in every update can optionally request this behavior from the provider.[2] Consumers define a positive stream identifier in requests. Providers define negative stream identifiers when pushing non-requested information (e.g. broadcast feed or non-interactive providers).

2. APIs can expose the **`streamID`** concept as a unique, API-assigned handle per stream. Enterprise Transport API exposes stream IDs, the Robust Foundation API exposes this differently between providers and consumers, and Enterprise Message API exposes handles on both sides.s.

5.2.2.2 Request

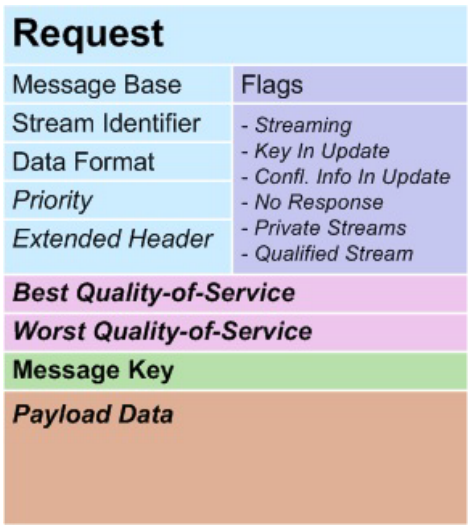

Figure 8. Request Message

A ***request*** message is sent from a consumer to a provider when it wants to request some data, or a capability, available from the provider. It can also be used to obtain a new response (e.g. synchronization point) or change selected attributes (e.g. priority) for an already open event stream. The generic attributes, and their meanings, that make up this message include:

- ***Message Base***: domain model identifier (see Section 5.2.2.1).
- ***Stream Identifier***: integer value representing the event stream (see Section 5.2.2.1). It can also be used to match the request and responses.
- ***Flags***: specifies some of the different request options available.
  - Streaming: the application wishes to create an event stream based on this request (i.e. the request/response with interest interaction paradigm).
  - Key In Update: the consumer wants the key encoded in every update.
  - Conflation Information In Update: the consumer wants any update conflation information (e.g. number of updates conflated) included in the update.
  - No Response: the consumer is trying to update some attribute information (e.g. priority) about a previous request, or event stream, and does not want any responses.
  - Private Streams: Indicates that the resulting stream should be open and be private – not fanned out or cached to be shared with other users. If a provider can honor this request, it responds with a Private Streams flag in a Refresh or Status and then subsequent messages on the stream are considered private (e.g., not cached or fanned out, with no stream recovery by intermediate components).
  - Qualified Stream: Indicates that stream behaviors are requested in the payload. Stream behaviors include, but are not limited to: end-to-end flow control, end-to-end reliability, end-to-end guaranteed delivery, and end-to-end authentication.
- ***Data Format***: generic format of the payload data (see Section 5.3).
- ***Priority***: when specified, indicates the relative importance of the request and resulting event stream.
- ***Extended Header***: an optional extension to the request message in case an attribute is identified that currently doesn't fit into the request message header.
- ***Best Quality-of-Service***: when specified, indicates the upper bounds of the quality-of-service required by the application (e.g. application prefers realtime/tick-by-tick data but will accept down to the worst QoS).
- ***Worst Quality-of-Service***: when specified, indicates the lower bounds of the quality-of-service required by the application. When not specified, the best quality-of-service defines the exact quality-of-service required by the application (e.g. application requires realtime/tick-by-tick data).[3]

3. Worst QoS currently not supported.

- ***Message Key***: The key (e.g. service identifier, symbol, symbology, etc.) representing the data content or capability requested (see Section 5.2.1.2).
- ***Payload Data***: the actual raw encoded data buffer. The actual data format type is identified by the data format attribute above. Current service providers, that primarily publish data, typically don't accept payload data in requests. However, new types of services have been identified that may require payload data within requests. Payload limitations are determined by the transport layer that transmits it. For example, if an API from the RTSDK (e.g., Enterprise Transport API) uses TCP or Reliable multicast transport, it is going to be a 2GB maximum size message presuming you use the full extent of the transport fragmentation. But if using shared memory or unreliable multicast, message size is seriously curtailed (i.e., 3k – 6k depending on the configuration). A Refinitiv Real-Time Distribution System backbone (using Refinitiv Reliable Control Protocol/Refinitiv Reliable Management Protocol) has other limitations. For specific details, refer to the appropriate product documentation.

