We are now Refinitiv, formerly the Financial and Risk business of Thomson Reuters. We've set a bold course for the future – both ours and yours – and are introducing our new brand to the world.

As our brand migration will be gradual, you will see traces of our past through documentation, videos, and digital platforms.

Thank you for joining us on our brand journey.

REFINITIV™

Elektron API

ELEKTRON APIS CONCEPTS GUIDE

Document Version: 1.3
Date of issue: 28 April 2017
Document ID: API310UM.170

THOMSON REUTERS

Contents

List of Figures

List of Tables

Chapter 1 Guide Introduction

1.1 About this Manual

This document is authored by Elektron API architects and programmers who encountered and resolved many of the issues the reader might face. Several of its authors have designed, developed, and maintained Elektron API products and other Thomson Reuters products which leverage them.

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

1.2 Audience

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

While technically RFA is not an Elektron API, the content presented herein also accurately describes the structure and concepts of the RFA. For simplicity, whenever the manual refers to the Elektron APIs, RFA is also included in its scope. Additionally, UPA is part of the Elektron APIs and has been rebranded to ETA.

1.3 Programming Languages

This guide discusses concepts and architecture specific to the Elektron API suite. Any code examples in this document are either language-neutral or labeled according to the language used in the example. Example applications provided with a specific API product are written in the relevant product's language (i.e., C++, Java, etc.).

1.4 Acronyms and Abbreviations

ACRONYM	MEANING
ADH	Advanced Data Hub is the horizontally scalable service component within Thomson Reuters Enterprise Platform (TREP) providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	Advanced Distribution Server is the horizontally scalable distribution component within Thomson Reuters Enterprise Platform (TREP) 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

Table 1: Acronyms and Abbreviations

ACRONYM	MEANING
ATS	Advanced Transformation System
DACS	Data Access Control System
DMM	Domain Message Model
EED	Elektron Edge Device
EMA	Elektron Message API, referred to simply as the Message API
EOA	Elektron Object API, referred to simply as the Object API.
ETA	Elektron Transport API, referred to simply as the Transport API. Formerly referred to as UPA.
EWA	Elektron Web API
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
IDN	Integrated Data Network
NIP	Non-Interactive Provider
OMM	Open Message Model
QoS	Quality of Service
RDM	Reuters Domain Model
Reactor	The Reactor is a low-level, open-source, easy-to-use layer above ETA. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users.
RFA	Robust Foundation API
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format, a Thomson Reuters proprietary format.
SOA	Service Oriented Architecture
SSL	Source Sink Library
TR-DFD	Thomson Reuters Data Feed Direct
TREP	Thomson Reuters Enterprise Platform
UML	Unified Modeling Language
UPA	Ultra Performance API is the a low-level API, currently used by the Thomson Reuters Enterprise Platform (and its dependent APIs) for optimized distribution of OMM/RWF data. UPA has been rebranded as ETA and is part of the Elektron SDK.
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 References

1. Elektron API-Specific *RDM Usage Guides*
2. Elektron API-Specific *ANSI Library Reference Manuals*
3. Elektron API-Specific *DACS LOCK Library Reference Manuals*
4. *Transport API Edition Value Added Components Developers Guide*
5. *Transport API Developers Guide* specific to the API and programming language you use.
6. The Thomson Reuters Professional Developer Community

1.6 Documentation Feedback

While we make every effort to ensure the documentation is accurate and up-to-date, if you notice any errors, or would like to see more details on a particular topic, you have the following options:

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

1.7 Document Conventions

This document uses the following types of conventions:

- Typographic
- Diagrams

1.7.1 Typographic

- Structures, methods, in-line code snippets, and types are shown in `orange, Courier New` font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.

1.7.2 Diagrams

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

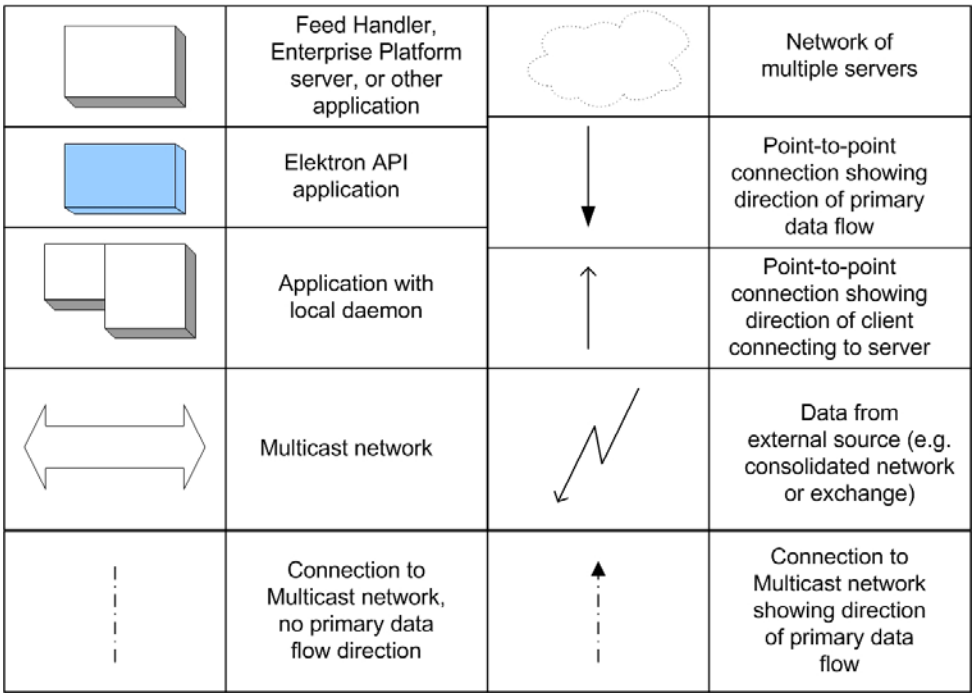

Figure 1. Network Diagram Notation

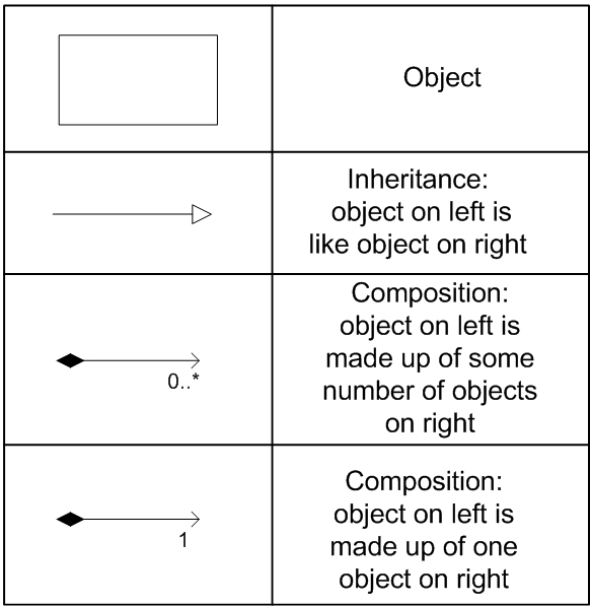

Figure 2. UML Diagram Notation

Chapter 2 Product Description

2.1 What is an Elektron API?

The Elektron API suite of products offers you flexibility in selecting the API best suited to your TREP needs. Whether you need to achieve the highest throughput possible, realize the lowest latency, or rapidly build applications that allow easy access to content, the Elektron APIs offer you the broadest range of capabilities to make it possible.

The Elektron APIs are currently used by products such as the Advanced Distribution Server (ADS), Advanced Data Hub (ADH), EDF-D, Elektron, and Eikon. Certain TREP products (such as ADS, ADH, EDF, and Elektron) even use one of the Elektron APIs (the Transport API) as a foundation.

Elektron APIs support all constructs available as part of the Open Message Model. Using an Elektron API, customers can choose between an easy-to-use session-level API (i.e., the Message API) and a high-performance transport-level API (i.e., the Transport API).

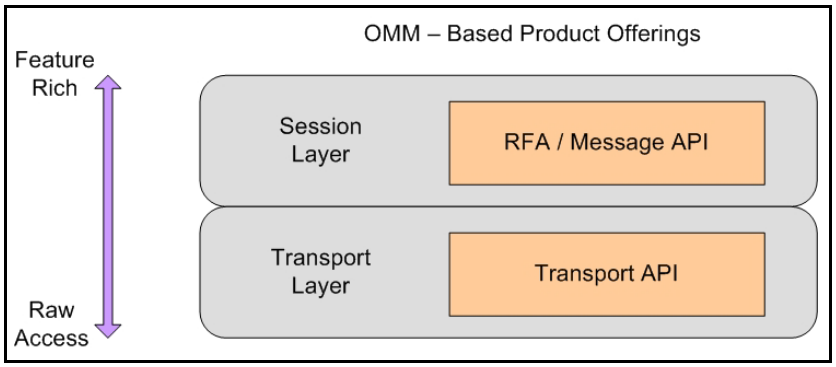

Figure 3. OMM-Based Product Offerings

The Elektron APIs provide application developers with the most flexible development environment and are the foundation on which all Thomson Reuters OMM-based components are built.

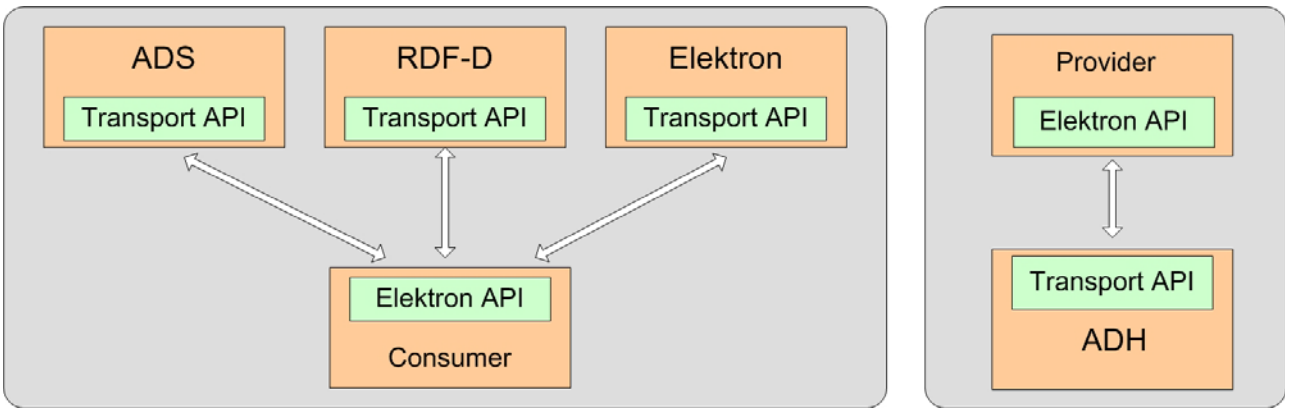

Figure 4. Elektron API: Core Diagram

2.2 API Features

The Elektron APIs are:

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

2.2.1 General Capabilities

Elektron APIs provide general capabilities independent of the type of application. The Elektron APIs:

- Supports fully connected or unified network topologies as well as segmented topologies.
- Supports multiple network session types, including TCP, HTTP, and multicast-based networks.
- Can internally fragment and reassemble large messages.
- Can pack multiple, small messages into the same network buffer.
- Can perform data compression and decompression internally.

2.2.2 Consumer Applications

You can use the Elektron APIs to create consumer-based applications that can:

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

2.2.3 Provider Applications: Interactive

You can use the Elektron APIs to create interactive providers that can:

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

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

2.2.4 Provider Applications: Non-Interactive

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

2.3 Performance and Feature Comparison

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

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

Tip: The Elektron API now ship with API performance tools and additional documentation to which you can refer which you can use to arrive at more-specific results for your environment.

