Fixnetix Market Data Platform

Functional Overview

A document describing the system functions of the Fixnetix Market Data Platform

November 10th 2009
Rev. 1.3

Prepared by: Fixentix Inc.

Table Of Contents

Functional Overview

Introduction

Purpose

This document provides an overview of the functionality and design of the Fixnetix Market Data Platform and the iX-Max component technology.

Scope

This document describes basic functionality only, as it relates to the acquisition, normalization and distribution of market data.

The next version of this document will provide information on resilience and monitoring.

Intended Audience

The audience for this document includes the following:

- Fixnetix management, technical and project staff.
- Selected clients, technical and project staff.

This document was prepared with the assumption that the reader is familiar with the systems and terminology used to describe ticker and message processing systems as well as the name and function of related systems in use at Fixnetix.

Revision History

This is document represent an initial draft revision that consolidates information previously distributed to Fixnetix.

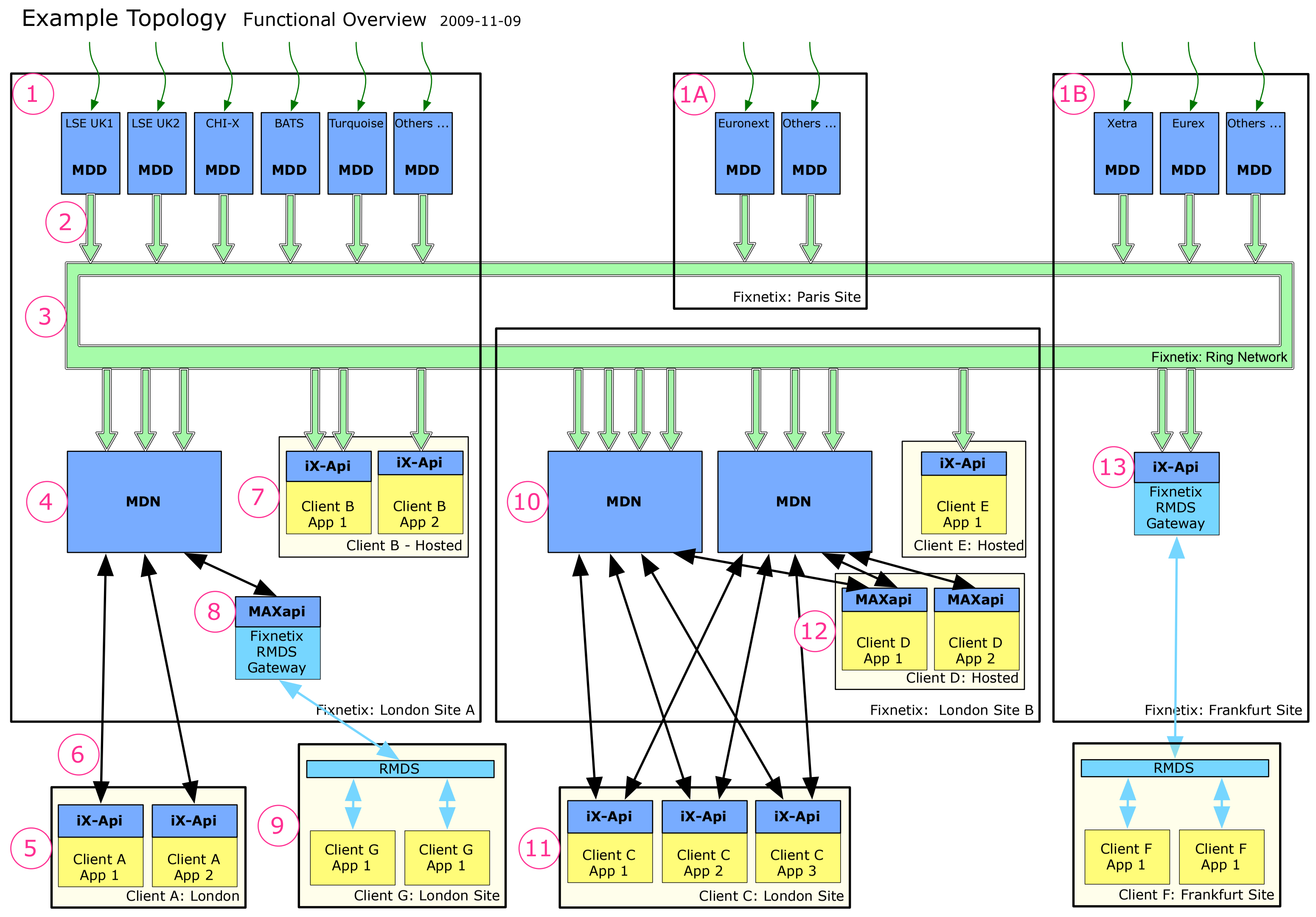
Name	Date	Reason For Change	Version
Trevor Miller	Nov 10th	Rebranding, and minor corrections.	1.3

References

iX-Api C++ Reference Manual 1.1

Macro Topology

This section illustrates and describes the potential deployment of iX-MAX components within a Fixnetix topology.