5.2.2.3 Refresh

Figure 9. Refresh Message

The ***response*** message is used to respond to a request or can be used to asynchronously change the attribute information or data of an already opened event stream (i.e. unsolicited images). The ***solicited*** flag within the options identifies whether or not the message was solicited (i.e. a response to a request). Response messages with negative stream identifiers allow providers to asynchronously create event streams without the consumer ever requesting the stream (e.g. non-interactive). The generic attributes, and their meanings, that make up this message include:

- ***Message Base***: domain model identifier (refer to Section 5.2.2.1).
- ***Stream Identifier***: integer value representing the event stream (refer to Section 5.2.2.1). It can also be used to match the request and responses.
- ***Flags***: specifies some of the different refresh options available.
  - Solicited: indicates whether the message is a solicited response to a request or an unsolicited response to an existing event stream.
  - Complete: indicates that the payload data in the response is complete. Some domain models require a single response with payload data; others allow multi-part responses of payload data that will have this flag set in the last message.
  - Clear Cache: an indication that any previous last value payload data cache for the event stream needs to be deleted.
  - Do Not Cache: it does not make sense to apply the payload data in this response to a last value cache (e.g. responses for unique queries like time-series data).
  - Private Streams: Indicates the successful completion of a private stream (if present on the initiating request).
  - Qualified Stream: Indicates that the response payload includes qualified behaviors. Stream behaviors include, but are not limited to: end-to-end flow control, end-to-end reliability, end-to-end guaranteed delivery, and end-to-end authentication.
- ***Data Format***: generic format of the payload data (refer to Section 5.3).

- ***Group Identifier***: the group identifier of the event stream (refer to Section 5.2.1.5).
- ***Sequence Number***: when specified, indicates the last sequence number associated with the event stream as received by the concrete service provider (e.g. exchange sequence number). These sequence numbers do not have to be sequential for a single event stream (e.g. market-price updates for IBM.N may be non-contiguous).
- ***Permissions Expression***: the requirements needed to access the item and/or event stream. Typically utilized in the access point and only sent to the consumer application when requested using the login domain.
- ***Extended Header***: an optional extension to the response message in case an attribute is identified that currently doesn't fit into the response message header.
- ***State***: indicates the stream state and data state for the response (refer to Section 5.2.1.3).
- ***Quality of Service***: when specified, indicates the actual quality-of-service of the actual response and/or event stream (refer to Section 5.2.1.4).
- ***Message Key***: the key (e.g. service identifier, symbol, symbology, etc.) representing the data content or capability in the response (refer to Section 5.2.1.2). The response key can be different from the request key if the provider supports aliasing (i.e. symbology mapping). For example, a consumer may request market-price information for an ISIN, and the provider can respond with the data and indicate in the response key that the item is actually referred to as a RIC.
- ***Payload Data***: the actual raw encoded data buffer. The actual data format type is identified by the data format attribute above.

5.2.2.4 Update

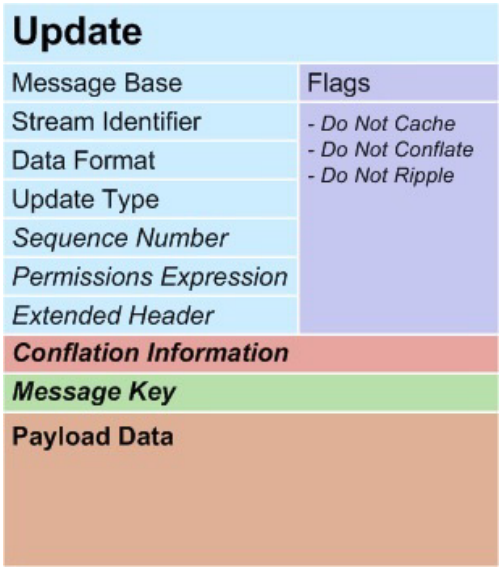

Figure 10. Update Message

The ***update*** message is used to represent asynchronous payload data events associated with an already opened event stream. Domain models may assign different meaning to updates depending on the actual content modelled. The generic attributes, and their meanings, that make up this message include:

- ***Message Base***: domain model identifier (refer to Section 5.2.2.1).
- ***Stream Identifier***: integer value representing the event stream (refer to Section 5.2.2.1).
- ***Flags***: specifies some of the different update options available. In some cases update options allow a provider to put some context into the update that in the past had to be inferred by all recipients.
  - Do Not Cache: it does not make sense to apply the payload data in this update to a last value cache (e.g. news headlines, indications-of-interest, advertised-trades, etc.).
  - Do Not Conflate: the payload data in this particular update should not be conflated (e.g. market-price trades, news headlines, etc.).
  - Do Not Ripple: do not ripple any fields within the update (e.g. market-price closing run).
- ***Data Format***: generic format of the payload data (refer to Section 5.3).
- ***Update Type***: the type of update as defined by the domain model (e.g. trade, quote, news event for market-price domain). Update types are represented as an expandable enumeration.
- ***Sequence Number***: when specified, indicates the sequence number associated with the event stream as received by the concrete service provider (e.g. exchange sequence number). These sequence numbers do not have to be sequential for a single event stream (e.g. market-price sequence numbers for IBM.N may be non-contiguous).
- ***Permissions Expression***: the requirements needed to access this particular update (e.g. each news headline is separately permissioned). It does not affect the permissions expression used for accessing the entire event stream. Typically utilized in the access point and only sent to the consumer application when requested using the login domain.
- ***Extended Header***: an optional extension to the update message in case an attribute is identified that currently doesn't fit into the update message header.
- ***Conflation Information***: when requested, provides the information about any conflation logic that may have been applied to this event. Current parameters include the number of events conflated and/or the time between the conflated events.
- ***Message Key***: when requested, the key (e.g. service identifier, symbol, symbology, etc.) representing the event stream for the update (refer to Section 5.2.1.2). If the provider aliased the key in the response, then it matches the aliased key. By default the update key is not included in the update in order to save bandwidth.
- ***Payload Data***: the actual raw encoded data buffer. The actual data format type is identified by the data format attribute above.