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

API	THREAD SAFETY	THROUGHPUT	LATENCY	MEMORY FOOTPRINT
Transport API	Safe and Aware	Very High	Lowest	Lowest
Message API	Safe and Aware	High	Low	Medium (watch list[a])
Reactor[b]	Safe and Aware	Very High	Low	Medium (watch list optional)
RFA	Safe and Aware	High	Low	Medium (watch list, allows optional queues)
SFC C++	None	Medium	High	Medium – High (watch list, cache)

Table 2: API Performance Comparison

a. The Elektron Message API leverages the reactor watchlist.
b. The Reactor is an ease-of-use layer provided with the Elektron Transport API.

2.4 Functionality: Which API to Choose?

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

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

The following table lists API capabilities using the following legend:

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

CAPABILITY TYPE	CAPABILITY	TRANSPORT API 3.X	ETA REACTOR[A]	MESSAGE API 3.X[B]	OBJECT API 3.0	ELEKTRON WEB API 1.7	RFA 8.0
Transport	Compression via OMM	X	X**	X**	Future		X
	HTTP Tunneling (RWF)	X	X**	X**	Future		X
	TCP/IP: RWF	X	X**	X**	Future		X
	Reliable Multicast: RWF	X	X**	X**	Future		X
	Sequenced Multicast	X					
	Websocket					X	
	Unidirectional Shared Memory	X					
Application Type	Consumer	X	X	X**	Future	X	X
	Provider: Interactive	X	X	X**	Future		X
	Provider: Non-Interactive	X	X	X**	Future		X

Table 3: Capabilities by API

CAPABILITY TYPE	CAPABILITY	TRANSPORT API 3.X	ETA REACTOR[A]	MESSAGE API 3.X[B]	OBJECT API 3.0	ELEKTRON WEB API 1.7	RFA 8.0
General	Batch Request	X	X	X	Future		X
	Batch Re-issue and Close	X	X				X
	Generic Messages	X	X	X	Future		X
	Pause/Resume	X	X	X	Future		X
	Posting	X	X	X	Future		X
	Snapshot Requests	X	X	X	Future	X	X
	Streaming Requests	X	X	X	Future	X	X
	Private Streams	X	X	X	Future	Future	X
	Qualified Streams	X	X	X	Future	Future	
	Views	X	X	X	Future		X
Domain Models	Custom Data Model Support	X	X	X	Future		X
	RDM: Dictionary	X	X	X	Future		X
	RDM: Enhanced Symbol List	X	X	X**			X
	RDM: Login	X	X	X	Future		X
	RDM: Market Price	X	X	X	Future	X	X
	RDM: MarketByOrder	X	X	X	Future	Future	X
	RDM: MarketByPrice	X	X	X	Future	Future	X
	RDM: Market Maker	X	X	X	Future	Future	X
	RDM: Service Directory	X	X	X	Future	Future	X
	RDM: Symbol List	X	X	X	Future	Future	X
	RDM: Yield Curve	X	X	X	Future	Future	X

Table 3: Capabilities by API (Continued)

CAPABILITY TYPE	CAPABILITY	TRANSPORT API 3.X	ETA REACTOR[A]	MESSAGE API 3.X[B]	OBJECT API 3.0	ELEKTRON WEB API 1.7	RFA 8.0
Encoders/Decoders	AnsiPage	X	X**	X**			Legacy
	DACS Lock	X	X**	X**	Future		X
	OMM	X	X	X**	Future	Future	X
	RMTES	X	X	X**	Future	Future	X
Layer Specific	Config: file-based			X	Future		X
	Config: programmatic	X	X	X	Future		X
	Group fanout to items		X	X**	Future		X
	Load balancing: API-based						X
	Logging: file-based			X	Future		X
	Logging: programmatic	X	X	Future	Future		X
	QoS Matching		X	X**	Future		X
	Network Pings: automatic		X	X**	Future		X
	Recovery: connection		X	X**	Future		X
	Recovery: items		X	X**	Future		X
	Request routing		X	X**	Future		X
	Session management		X	X	Future		X
	Service Groups						X
	Single Open: API-based		X	X**	Future		X
	Warm Standby: API-based						X
	Watchlist		X	X**	Future		X
	Controlled fragmentation and assembly of large messages	X	X**	X**			
	Controlled locking / threading model	X					

Table 3: Capabilities by API (Continued)

CAPABILITY TYPE	CAPABILITY	TRANSPORT API 3.X	ETA REACTOR[A]	MESSAGE API 3.X[B]	OBJECT API 3.0	ELEKTRON WEB API 1.7	RFA 8.0
Layer Specific (*Continued*)	Controlled dynamic message buffers with ability to programmatically modify during runtime	X	X**				
	Controlled message packing	X	X**				
	Messages can be written at different priority levels	X	X**				

Table 3: Capabilities by API (Continued)

a. The Reactor is an open source component that functions within the ETA.
b. Not yet released. This version number is tentative and subject to change.

2.5 API Models

2.5.1 Open Message Model (OMM)

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

The layout and interpretation of any specific OMM model (also referred to as a domain model) is described within that model's definition and is not coupled with the API. The OMM is a flexible and simple tool that provides the building blocks to design and produce domain models to meet the needs of the system and its users. The Elektron API provide structural representations of OMM constructs and manages the RWF binary-encoded representation of the OMM. Users can leverage Thomson Reuters-provided OMM constructs to consume or provide OMM data throughout the Enterprise Platform.

2.5.2 Reuters Wire Format (RWF)

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

2.5.3 Domain Message Model

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

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

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

2.5.3.1 Reuters Domain Model

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

2.5.3.2 User-Defined Domain Model

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

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

Chapter 3 Consumers and Providers

3.1 Overview

For those familiar with Thomson Reuters's API products or concepts from TREP, Rendezvous, or Triarch, we map how the Elektron API implement the same functionality.

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

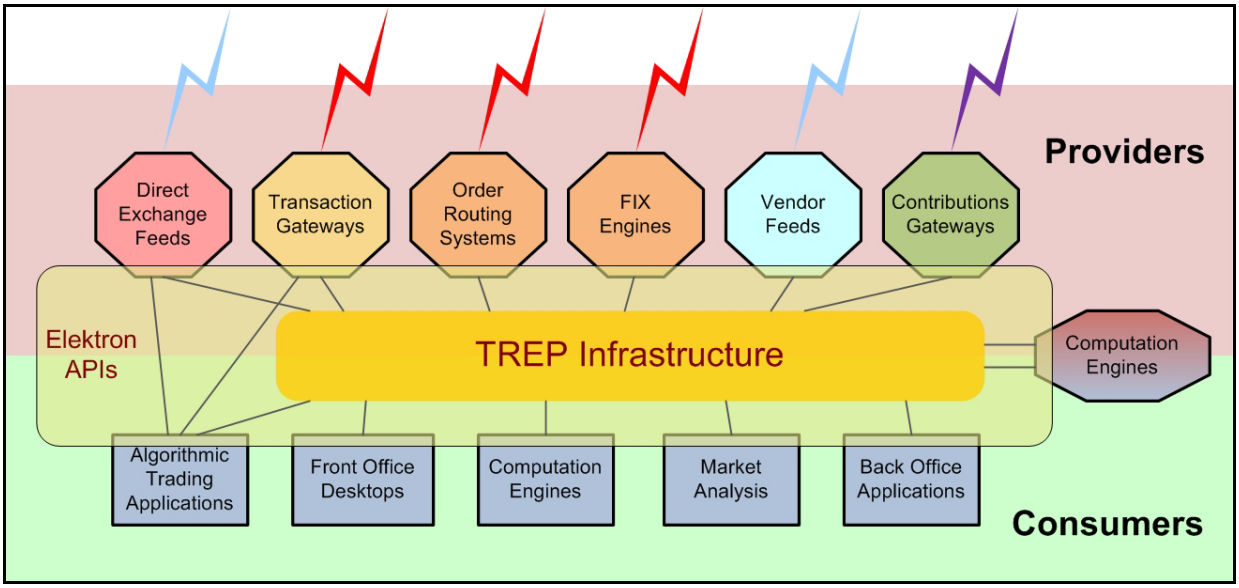

Figure 5. TREP Infrastructure

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

3.2 Consumers

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

- A ***concrete access point***. A concrete access point is implemented by the service-provider application if it supports direct connections from consumers. The right-side diagram in Figure 6 illustrates an API consumer connecting to Elektron via a direct access point.
- A ***proxy access point***. A proxy access point is point-to-point based or multicast (according to your needs) and implemented by a TREP Infrastructure component (i.e., an ADS). Figure 6 also illustrates an API consumer connecting to the provider by first passing through a proxy access point.

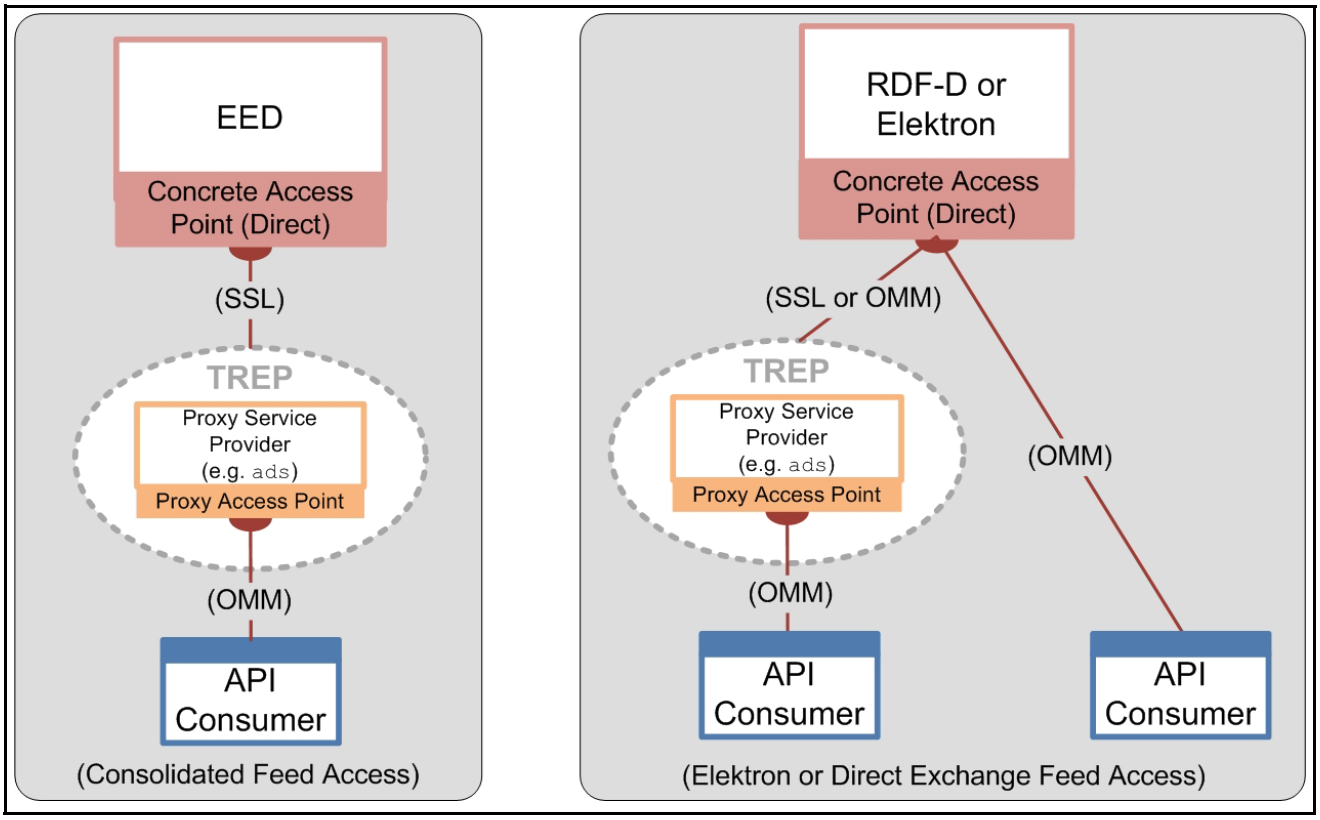

Figure 6. Elektron API as Consumers

Examples of consumers include:

- An application that subscribes to data via TREP, EDF, or Elektron.
- An application that posts data to TREP or Elektron (e.g., contributions/inserts or local ublication into a cache).
- An application that communicates via generic messages with TREP or Elektron.
- An application that does any of the above via a private stream.