The diagram illustrates multiple Fixnetix sites and possible configurations of iX-MAX components at each site. It illustrates primary functional components and primary message data flow only. No attempt is made to show redundancy or back channels used for requesting refresh/repair transmissions.

Macro Topology

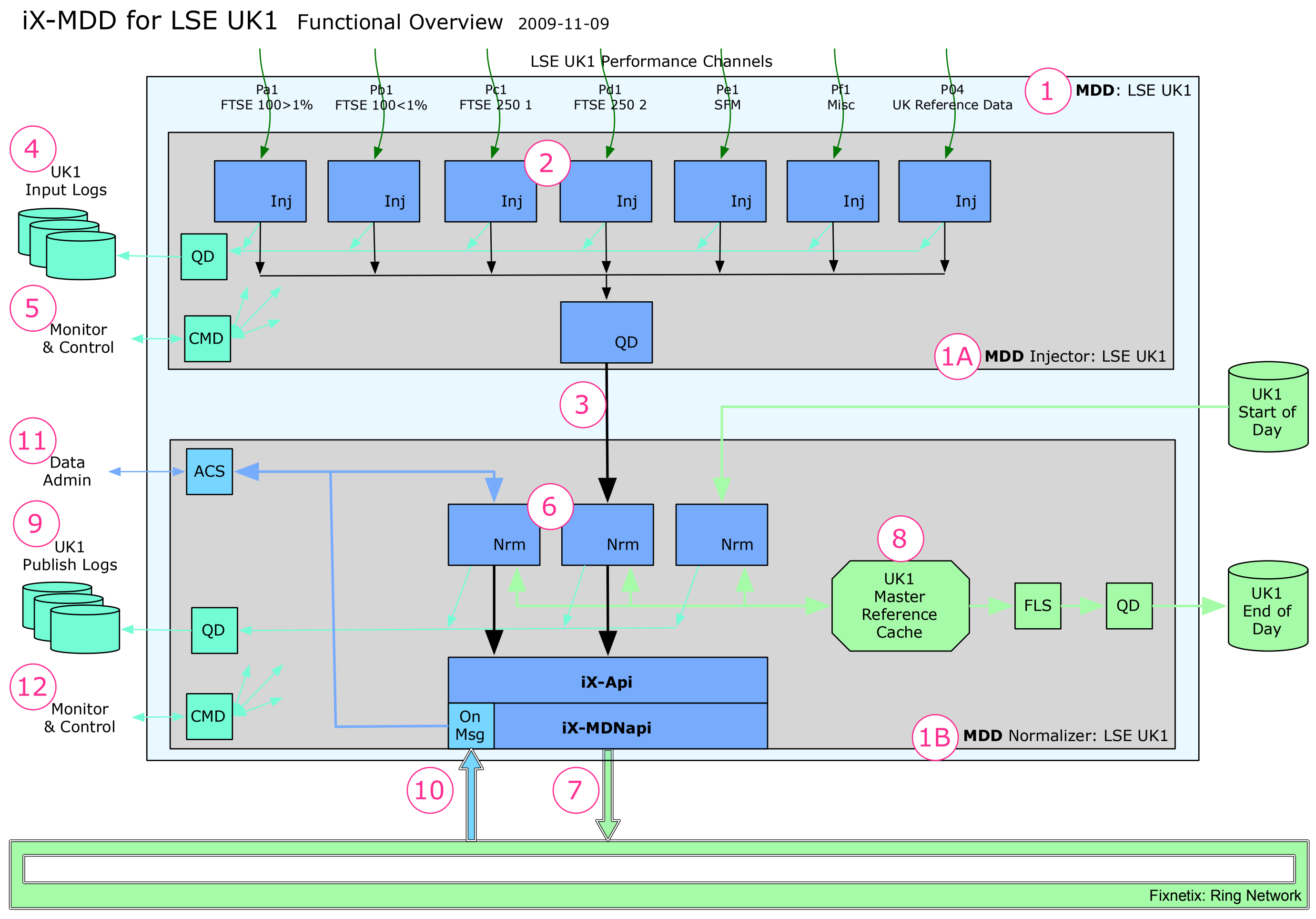
The notes below reference the numeric labels on the diagram.