5.2.2.5 Status

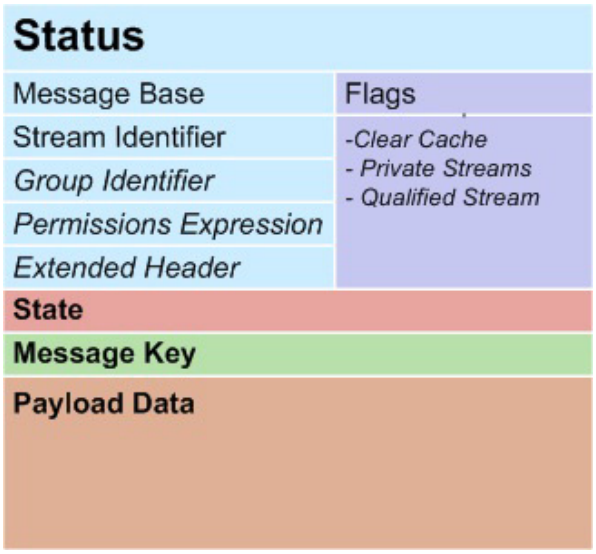

Figure 11. Status Message

The ***status*** message is used to represent asynchronous attribute changes associated with an already opened event stream. The generic attributes, and their meanings, that make up this message include:

- ***Message Base***: domain model identifier (refer to Section 5.2.2.1).
- ***Stream Identifier***: integer value representing the event stream (refer to Section 5.2.2.1).
- ***Flags***: specifies some of the different status options available, including:
  - Clear Cache: specifies to delete any previous, last-value, payload data cache for the item.
  - Private Streams: If this flag was also present on the initiating request, its presence here indicates successful completion of the private stream. If this flag was absent from the initiating request, its presence here indicates that the stream is available only as private.
  - Qualified Stream: Indicates that the status payload includes qualified behavior requirements (if requested options are not supported but others are available, this is also indicated in the status payload). Stream behaviors include, but are not limited to: end-to-end flow control, end-to-end reliability, end-to-end guaranteed delivery, and end-to-end authentication.
- ***Group Identifier***: when present, defines the new group identifier of the event stream (refer to Section 5.2.1.5).
- ***Permissions Expression***: the new requirements needed to access the event stream (i.e. permissions change for access to an instrument). Typically utilized in the access point and only sent to the consumer application when requested using the login domain.
- ***Extended Header***: an optional extension to the status message in case an attribute is identified that currently doesn't fit into the status message header.
- ***State***: indicates the new event stream state and data state for the item (refer to Section 5.2.1.3).
- ***Message Key***: the key (e.g. service identifier/symbol/symbology) identifying the event stream (refer to Section 5.2.1.2). If the provider aliased the key in the response, then it matches the aliased key.
- ***Payload Data***: the actual raw encoded data buffer.

5.2.2.6 Close[4]

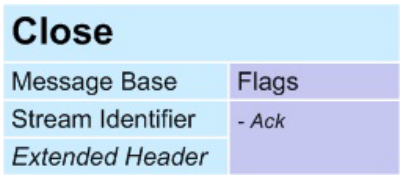

Figure 12. Close Message

The ***close*** message is used to cancel an outstanding request or to stop an existing event stream. The generic attributes, and their meanings, that make up this message include:

- ***Message Base***: domain model identifier (refer to Section 5.2.2.1).
- ***Stream Identifier***: integer value representing the request or event stream to close (refer to Section 5.2.2.1).
- ***Flags***: specifies some of the different close options available including: ***Ack*** (the provider should acknowledge the close when received and applied).
- ***Extended Header***: an optional extension to the close message in case an attribute is identified that currently doesn't fit into the close message header.

5.2.2.7 Acknowledgement

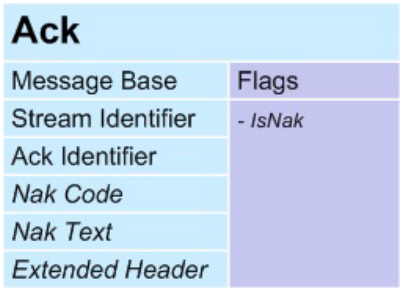

Figure 13. Ack/Nak Message

An ***ack*** message is used to acknowledge an outstanding request or close. The generic attributes, and their meanings, that make up this message include:

- ***Message Base***: domain model identifier (refer to Section 5.2.2.1).
- ***Stream Identifier***: integer value representing the request or event stream to acknowledge (refer to Section 5.2.2.1).
- ***Flags***: specifies some of the different acknowledgment options available, including: **IsNak** (this particular message represents a nak and contains the nak code and text).
- ***Ack Id***: the acknowledgment identifier.
- ***Nak Code***: the nak code (only set for nak messages).
- ***Nak Text***: the nak text (only set for nak messages).
- ***Extended Header***: an optional extension to the ack message in case an attribute is identified that currently doesn't fit into the ack message header.