3.2.1 Subscriptions: Request/Response

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

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

Specialized cases of request / response include:

- Batches
- Views
- Symbol Lists
- Server Symbol Lists

3.2.2 Batches

A consumer can request multiple items using a single, client-based, request called a ***batch*** request. After the consumer sends an optimized batch request to the ADS, the ADS responds by sending the items as if they were opened individually so the items can be managed individually.

Figure 7 illustrates a consumer issuing a batch request for "TRI, "GE", and "INTC.O" and the resulting ADS responses.

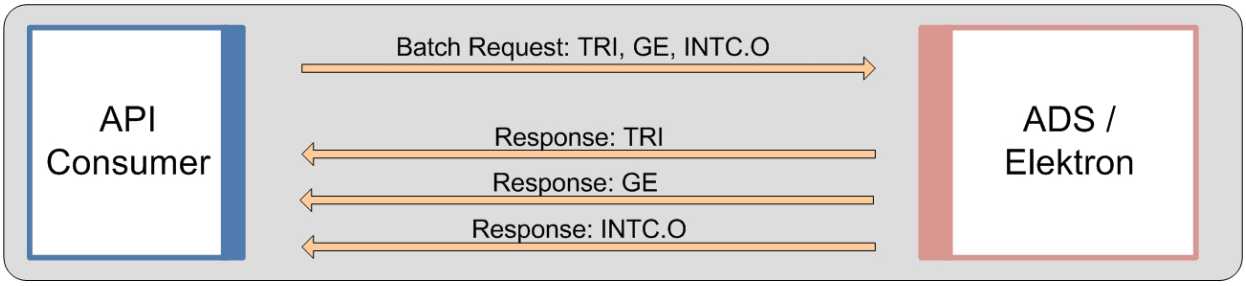

Figure 7. Batch Request

3.2.3 Views

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

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

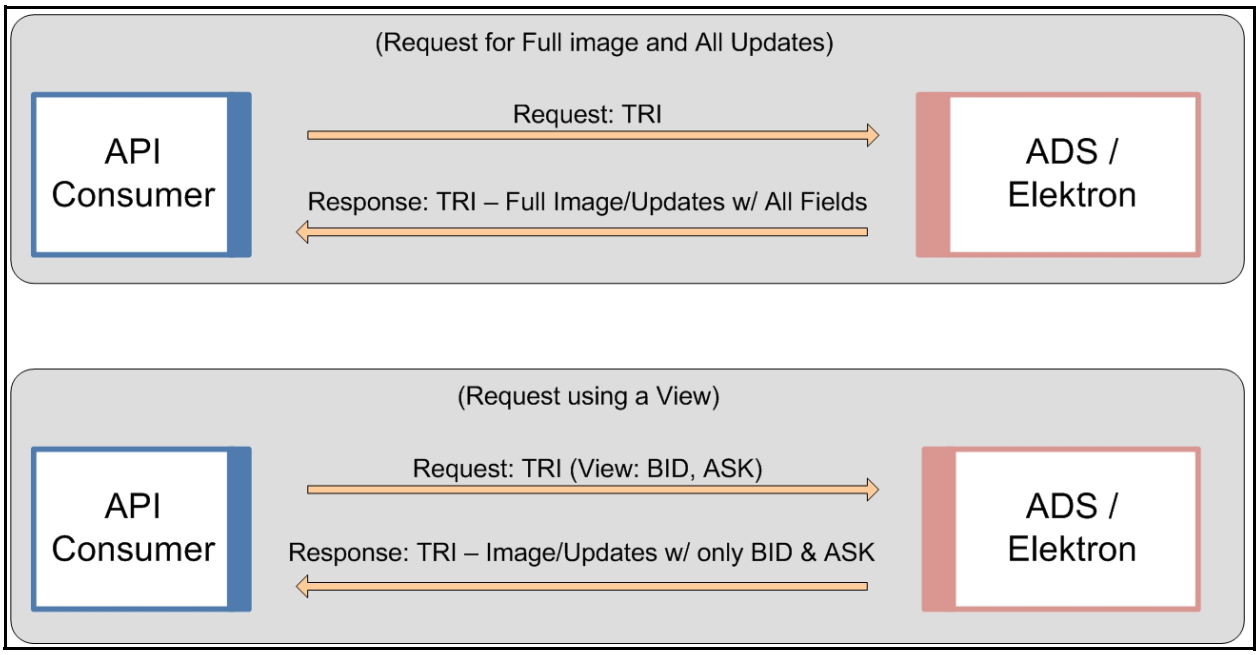

Figure 8. View Request Diagram

Views were designed to provide the same filtering functionality as the Legacy STIC device and SFC (based on its own internal cache) while optimizing network traffic.

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

3.2.4 Pause and Resume

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

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

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

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

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

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

In this case, the client application:

- Sends a single PAUSE message to the ADS to pause all the items it has open.
- Performs all needed back-end processing.
- Sends a Resume request to resume all the items it had paused.

After receiving the Resume request, the ADS sends a refresh (i.e., current image), to the client for all paused items and then continues to send any subsequent messages.

3.2.4.2 Pause / Resume Use Case 2: Display Applications

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

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

When Window 2 is again visible, the ADS sends a refresh, or current image, to the client for the item "INTC.O" and then continues to send any subsequent messages.

3.2.5 Symbol Lists

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

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

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

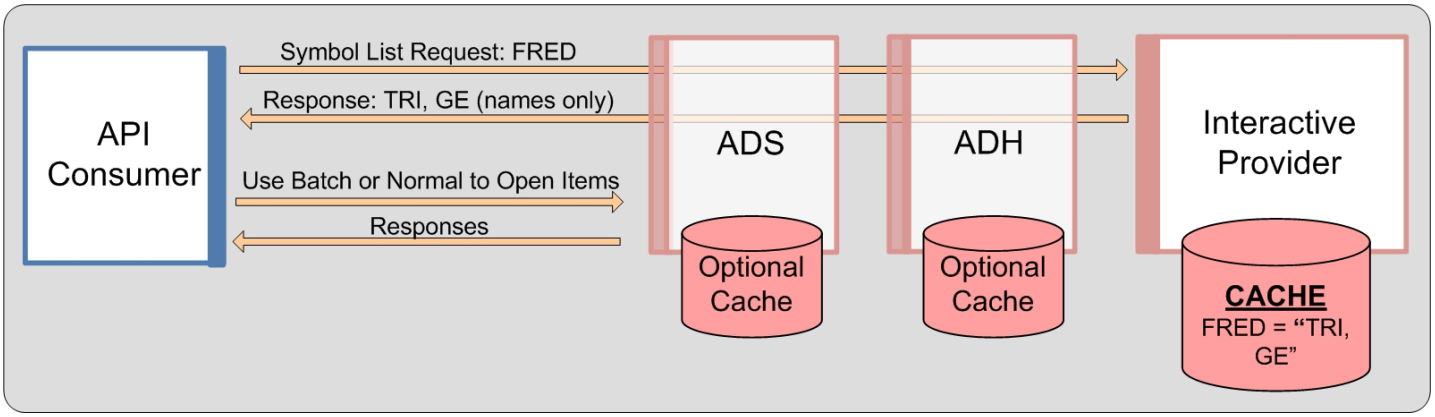

Figure 9. Symbol List: Basic Scenario

The following diagram illustrates how a consumer can access all items in the ADS Cache, effectively dumping the cache to the OMM client. In this scenario, the client requests the symbol list **_ADS_CACHE_LIST**. The ADS receives the request and responds with the names of all items in its cache. The client can then choose to open items individually, or make a batch request to open multiple items. The ADS provides an additional symbol list (**_SERVER_LIST**) for obtaining lists of items stored in specific ADH instances.

- For details on this symbol list, refer to the *ADS* and *ADH Software Installation Manuals*.
- For more detailed information on using symbol lists, refer to the developer's manual specific to the API you use.

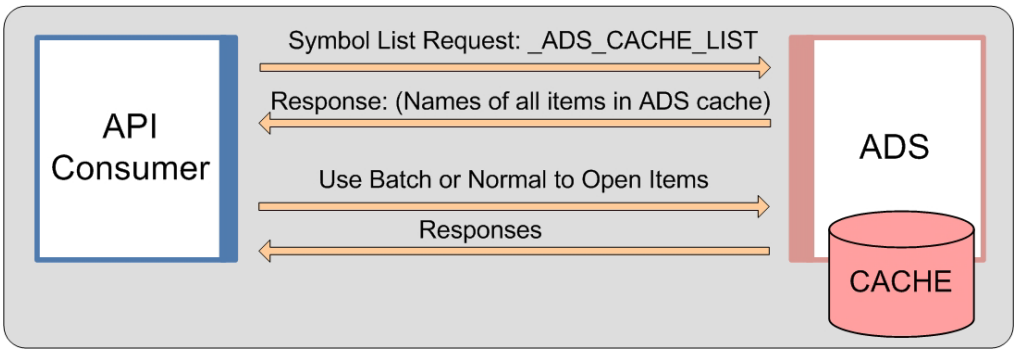

Figure 10. Symbol List: Accessing the Entire ADS Cache

3.2.6 Posting

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

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

3.2.6.1 Local Publication

The following diagram illustrates the benefits of posting.