4. Not an explicit message in Robust Foundation API 6.

5.3 Data Abstractions

The Open Message Model defines data abstractions that are used to represent disparate data models and are understood by applications and infrastructure components (Refinitiv Real-Time Distribution System, Refinitiv Data Feed Direct) alike. They include field/value pair constructs in conjunction with sequential and associative containers. Open Message Model data formats are sufficiently flexible to contain additional data abstractions in an arbitrary nesting hierarchy. This flexibility enables the domain messaging models to realize comprehensive data models on top of the Open Message Model. The payload data formats contained within the transport messages are defined by the data abstractions described in this section.

5.3.1 Concepts

This section describes basic concepts used in the definition of all of Open Message Model data abstractions:

- Primitive Types
- Multi-part Refresh
- Record-sets
- Summary Data

5.3.1.1 Primitive Types

Basic data types are contained within certain data formats and attribute information. They allow for the representation of many different types of data content within the given formats. The current defined data types include:

- Signed integer values (one sign bit and 63 significant bits)
- Unsigned integer values (64-bit and no sign)
- Real number values that contain both a coefficient and an exponent (64-bit coefficient with an exponent from +7 thru -15). Examples include:
  - 25.653 can be represented as a real number with coefficient = 25653 and exponent = -3
  - -100.01 can be represented as a real number with coefficient = -10001 and exponent = -2
  - 1256000 cam be represented as a real number with coefficient = 1256 and exponent = 3
- Time in the form hours, minutes, seconds, milliseconds, microseconds, and nanoseconds
- Date in the form day, month, year
- Combined date and time
- IEEE 754-1985 floating point numbers (32 and 64-bit)
- Enumeration (16-bit unsigned integers)
- Binary buffer
- ASCII and UTF-8 strings
- RMTES String: multilingual text encoding standard (can contain partial field update semantics)

5.3.1.2 Multi-part Refresh

All data abstractions generically support fragmentation across multiple message instances for potentially large-sized content. Fragments are broken on logical, entry boundaries to simplify receiving logic. Receiving applications can process each fragment independently (i.e. decode, cache) without waiting for all fragments. Domain models define whether or not they support fragmentation.

Most structure-based data abstractions that support fragmentation provide a total count hint, which is the sending application's suggestion about the total number of entries within the structure across all fragments. The receiving application may choose to use the hint to pre-allocate sufficient memory for processing.

5.3.1.3 Record-sets

The Open Message Model supports a ***record***-set concept that can be used to optimize bandwidth for record based data (i.e. field and element lists). Record data offers a way of representing logical information as a collection of field/value pairs. The field contains an entry identifier and possibly other attributes while the value contains the actual data content for the entry. Entry identifiers can be self describing (like names and data types in element lists) or reference an independently distributed data dictionary (like field identifiers in field lists).

Standard record based data is fully encoded as repeating field information followed by value data. This simple encoding unites the entry data and definition. The key benefits to this encoding are its ease-of-use and similarity to existing record based formats (e.g. Marketfeed, TibMsg and QForms).

Record-sets offer a bandwidth optimization that can be employed to split entry definitions from raw entry datum for record based content. They allow for the common entry definitions to be separately defined once and assigned an identifier (or set-id). The actual data for the entries are then encoded without the definitions; resulting in substantial bandwidth savings. This optimization can help with repetitively structured content where each entry in the structure contains the same definitions (e.g. order books and time-series). It can also help with repeating record entry definitions across multiple messages (e.g. North American equity trades and quotes).

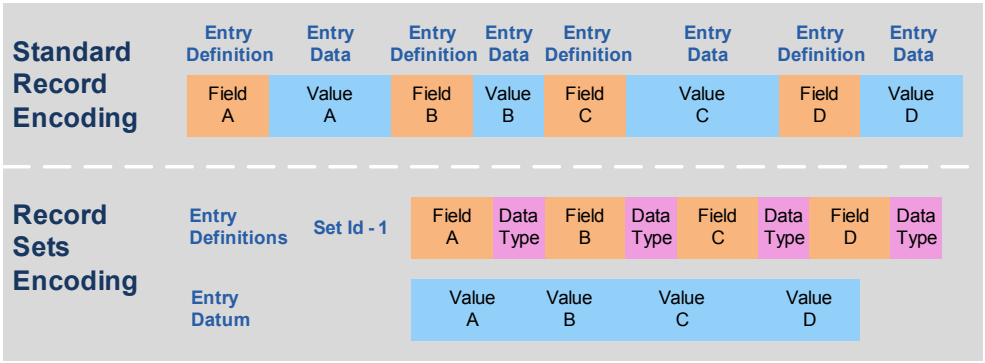

Figure 14. Record-Set Encodings

Record-sets are identifiable by a set-id and defined at either a ***message*** or ***global*** scope.[5] Message scope implies the entry definitions are sent in the same message as the entry datum. It is most commonly used for encoding repetitively structured content such as an order book or a time-series. Global scope implies the entry definitions are defined once, in a record-set dictionary, and re-used across many different messages. It is most commonly used for encoding many different messages that contain the same entry definitions (e.g. equity quotes, equity trades, etc.).

Consumer applications do not need to know the difference between standard record encoding and record-sets based encoding. This is because the decoding libraries default to making content encoded as record-sets look like field/value pairs (i.e. standard record encoding). However, providers do need to know the difference since they have to make a choice when encoding the actual record content.

5. Only Enterprise Transport API supports Global set definitions.

Developers can inter-mix standard record encoding with record-set encoding within a single message. This allows record-sets to be defined for the most common cases while supporting the extensibility needed in the enterprise. This is best described using an example. A system could define a global record-set to represent a market-price quote update. When a quote update is received that also modifies today's high price, the provider could encode the quote data using the quote record-set and then append standard record encoding for the extra fields (e.g. fields E and F in Figure 15).

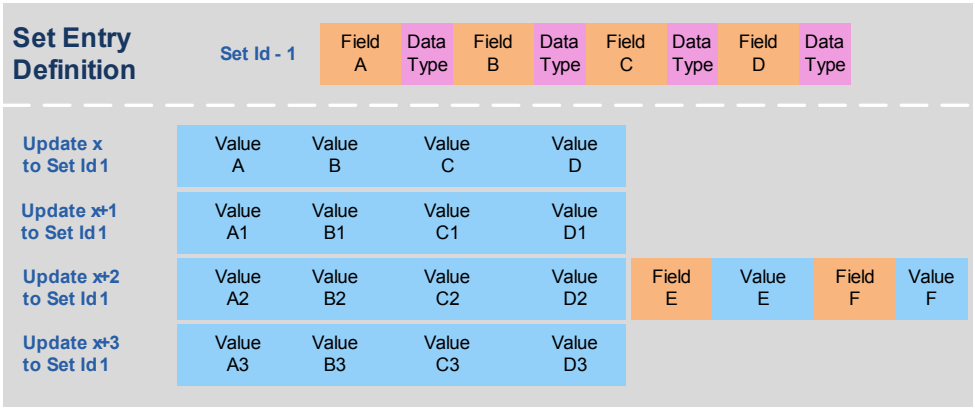

Figure 15. Extended Record Set

5.3.1.4 Summary Data

Repetitively structured content may have data that pertains to the entire structure. This meta-data typically describes the information contained in each structure entry. For example, ***summary data*** could specify the currency for the price of each entry, or rules regarding and sorting of the entries. Structured based data abstractions (e.g. vector, map and series) all support summary data to house this type of information efficiently.

5.3.2 Data Formats

The data abstractions supported through the Open Message Model are realized through data formats. This section defines the generic data formats that are available and must be used to model all content. The following data formats are defined in terms of structure and the different update semantics needed to keep those structures up-to-date within an event stream.

It is important to understand that these structures are in fact a way of encoding messages and not “in memory” data structures. These data abstractions provide a way for an application to serialize and send a data structure that it uses over the wire to another application.

5.3.2.1 Element List

An ***element list*** is the simplest form of logical, or record-based, content. It represents a sequential container of self-describing field/value pair entries; each known as an element entry. Element entries are identified with a string based tag that self-describes the actual type of data along with the data itself. Element lists do not need any meta-data (i.e. data dictionary) in order to make full sense of the content. An optional element list number, unique within a service, can exist to optimize any caching logic (i.e. element lists with the same element list number should contain the same entries/tags/types).

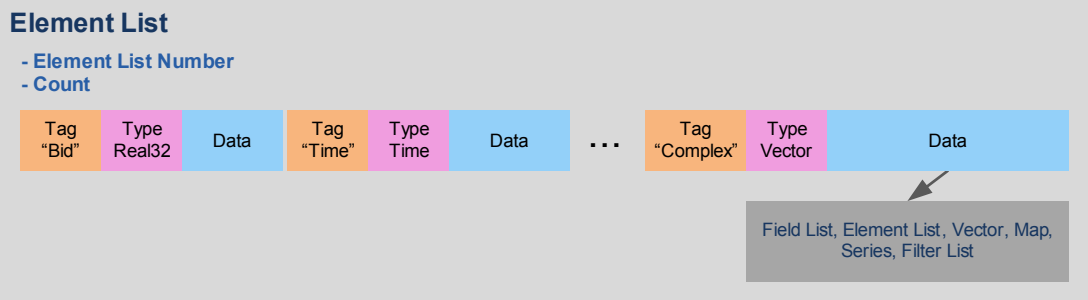

Figure 16. Element List

Element lists can be bandwidth intensive, but offer ease-of-use. They can make sense for domains that don’t have very high update rates. In certain circumstances, the record-set concept (refer to Section 5.3.1.2) can be used when encoding an element list in order to reduce the bandwidth needed.

5.3.2.2 Field List

A ***field list*** is a more complex, or optimized, form of logical/record-based content. It represents a sequential container of Field IDentifier/ value pair entries; each known as a field entry. Field entries are identified with a signed 2-byte integer and only contain the actual data for the field. A data dictionary is needed to convert the field identifiers into a tagged name, data type and possible maximum cache length. An optional field list number, unique within a service, can exist to optimize caching logic (i.e. field lists with the same field list number should contain the same entries/Field IDentifiers). Field list numbers are analogous to the record template numbers that exist in Marketfeed.