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

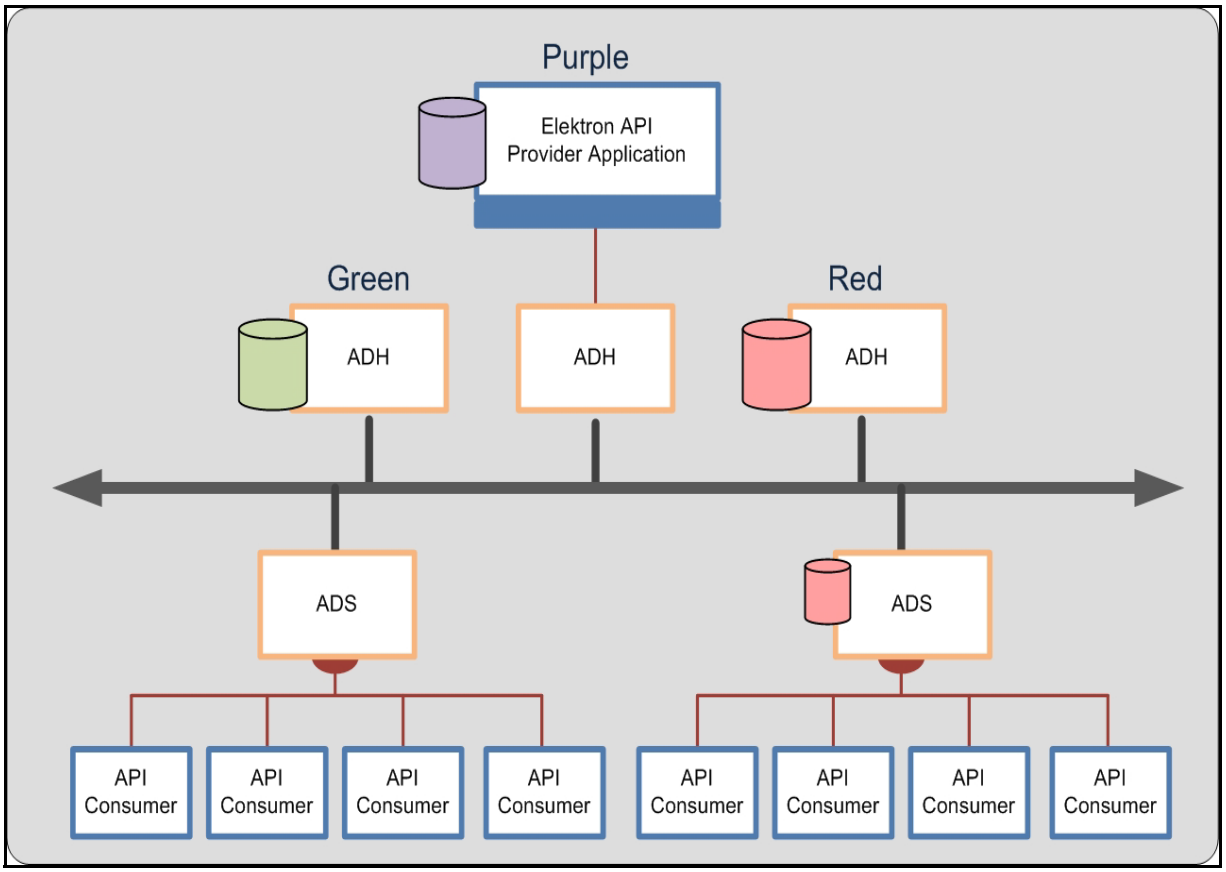

Figure 11. Posting into a Cache

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

3.2.6.2 Contribution/Inserts

Posting also allows OMM-based contributions. Through such posting, clients can contribute data to a device on the head end or to a custom-provider. In the following example, the Elektron API send an OMM post to a provider application that supports such functionality.

While this diagram is similar to the example in Figure 11, the difference is that core components (such as the ADS/ADH) in TREP can convert a post into an SSL Insert for legacy connectivity. This functionality is provided for migration purposes.

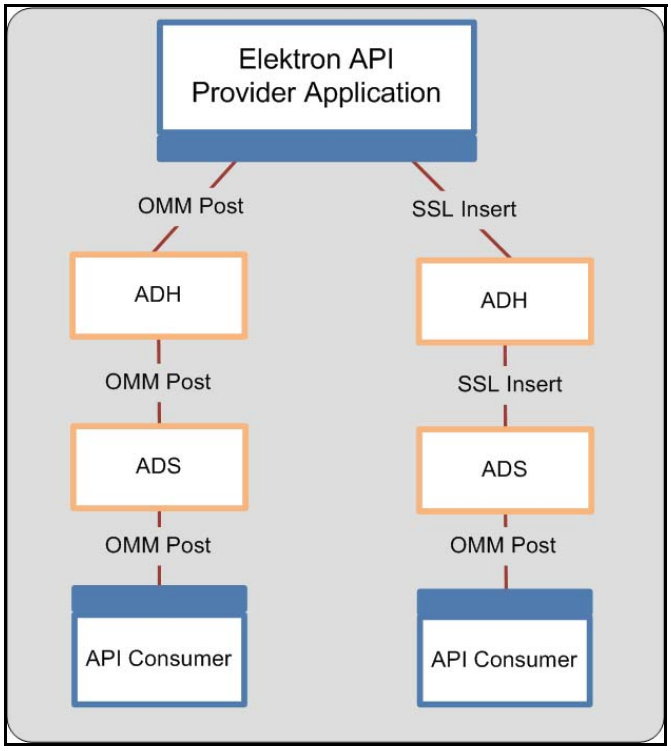

Figure 12. OMM Post with Legacy Inserts

3.2.7 Generic Message

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

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

3.2.8 Private Streams

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

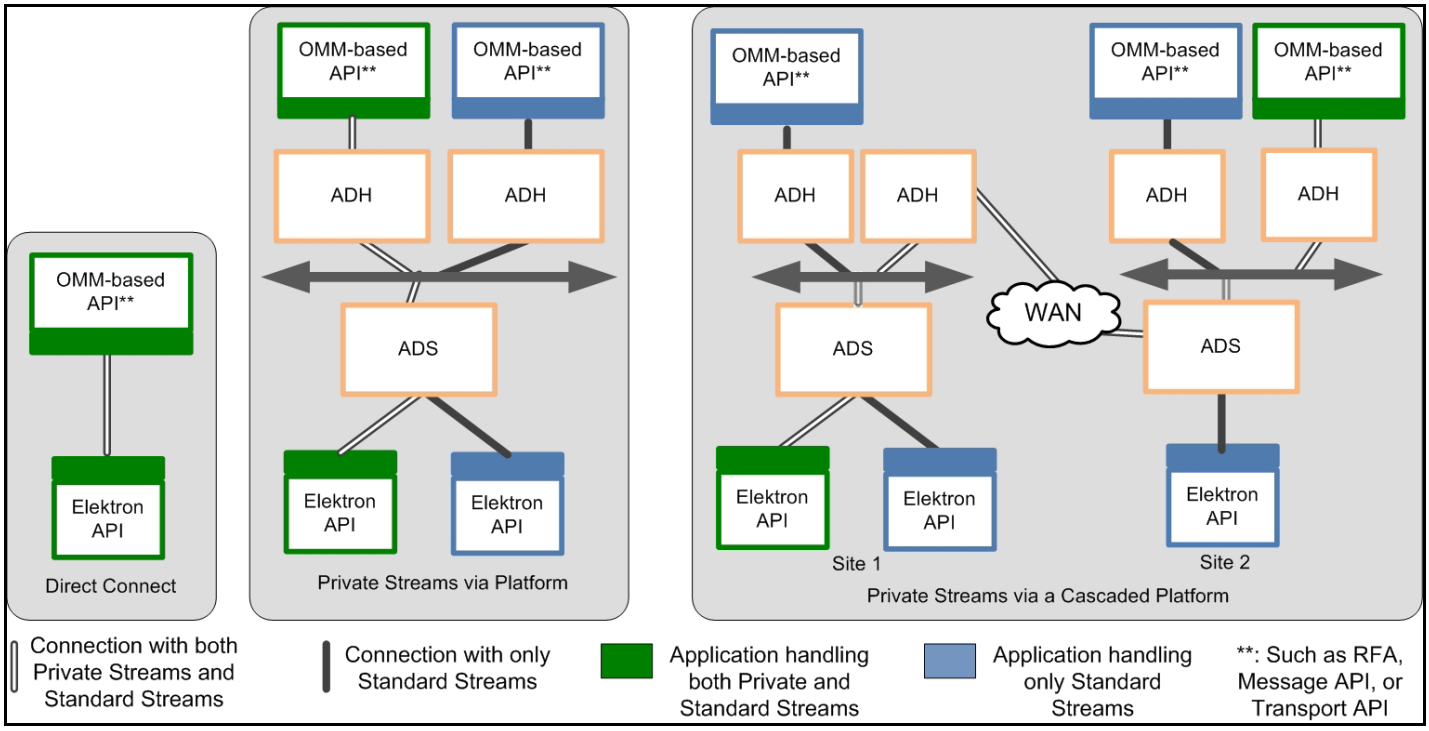

Figure 13. Private Stream Scenarios

A virtual private connection piggy backs on existing, individual point-to-point and multicast connections in the system (Figure 13 illustrates this behavior using a white connector). Messages exchanged via a Private Stream flow between a Consumer and an Interactive Provider using these existing underlying connections. However, unlike a regular stream, the Elektron API or TREP components do not fan out these messages to other consumers or providers.

In Figure 13, each diagram shows a green consumer creating a private stream with a green provider. The private stream, using existing infrastructure and network connections, is illustrated as a white path in each of the diagrams. When established, communications sent on a private stream flow only between the green consumer and the green provider to which it connects. Blue providers and consumers do not see messages sent via the private stream.

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

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

3.2.9 Tunnel Streams (Only Available in the ETA Reactor and in EMA)

The Reactor allows users to create and use special tunnel streams. A tunnel stream is a private stream with additional behaviors, such as end-to-end line of sight for authentication and guaranteed delivery. Tunnel streams are founded on the private streams concept, and the Transport API establishes them between consumer and provider endpoints (passing through any intermediate components, such as TREP or EED).

When creating a tunnel, the consumer indicates any additional behaviors to enforce, which is exchanged with the provider application end point. The provider end-point acknowledges creation of the stream as well as the behaviors that it will enforce on the stream. After the stream is established, the consumer can exchange any content it wants, though the tunnel stream will enforce behaviors on the transmitted content as negotiated with the provider.

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

In the following diagram, the orange cylinder represents a tunnel stream that connects the consumer application to the provider application. Notice that the tunnel stream passes directly through intermediate components: the tunnel stream has end-to-end line of sight so that the provider and consumer effectively talk to one another directly, though they traverse multiple devices in the system. Each black line flowing through the cylinder represents a different substream, where each substream transmits its own independent stream of information. Each substream could communicate different market content; for example one could be a Time Series request while another could be a request for Market Price content. A substream can also connect to a special provider application called a Queue Provider. A Queue Provider allows for persistence of content exchanged over the tunnel stream and substream, and helps provide content beyond the end-point visible to the consumer. For further details, refer to the *ETA Value Added Developers Guide* specific to the version of API that you use.