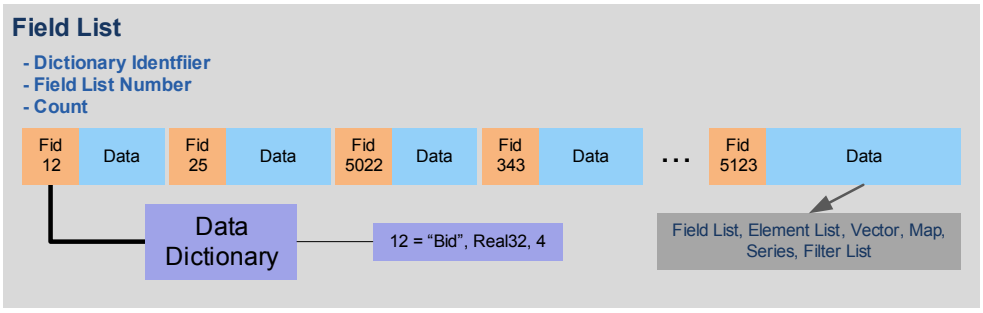

Figure 17. Field List

Multiple data dictionaries are supported and can be scoped (using namespaces) when needed. Dictionaries also have versions and can be downloaded by consumers from the attached access point. Field lists self-identify the required dictionary needed through the dictionary identifier parameter. A single field list can have fields that reference multiple data dictionaries (e.g. fields from a customer defined dictionary can be added to any vendor provided record).

A field list can be viewed as an element list with a compression technique applied. Instead of passing a full tag/name and data type in every message, an integer based field identifier is defined. A separate dictionary is needed to convert this Field IDentifier number into a tag and data type. The record-set concept (refer to Section 5.3.1.2) can be used when encoding a field list in order to even further reduce bandwidth requirements.

5.3.2.3 Vector

A ***vector*** defines a structure that contains highly manipulative position-oriented entries; each known as a vector entry. Each vector entry position is identified by an integer index value. The index starts at 0 and can go as high as 4,294,967,292. Entries in the vector can be set, updated or cleared and can optionally each have a separate permissions expression for even finer access control. A vector can also optionally support sorting operations (sorted vectors are identified in the response) such as insert and delete.

Vectors provide their largest benefit when combined with other data formats (e.g. vector of field lists, vector of vectors). The data format for each vector entry is identified once in the vector header (i.e. all vector entries must be the same data format). Vectors optionally contain summary data (refer to Section 5.3.1.4) for content that applies to the entire structure. Record-set definitions (refer to Section 5.3.1.2) can also optionally be defined for vector entries that contain repetitive record data.

Vector responses can be fragmented depending on the amount of data. When this occurs, the encoding application ensures that a single vector entry will not be broken across two different fragments. This allows the receiving application to process each fragment independently. An optional total count hint may be provided that indicates the total size of the vector across all fragments of the response (may be used for pre-allocation when caching).

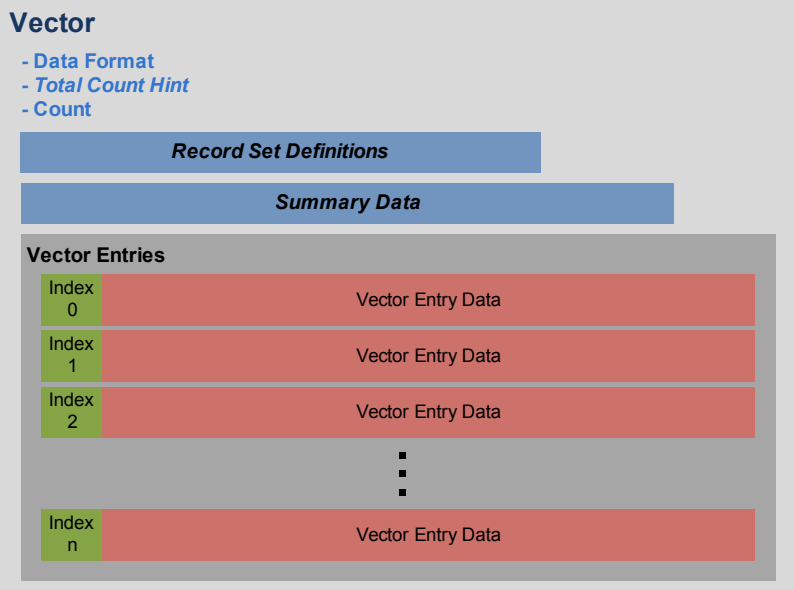

Figure 18. Vector

5.3.2.4 Map

A ***map*** defines a structure of highly manipulative, associative-referenced key-oriented entries; each known as a map entry. The Open Message Model map can be thought of as an STL map or a generic hash table. Each map entry is identified by a map key that can take the form of any basic data type (refer to Section 5.3.1.1). Examples could include map entries identified by an ASCII string, binary buffer or even a real number. Entries in the map can be added, updated or deleted and can optionally each have a separate permissions expression for even finer access control.[6]

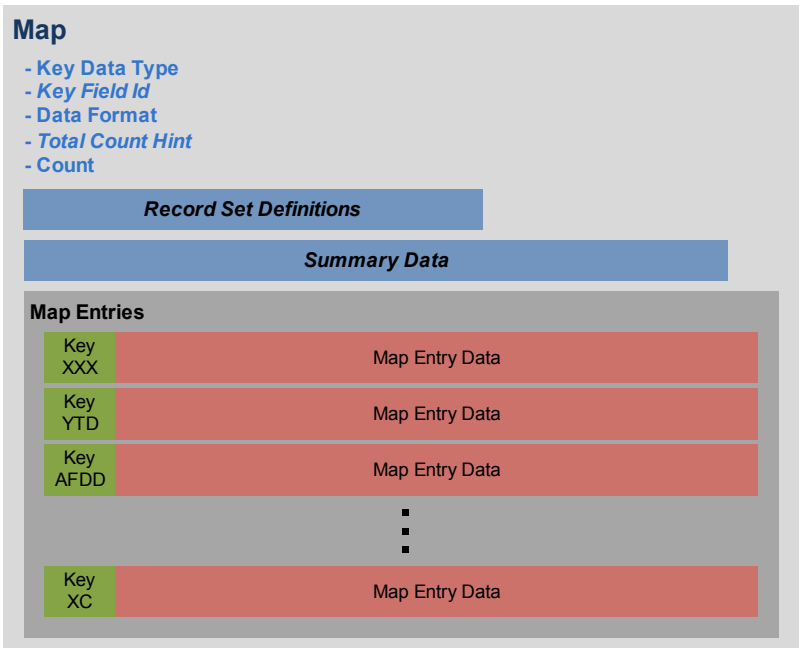

Figure 19. Map

Maps provide their largest benefit when combined with other data formats (e.g. map of field lists, map of vectors). The data format for each map entry is identified once in the map header (i.e. all map entries must be the same data format). Maps optionally contain summary data (refer to Section 5.3.1.4) for content that applies to the entire structure. Record-set definitions (refer to Section 5.3.1.2) can also optionally be defined when map entries contain repetitive record data.

Map responses can be fragmented depending on the amount of data. When this occurs the encoding application ensures that a map entry will not be broken across two different fragments. This allows the receiving application to process each fragment independently. An optional total count hint may be provided that indicates the total size of the map across all fragments of the response (may be used for pre-allocation when caching).

6. Map entry permission checks are currently not implemented.

5.3.2.5 Series

A ***series*** defines a structure of implicitly-indexed, accruable entries (each known as a series entry). A series is typically used to represent repetitively structured data. Series entries cannot be identified and typically have an implicit order (e.g. time, date). Operations on entries are not supported, since there is no way of entry identification.

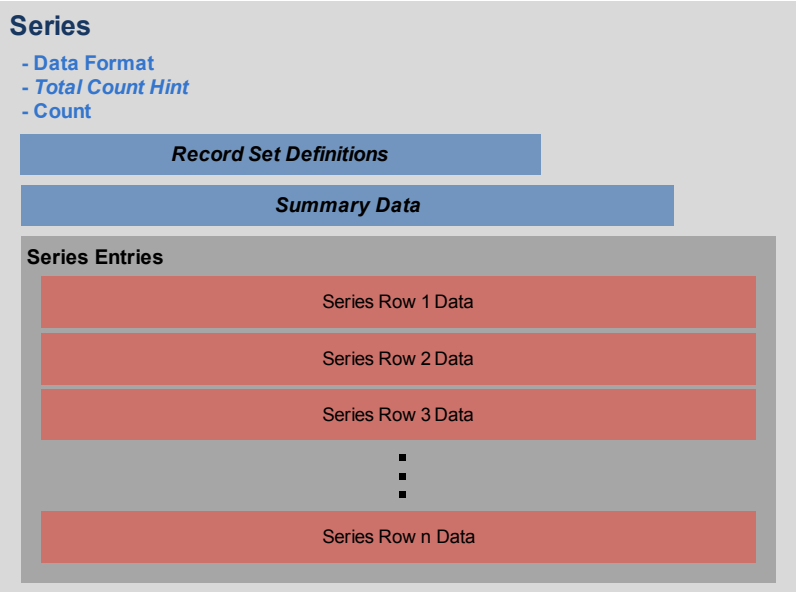

Figure 20. Series

Series provide their largest benefit when combined with other data formats (e.g. series of field lists, series of vectors). The data format for each series entry is identified once in the series header (i.e. all series entries must be the same data format). Series optionally contain summary data (refer to Section 5.3.1.4) for content that applies to the entire structure. Record-set definitions (refer to Section 5.3.1.2) can also optionally be defined when series entries contain repetitive record data.

Series responses can be fragmented depending on the amount of data. When this occurs the encoding application ensures that a series entry will not be broken across two different fragments. This allows the receiving application to process each fragment independently. An optional total count hint may be provided that indicates the total size of the series across all fragments of the response (may be used for pre-allocation when caching).