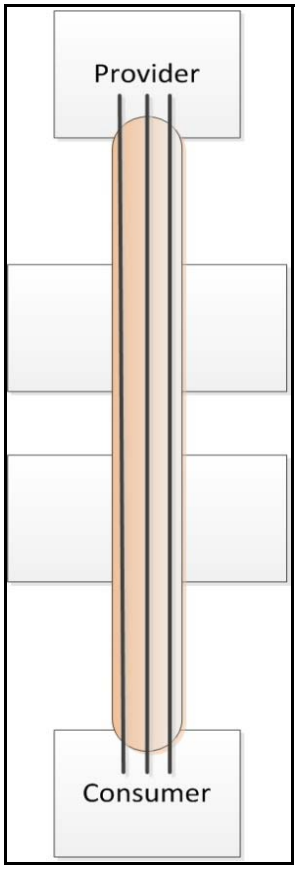

Figure 14. Tunnel Stream Illustration

3.2.10 Building an API Consumer

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

The general process can be summarized by the following steps[1]:

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

The example application included with each Elektron API product, provides an example implementation of an OMM consumer application. The application is written with simplicity in mind and demonstrates various aspects and features relevant to the API you use. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

1. Specific APIs might automatically rely on defaults unless overridden by the user.

3.3 Providers

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

Examples of providers include:

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

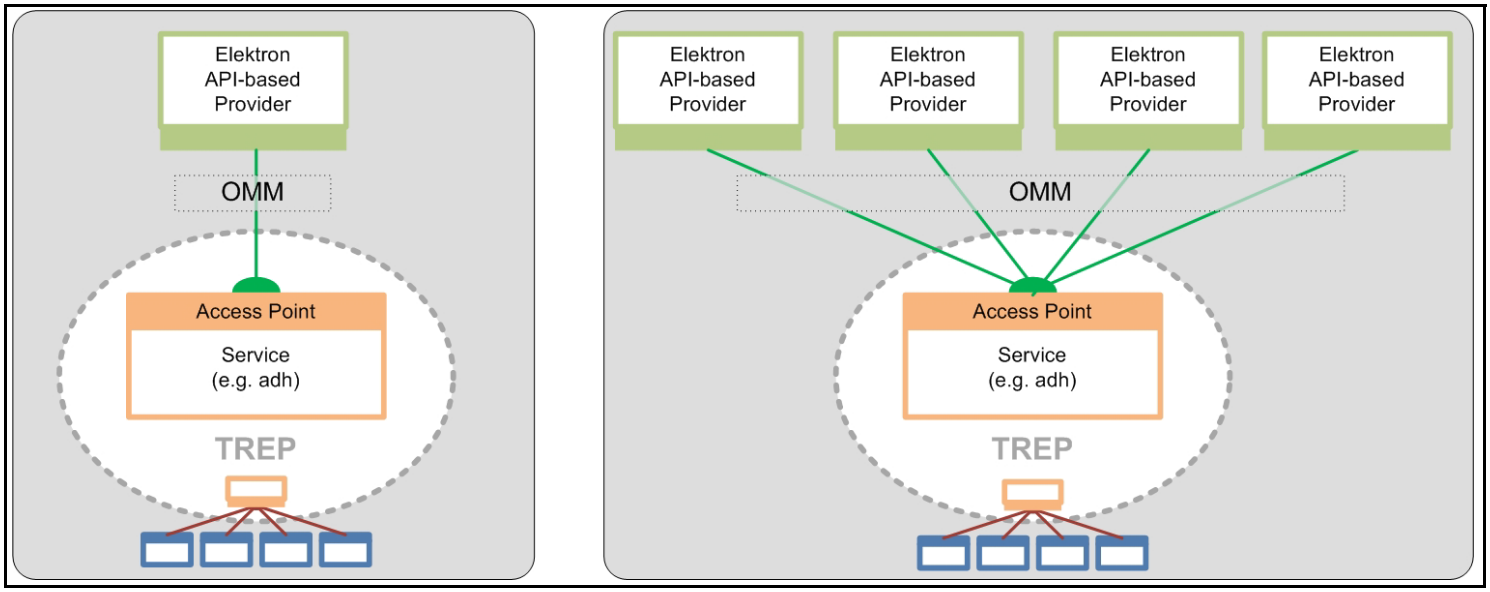

Figure 15. Provider Access Point

3.3.1 Interactive Providers

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

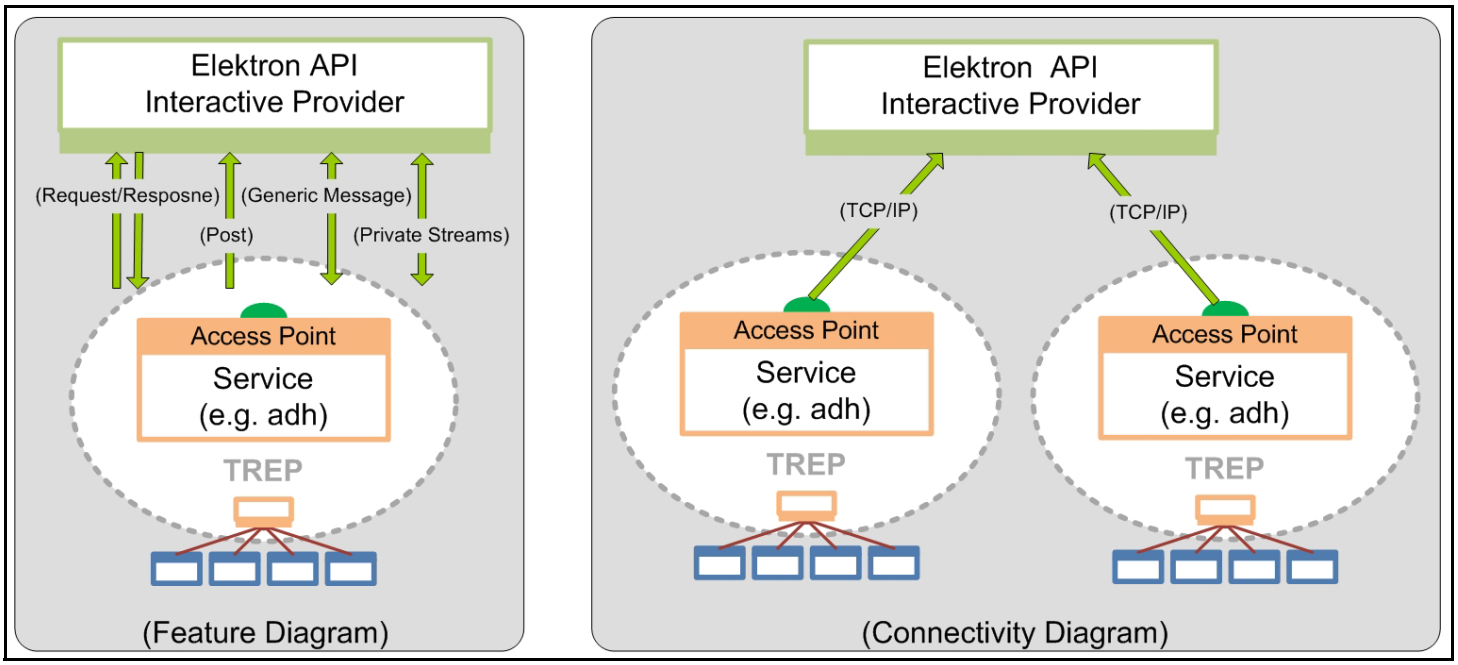

Figure 16. Interactive Providers

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

For legacy Triarch users or early TREP adopters, the Interactive Provider is similar in concept to the legacy Sink-Driven Server or Managed Server Application. Interactive Providers act like servers in a client-server relationship. An interactive provider can accept and manage connections from multiple TREP components.

3.3.1.1 Request /Response

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

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

3.3.1.2 Posts

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

3.3.1.3 Generic Messages

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

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

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

3.3.1.4 Private Streams

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

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

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

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

3.3.1.6 Building an Interactive Provider

An OMM interactive provider application opens a listening socket on a well-known port allowing OMM consumer applications to connect. After connecting, consumers can request data from the interactive provider.

The following steps summarize this process[2]:

- Establish network communication
- Accept incoming connections
- Handle login requests
- Provide source directory information
- Provide or download necessary dictionaries
- Handle requests and post messages
- Disconnect consumers and shut down

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

2. Specific APIs might automatically rely on defaults unless overridden by the user.

3.3.2 Non-Interactive Providers

3.3.2.1 Overview

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

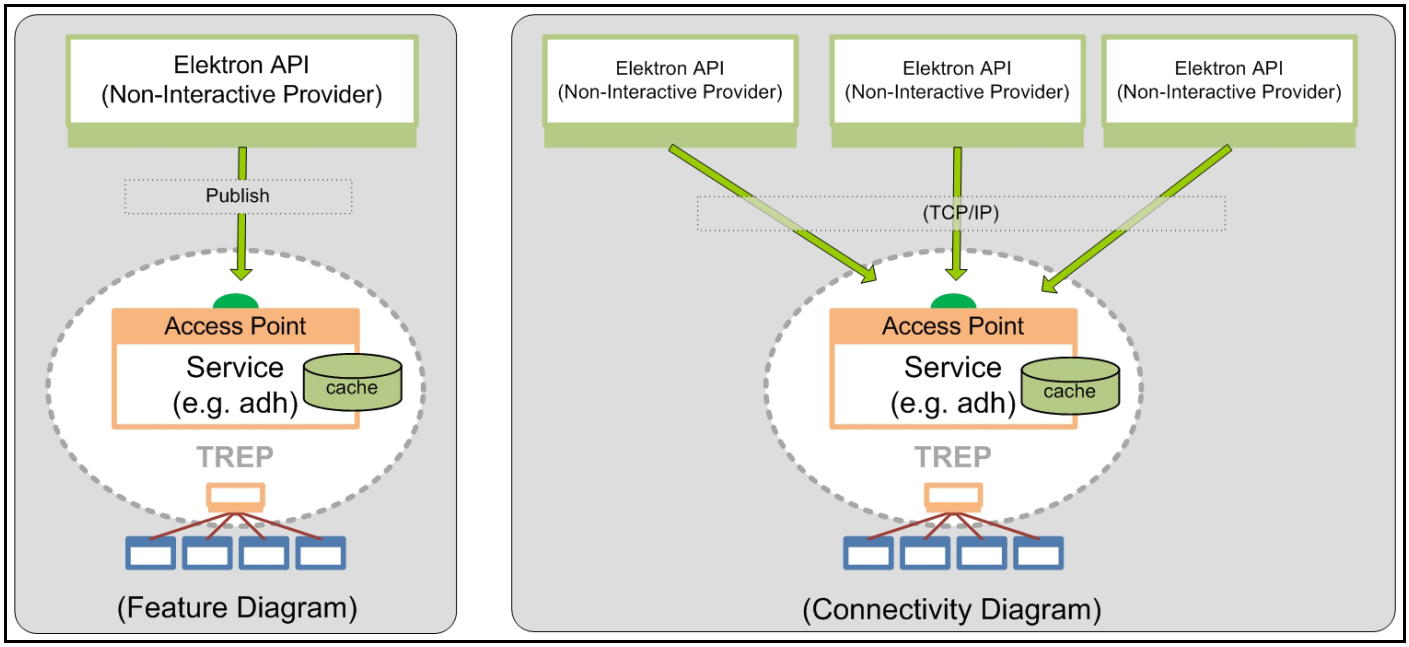

Figure 17. NIP: Point-To-Point

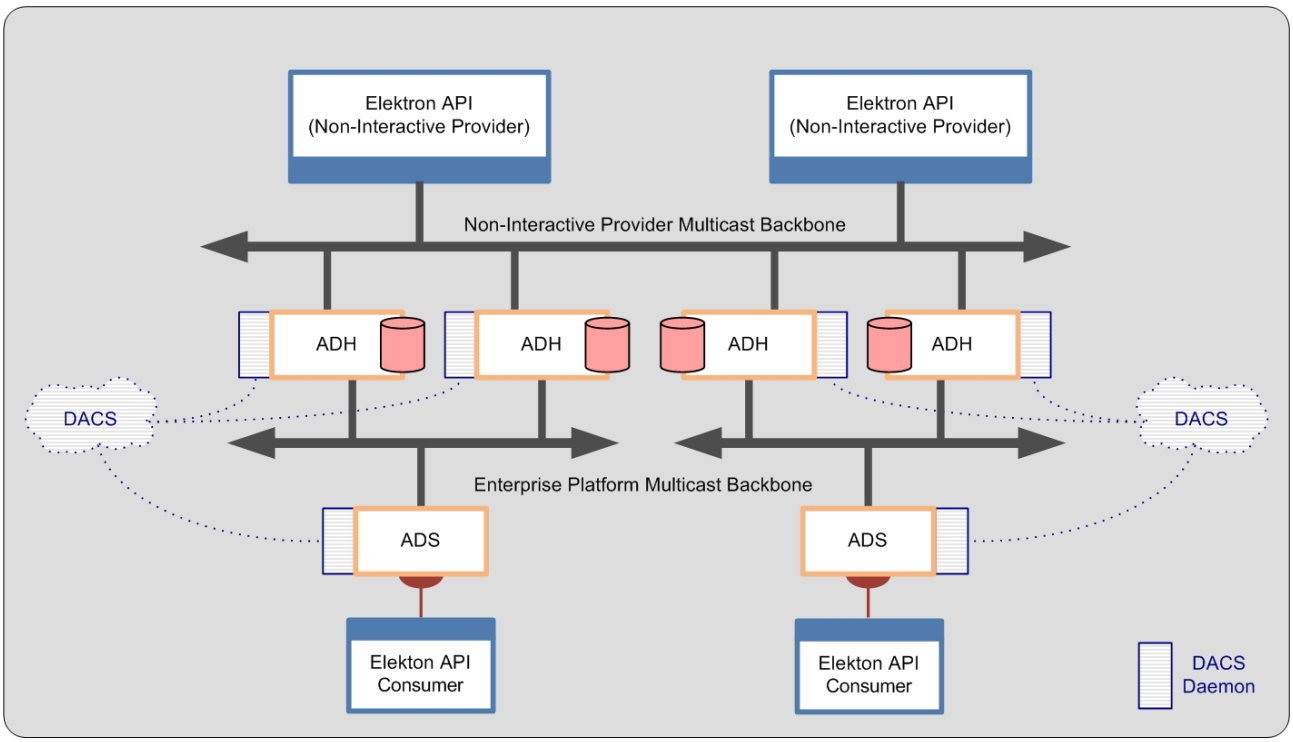

Figure 18. NIP: Multicast

After a NIP connects to TREP, the NIP can start sending information for any supported item and domain. For legacy Triarch users or early TREP adopters, the NIP is similar in concept to what was once called the Src-Driven, or Broadcast Server Application.