5.3.2.6 Filter List

A ***filter list*** defines a structure of loosely-coupled, associative-referenced entries; each known as a filter entry. Filter lists are defined by the service provider and are used to break up information into finer-grained entries (i.e. they are not meant for field level requests). Filter entries are identified by an 8-bit unsigned integer and can be requested by setting a bit in a bitmap (there are a maximum of 32 filter entries). Entries in the filter list can be set, updated or cleared and can optionally each have a separate permissions expression.

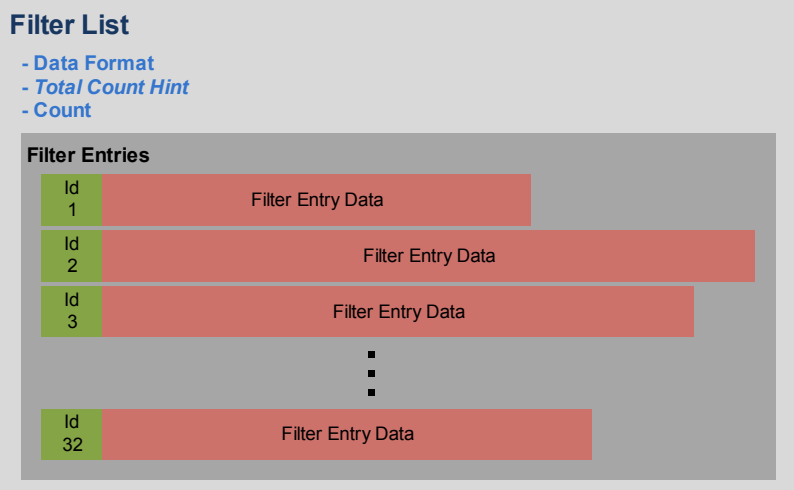

Figure 21. Filter List

Filter lists provide their largest benefit when combined with other data formats (e.g. filter list of element lists, filter list of maps). Even though the data format for each filter entry data is identified in the filter list header, filter entries can optionally define different data formats per entry.

Filter list responses can be fragmented depending on the amount of data. When this occurs the encoding application ensures that a filter entry will not be broken across two different fragments. This allows the receiving application to process each fragment independently. An optional total count hint may be provided that indicates the total size of the filter list across all fragments of the response (may be used for pre-allocation when caching).

6 Domain Message Models

6.1 Refinitiv Domain Models

Refinitiv has defined a set of domain models that will be used by all internal, and third-party, service providers and consumers. Clients that want to achieve maximum interoperability with these applications should utilize Refinitiv Domain Models. Each definition is broken into an item type model and a content definition model. Therefore, applications can still inter-operate as long as they conform to the item type models.

Defined, administrative domain models include:

- ***Login***: Login a user/application to a system access point (refer to Section 3.4).
- ***Directory***: Directory and detailed information about the services available to consumers.
- ***Dictionary***: Access to all required dictionaries (e.g. data dictionaries, enumerations files, etc.).

The defined domain models for instrument based market data include:

- ***Market-price***: Updating trades, quotes and inside top of book quotes (i.e. level 1 content).
- ***Market-by-order***: Updating instrument market information indexed by order (i.e. full order book: level 2 content).
- ***Market-by-price***: Updating top of book instrument market information indexed by best price (i.e. market depth: limited level 2 content).
- ***Market maker***: Updating market maker quotes and trade information from exchanges.
- ***Symbol lists***: Updating lists of symbols/instruments (e.g. .AV.O, S&P500, NASDAQ, etc.).
- ***Yield Curves***: The relation between the interest rate and term (time to maturity) associated with the debt of a borrower. The shape of the curve can help give an idea of future economic activity and interest rates.

The above list is not exhaustive and new domains are being defined. For the latest domain message models, API packages, and associated documentation sets, refer to the Refinitiv Developer Community portal.

6.2 Defining New Domains

Because the Open Message Model is an open model, you can extend an existing type or create your own domain models. You could define a yield curve, a volatility surface, a complex weather derivative contract, or a multi-asset portfolio as a few examples. You can define the types that your application needs.

New domain models are defined by specifying the item type model and the content definition model. To create a new type:

- Define the transport semantics including the interaction paradigm, the request key, the supported options, the request message, the response message, and other messages like the update, status, close, and ack.
- Design the data encoding using Open Message Model data abstractions. Define what, if any, optimizations are used including summary data and record-sets.
- Define an interaction scenario.
- Define the content definition model, including fields and data dictionaries specific to your provider's content.
- Publish and share documentation for developers in your organization or to developers that will interact with your provider of this content. Developers that need to consume this type will write Robust Foundation API code to handle this new type model.

If you find that you need assistance as you develop your first few domain models, Refinitiv can assist you with your design needs for Open Message Model domain models via training, consulting, or support. In some cases, especially if this is a model that will have wide applicability, an architect in the development organization may want to work directly with you. Contact Refinitiv through your account team to discuss the best way of gaining assistance. You can also interact directly with other developers via the Refinitiv Developer Community portal to discuss creating a domain model.

z

Document ID: OMM351WP.200
Date of issue: August 2020