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

NIP applications can connect using a point-to-point TCP-based transport as shown in Figure 17, or using a multicast transport as shown in Figure 18.

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

3.3.2.2 Building a Non-Interactive Provider

An OMM NIP can publish information into the ADH cache without needing to handle requests for the information. The ADH can cache the information and along with other Enterprise Platform components, provide the information to any OMM consumer applications that indicate interest.

The general process can be summarized by the following steps:[3]

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

Included with the Elektron API package, the NIP example application provides an implementation of an NIP written with simplicity in mind and demonstrates the use of the appropriate Elektron API. Portions of the functionality are abstracted for easy reuse, though you might need to modify it to achieve your own performance and functionality goals.

Content is encoded and decoded depending on the API that you use.

3. Specific APIs might automatically rely on defaults unless overridden by the user.

Chapter 4 System View

4.1 System Architecture Overview

A TREP network typically hosts the following:

- Core Infrastructure (i.e., ADS, ADH, etc.)
- Consumer applications that typically request and receive information from the network
- Provider applications that typically write information to the network. Provider applications fall into one of two categories:
  - Interactive provider applications which receive and interpret request messages and reply back with any needed information.
  - NIP applications which publish data, regardless of user requests or which applications consume the data.
- Permissioning infrastructure (i.e., DACS)
- Devices which interact with the markets (i.e., Data Feed Direct and the Elektron Edge Device)

The following figure illustrates a typical deployment of a TREP network and some of its possible components. The remainder of this chapter briefly describes the components pictured in the diagram and explains how the Elektron API integrate with each.

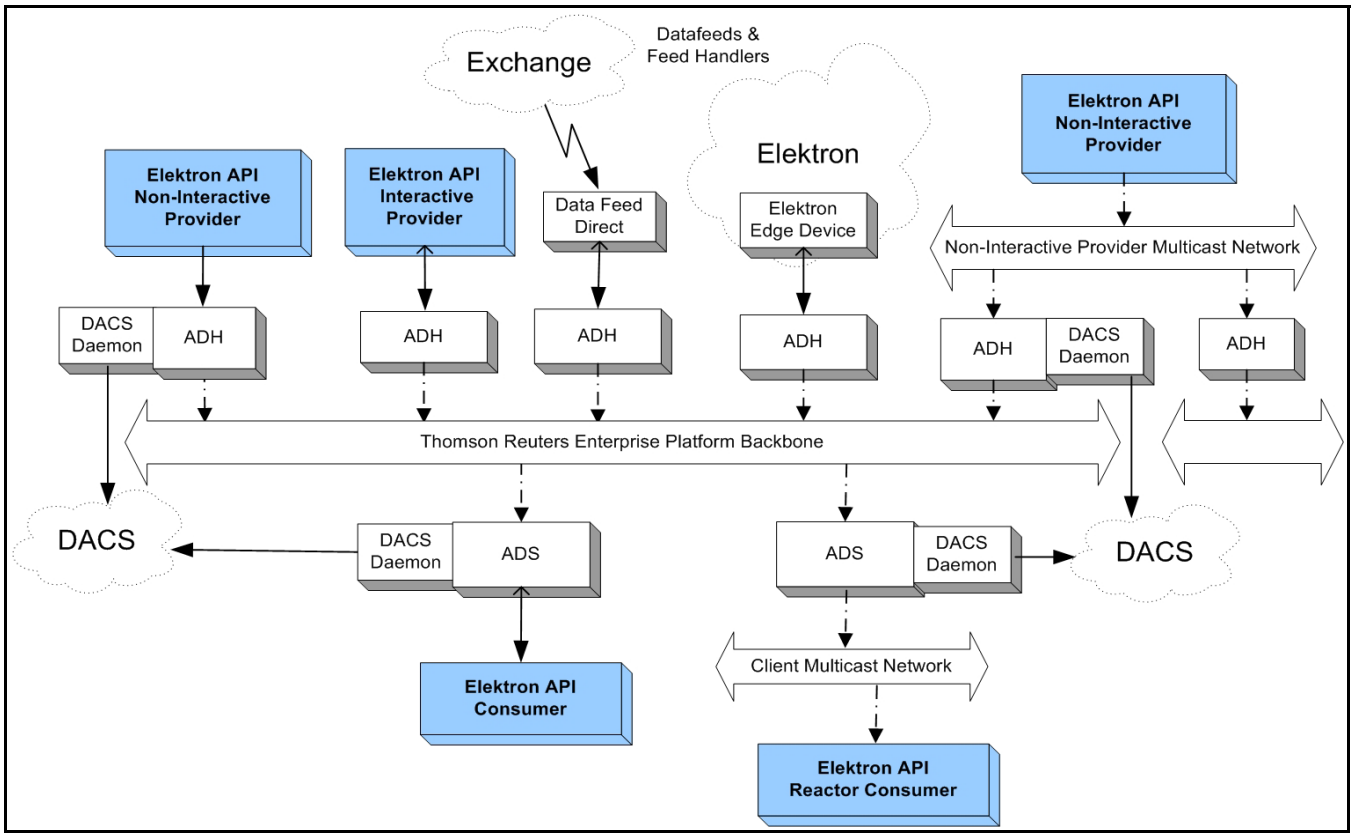

Figure 19. Typical TREP Components

4.2 Advanced Distribution Server (ADS)

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

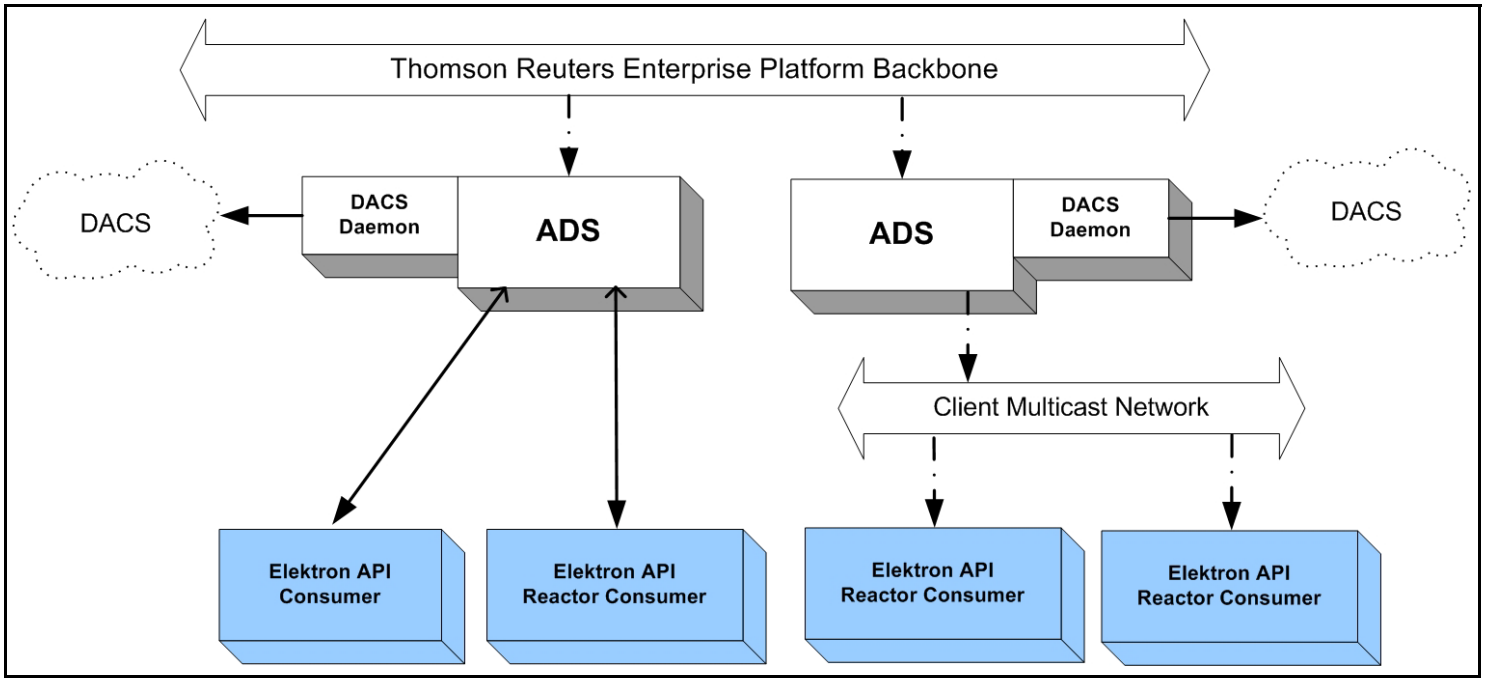

Figure 20. Elektron API and Advanced Distribution Server

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

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

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

To take advantage of multicast communications, consumers must use a Value-Add component. For information on using a Value Add component to receive communications via multicast, refer to the *Value Added Components Developers Guide* specific to the TREP API you use.

4.3 Advanced Data Hub (ADH)

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

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

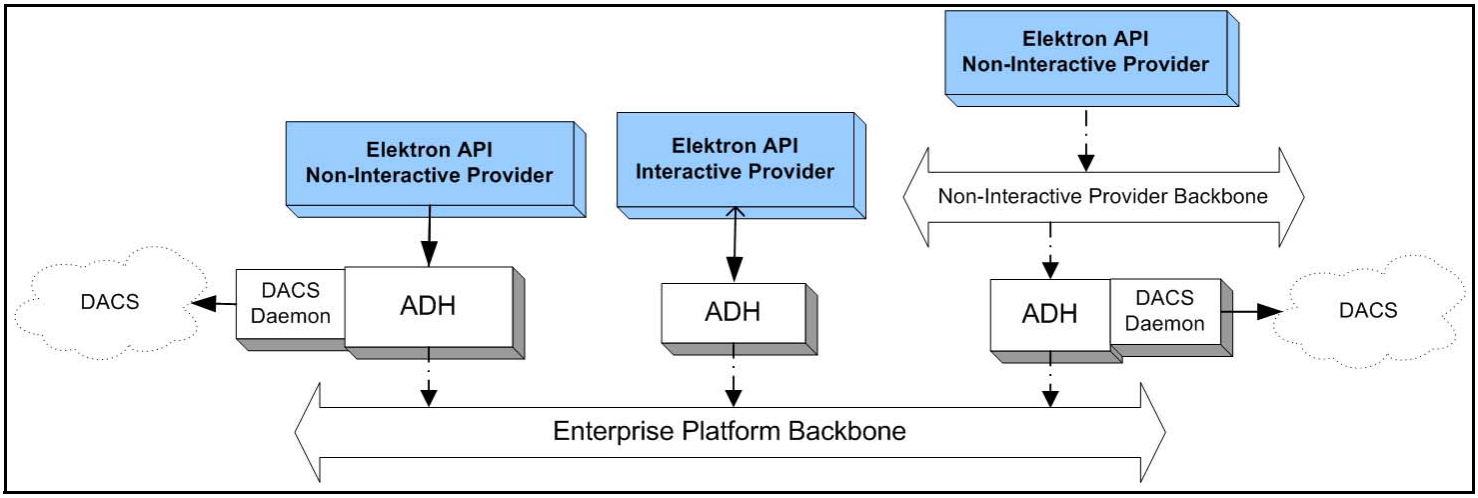

Figure 21. Elektron API and the Advanced Data Hub

4.4 Elektron

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

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

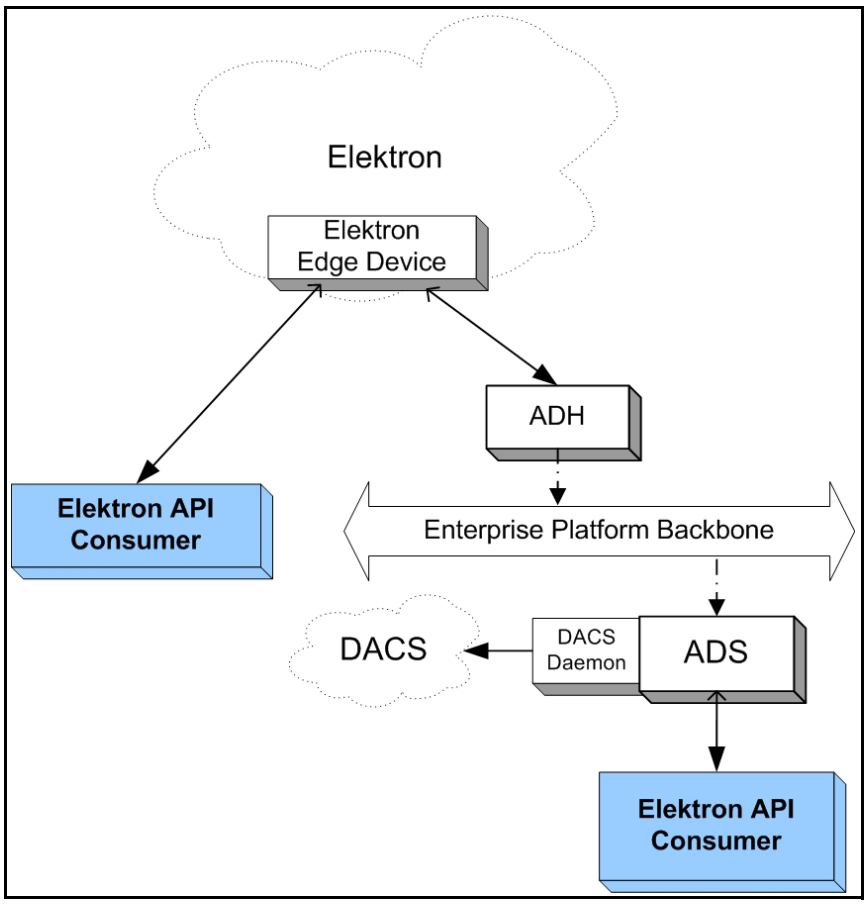

Figure 22. Elektron API and Elektron

4.5 Data Feed Direct

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

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

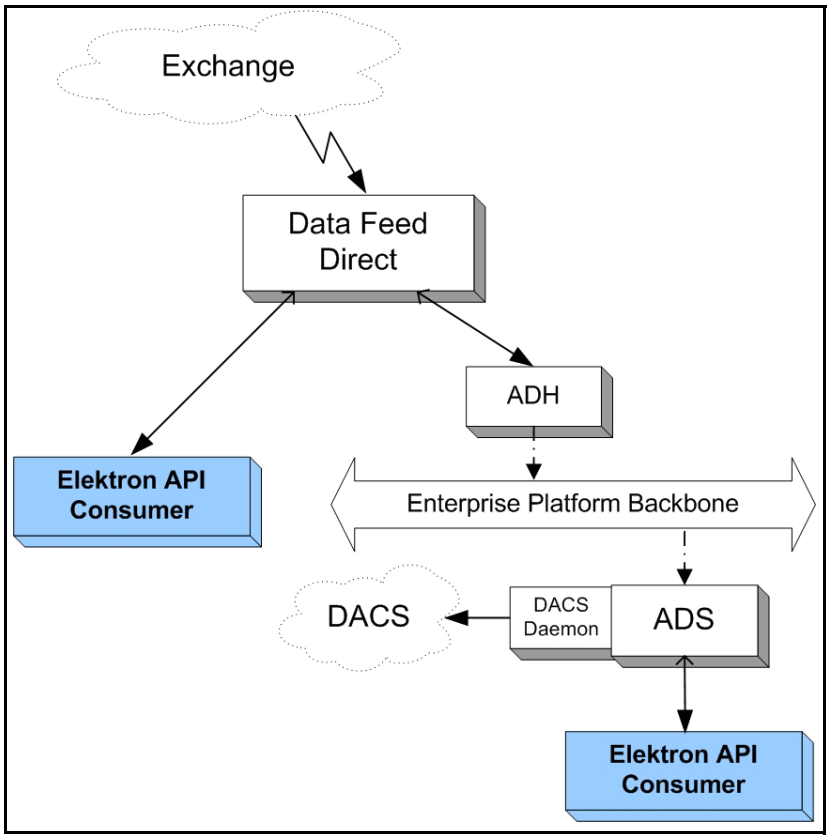

Figure 23. Elektron API and Data Feed Direct

4.6 Internet Connectivity via HTTP and HTTPS

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

- OMM consumer and NIP applications can establish connections via HTTP tunneling.
- ADS and OMM Interactive Provider applications can accept incoming Elektron API connections tunneled via HTTP (such functionality is available across all supported platforms).
- Consumer applications can leverage HTTPS to establish an encrypted tunnel to certain Thomson Reuters Hosted Solutions, performing key and certificate exchange.
- Consumer-side functionality leverages Microsoft WinINET. Users can configure certificate and proxy use via Internet Explorer. Because of its dependency on the Microsoft WinINET library, consumer HTTP and HTTPS tunneling are available only on supported Windows platforms.

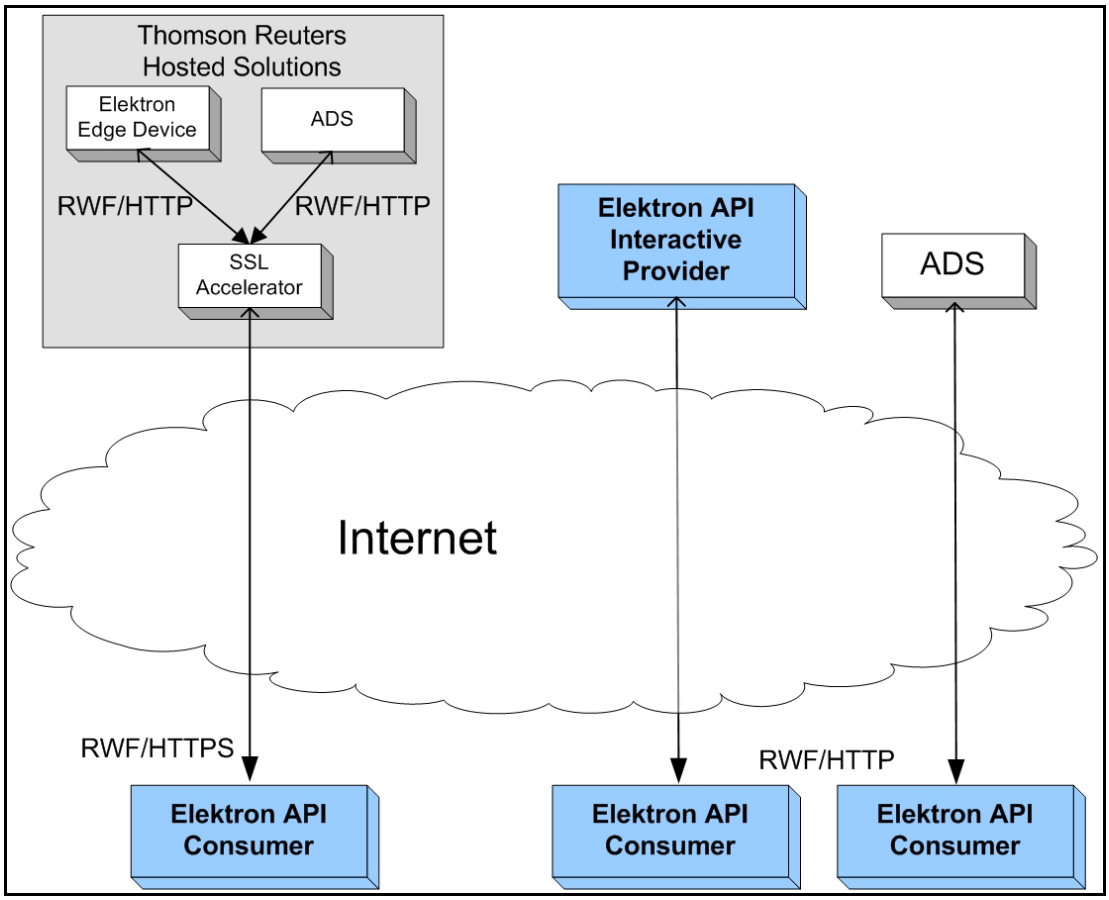

Figure 24. Elektron API and Internet Connectivity

4.7 Direct Connect

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

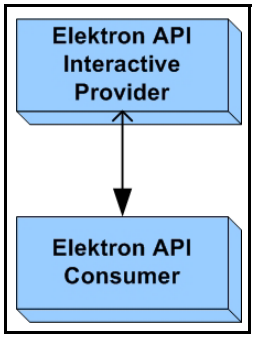

Figure 25. Transport API and Direct Connect

Chapter 5 Data Types and Messaging Concepts

5.1 Overview of Data Types

The Elektron API offer a wide variety of data types categorized into two groups:

- ***Primitive Types***: A primitive type represents simple, atomically updating information such as values like integers, dates, and ASCII string buffers (refer to Section 5.2).
- ***Container Types***: A container type can model data representations more intricately and manage dynamic content at a more granular level than primitive types. Container types represent complex information such as field identifier-value, name-value, or key-value pairs (refer to Section 5.3). Elektron API offers several uniform, homogeneous container types (i.e., all entries house the same type of data). Additionally, there are several non-uniform, heterogeneous container types in which different entries can hold different types of data.

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

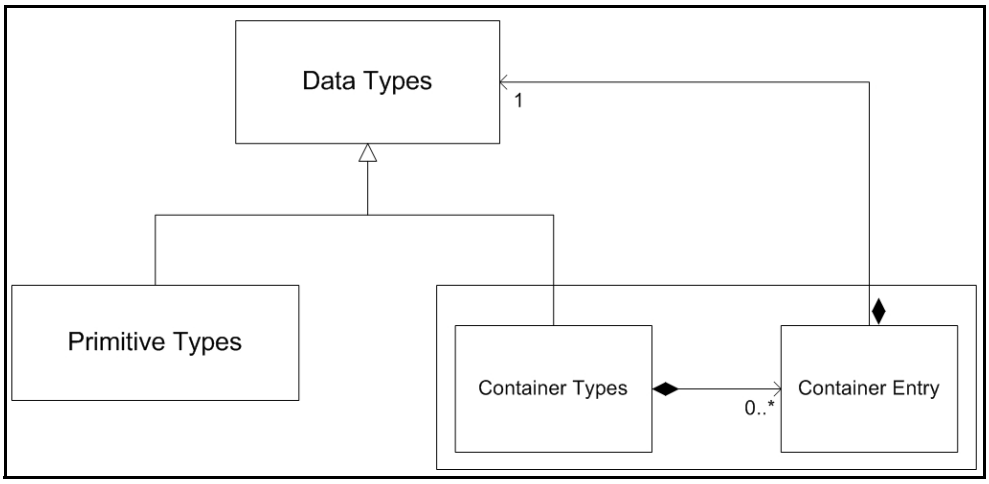

Figure 26. Elektron API and the Composite Pattern

The diagram highlights the following:

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

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

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

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

5.2 Primitive Types

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

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

PRIMITIVE TYPE	TYPE DESCRIPTION
None	Indicates that the type is unknown. This type is valid only when decoding a Field List type and a dictionary look-up is required to determine the type. This type cannot be passed into encoding or decoding functions.
Int[a]	A signed integer type. Can currently represent a value of up to 63 bits along with a one bit sign (positive or negative).
UInt[b]	An unsigned integer type. Can currently represent an unsigned value with precision of up to 64 bits.
Float	A four-byte, floating point type. Can represent the same range of values allowed with the system `Float` type. Follows IEEE 754 specification.
Double	An eight-byte, floating point type. Can represent the same range of values allowed with the system `Double` type. Follows IEEE 754 specification.
Real[c]	An optimized RWF representation of a decimal or fractional value which typically requires less bytes on the wire than `Float` or `Double` types. The user specifies a value with a hint for converting to decimal or fractional representation.
Date	Defines a date with month, day, and year values.
Time	Defines a time with hour, minute, second, millisecond, microsecond, and nanosecond values.
DateTime	Combined representation of date and time. Contains all members of date and time constructs.
Qos	Defines QoS information such as data timeliness (e.g., real time) and rate (e.g., tick-by-tick). Allows a user to send QoS information as part of the data payload. Similar information can also be conveyed using multiple message headers.
State	Represents data and stream state information. Allows a user to send state information as part of data payload. Similar information can also be conveyed in several message headers.
Enum[d]	Represents an enumeration type, defined as an unsigned, two-byte value. Many times, this enumeration value is cross-referenced with an enumeration dictionary (e.g., **enumtype.def**) or a well-known, enumeration definition (e.g., those contained in the package).
Array	The array type allows users to represent a simple base primitive type list (all primitive types except `Array`). The user can specify the base primitive type that an array carries and whether each is of a variable or fixed-length. Because the array is a primitive type, if any primitive value in the array updates, the entire array must be resent.
Buffer[e]	Represents a raw byte buffer type. Any semantics associated with the data in this buffer is provided from outside of the Elektron API, either via a field dictionary (e.g., **RDMFieldDictionary**) or a DMM definition.

Table 4: Elektron API Primitive Types

PRIMITIVE TYPE	TYPE DESCRIPTION
Buffer or String[e] (depends on the API)	Represents an ASCII string which should contain only characters that are valid in ASCII specification. Because this might be NULL terminated, use the provided length when accessing content. The Elektron API do not enforce or validate encoding standards: this is the user's responsibility.
Buffer[e]	Represents a UTF8 string which should follow the UTF8 encoding standard and contain only characters valid within that set. Because this might be NULL terminated, use the provided length when accessing content. The Elektron API do not enforce or validate encoding standards: this is the user's responsibility.
Buffer or RMTES buffer[f] (depends on the API)	Represents an RMTES (Reuters Multilingual Text Encoding Standard) string which should follow the RMTES encoding standard and contain only characters valid within that set. . The Elektron API provides utility functions to help with proper storage and converting RMTES strings.

Table 4: Elektron API Primitive Types (Continued)

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

5.3 Container Types

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

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

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

CONTAINER TYPE	DESCRIPTION	ENTRY TYPE INFORMATION
FieldList	A highly optimized, non-uniform type, that contains field identifier-value paired entries. `fieldId` refers to specific name and type information as defined in an external field dictionary (such as **RDMFieldDictionary**). You can further optimize this type by using set-defined data.	Entry type is `FieldEntry`, which can house any `DataType`, including set-defined data, base primitive types (Section 5.2), and container types. • If the information and entry being updated contains a primitive type, previously stored or displayed data is replaced. • If the entry contains another container type, action values associated with that type specify how to update the information.
ElementList	A self-describing, non-uniform type, with each entry containing `name`, `dataType`, and a value. This type is equivalent to `FieldList`, but without the optimizations provided through `fieldId` use. Use of set-defined data allows for further optimization.	Entry type is `ElementEntry`, which can house any `DataType`, including set-defined data, base primitive types (Section 5.2), and container types. • If the updating information and entry contain a primitive type, any previously stored or displayed data is replaced. • If the entry contains another container type, action values associated with that type specify how to update the information.
Map	A container of key-value paired entries. `Map` is a uniform type, where the base primitive type of each entry's key and the `containerType` of each entry's payload are specified on the `Map`.	Entry type is `MapEntry`, which can include only container types, as specified on the `Map`. Each entry's key is a base primitive type, as specified on the `Map`. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.
Series	A uniform type, where the `containerType` of each entry is specified on the `Series`. This container is often used to represent table-based information, where no explicit indexing is present or required. As entries are received, the user should append them to any previously-received entries.	Entry type is `SeriesEntry`, which can include only container types, as specified on the `Series`. `SeriesEntry` types do not contain explicit actions; though as entries are received, the user should append them to any previously received entries.

Table 5: Elektron API Container Types

CONTAINER TYPE	DESCRIPTION	ENTRY TYPE INFORMATION
Vector	A container of position index-value paired entries. This container is a uniform type, where the `containerType` of each entry's payload is specified on the `Vector`. Each entry's `index` is represented by an unsigned integer.	Entry type is `VectorEntry`, which can house only container types, as specified on the `Vector`. Each entry's `index` is an unsigned integer. Each entry has an associated action, which informs the user on how to apply the information stored in the entry.
FilterList	Entry type is `FilterEntry`, which can house only container types. Though the `FilterList` can specify a `containerType`, each entry can override this specification to house a different type. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.	Entry type is `FilterEntry`, which can house only container types. Though the `FilterList` can specify a `containerType`, each entry can override this specification to house a different type. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.
Msg	Indicates that the contents are another message. This allows the application to house a message within a message or a message within another container's entries. This type is typically used with posting.	None

Table 5: Elektron API Container Types (Continued)

5.4 Summary Data

Some container types allow summary data. ***Summary data*** conveys information that applies to every entry housed in the container. Using summary data ensures data is sent only once, instead of repetitively including data in each entry. An example of summary data is the currency type because it is likely that all entries in the container share the same currency. Summary data is optional and applications can determine when to employ it.

Specific domain model definitions typically indicate whether summary data should be present, along with information on its content. When included, the `containerType` of the summary data is expected to match the `containerType` of the payload information (e.g., if summary data is present on a `Vector`, the `Vector.containerType` defines the type of summary data and `VectorEntry` payload).

5.5 Messaging Concepts

Messages communicate data between system components: to exchange information, indicate status, permission users and access, and for a variety of other purposes. Many messages have associated semantics for efficient use in market data systems to request information, respond to information, or provide updated information. Other messages have relatively loose semantics, allowing for a more dynamic use either inside or outside market data systems.

An individual flow of related messages within a connection is typically referred to as a ***stream***, and the message package allows multiple simultaneous streams to coexist in a connection. An information stream is instantiated between a consuming application and a providing application when the consumer issues an `RequestMsg` followed by the provider responding with an `RefreshMsg` or `StatusMsg`. At this point the stream is established and allows other messages to flow within the stream. The remainder of this chapter discusses streams, stream identification, and stream uniqueness..

5.6 Message Class Information

MESSAGE CLASS	DESCRIPTION
Request Message	Consumers use **`RequestMsg`** to express interest in a new stream or modify some parameters on an existing stream; typically results in the delivery of an **`RefreshMsg`** or **`StatusMsg`**.
Refresh Message	The Interactive Provider can use this class to respond to a consumer's request for information (solicited) or provide a data resynchronization point (unsolicited). The NIP can use this class to initiate a data flow on a new item stream. Conveys state information, QoS, stream permissioning information, and group information in addition to payload.
Update Message	Interactive or NIPs use the **`UpdateMsg`** to convey changes to information on a stream. Update messages typically flow on a stream after delivery of a refresh.
Status Message	Indicates changes to the stream or data properties. A provider uses **`StatusMsg`** to close streams and to indicate successful establishment of a stream when there is no data to convey. This message can indicate changes: • In **`streamState`** or **`dataState`** • In a stream's permissioning information • To the item group to which the stream belongs
Close Message	A consumer uses **`CloseMsg`** to indicate no further interest in a stream. As a result, the stream should be closed. • The Transport API allows direct use of the Close message • The Message API implicitly handles this messaging functionality whenever a user unregisters.
Generic Message	A bi-directional message that does not have any implicit interaction semantics associated with it, thus the name generic. After a stream is established via a request-refresh/status interaction: • A consumer can send this message to a provider. • A provider can send this message to a consumer. • NIPs can send this message to the ADH.
Post Message	A consumer uses **`PostMsg`** to push content upstream. This information can be applied to an Enterprise Platform cache or routed further upstream to a data source. After receiving posted data, upstream components can republish it to downstream consumers.
Ack Message	A provider uses **`AckMsg`** to inform a consumer of success or failure for a specific **`PostMsg`** or **`CloseMsg`**.

Table 6: Message Class Information

5.7 Permission Data

Permission Data is optional authorization information. The DACS Lock API provides functionality for creating and manipulating permissioning information. For more information on DACS usage and permission data creation, refer to the *DACS LOCK Library Reference Manual* specific to the API you use.

Permission data can be specified in some messages. When permission data is included in a **`RefreshMsg`** or a **`StatusMsg`**, this generally defines authorization information associated with all content on the stream. You can change permission data on an existing stream by sending a subsequent **`StatusMsg`** or **`RefreshMsg`** which contains the new permission data. When permission data is included in an **`UpdateMsg`**, this generally defines authorization information that applies only to that specific **`UpdateMsg`**.

Permission data can also be specified in some container entries. When a container entry includes permission data, it generally defines authorization information that applies only to that specific container entry. Specific usage and inclusion of permissioning information can be further defined within a domain model specification.

Permission data typically ensures that only entitled parties can access restricted content. On TREP, all content is restricted (or filtered) based on user permissions.

When content is contributed, permission data in a **`PostMsg`** is used to permission the user who posts the information. If the payload of the **`PostMsg`** is another message type with permission data (i.e., **`RefreshMsg`**), the nested message's permissions can change the permission expression associated with the posted item. If permission data for the nested message is the same as permission data on the **`PostMsg`**, the nested message does not need permission data.

Document ID: API310UM.170
Date of issue: 28 April 2017