1. A MDD instance will be deployed and configured to acquire and process input from a selected market data source. The term “feed handler” is commonly used to describe this layer of functionality within a real-time market data processing system. Hence each MDD can be thought of as a feed handler. MDD components are illustrated as being located at Fixnetix sites (1, 1A, 1B) physically close to origination points for each feed.
2. Each MDD will be configured to output a normalized message stream of selected content to a multicast channel. A reliable broadcast transport protocol will be used.
3. Fixnetix backbone “ring” distributes multicast streams to all other Fixnetix sites.
4. MDN components are deployed to Fixnetix sites and configured to receive required multicast channels. MDN components support caching and distribution functionality.
5. The iX-Api is embedded into client applications executing at client sites. Client applications may be configured to connect to target MDN connection points. Client applications establish subscriptions to solicit delivery of real-time market data. Client applications receive OnMessage data published on topics to which they subscribe. Each topic identifies a valid security traded (or reported) by a market center or trading venue.
6. TCP/IP transport protocol is used between client applications and MDN.
7. Client applications requiring ultra low-latency delivery may be hosted at Fixnetix sites. Hosted client applications may be configured to connect “directly” to multicast channels. Hosted client applications establish subscriptions to solicit delivery of real-time market data.
8. The iX-Api is embedded into Fixnetix RMDS gateways executing at Fixnetix sites. RMDS gateways may be configured to connect to target MDN connection points. RMDS gateways establish subscriptions to solicit delivery of real-time market data.
9. Client environments running RMDS receive real time market data from Fixnetix RMDS gateways.
10. An array of MDNs may be configured at Fixnetix sites, with content sets distributed across the array. Each MDN in the array is configured to receive a different set of multicast channels. This allows load to be distributed across multiple MDNs.
11. Client applications may be configured to connect to target MDN connection points. Each MDN connection identifies a “host group.” Each host group is configured as one or more TCP/IP socket connections to target MDN nodes. Client applications establish subscriptions to solicit delivery of real-time market data, across multiple MDN nodes.
12. Client applications may be hosted at Fixnetix sites. Hosted client applications may be configured to connect to target MDN connections. Each MDN connection identifies a host group. Each host group is configured as one or more TCP/IP socket connections to target MDNs. Client applications establish subscriptions to solicit delivery of real-time market data, across multiple MDN nodes.

13. Fixnetix RMDS gateways requiring ultra low latency execute at Fixnetix sites embed the iX-Api. RMDS gateways may be configured to connect “directly” to multicast channels. RMDS gateways establish subscriptions to solicit delivery of real-time market data.

iX-MDD

This section illustrates and describes the internal structure and functionality of an MDD component. MDD components are deployed to acquire and process market data feeds. The term “feed handler” is commonly used to describe this layer of functionality within a real-time market data processing system. Hence each MDD can be thought of as a feed handler component.



The above diagram illustrates the primary functionality of an MDD component. No attempt is made to show redundant components which address fault tolerant requirements.

iX-MDD

Primary message data flow through this MDD instance is shown along with back channels used for requesting refresh or repair transmissions.

The notes below reference the numeric labels on the diagram.

1. MDD for LSE UK1

An MDD instance will be deployed and configured to acquire and process input from a selected market data source. The term “feed handler” is commonly used to describe this layer of functionality within a real-time market data processing system. Hence each MDD can be thought of as a feed handler.

Each MDD will be located at a Fixnetix site, typically close to the origination point for the market data feed which it will be configured to process. These MDDs may be designated as a primary instance, and will run hot, processing live market data and actively generating output to downstream components. A secondary or duplicate instance of a primary MDD may be configured to run warm, processing live market data but inhibited from generating output to downstream components. A secondary MDD may be located at the same Fixnetix site as the primary MDD, for which it is a duplicate, or it may run at an alternative Fixnetix site.

Each MDD consists of two cooperating multithreaded processes, configured to run on a single server. The diagram illustrates the internal structure and data flow for a MDD instance configured to process LSE Level 1 data sourced from a subset of the LSE Infolect Performance Channels.

1.A. MDD Injector Process

A MDD Injector Process will acquire data from one or more input message streams, verify, filter and integrate required messages into an output message stream, which is forwarded for processing to the MDD Normalizer Process.

1.B. MDD Normalizer Process

A MDD Normalizer Process will receive a message stream forwarded by the MDD Injector Process. It will process each message to generate and publish a message stream as output, which expresses the information conveyed in the received messages. The output message stream will use the ML2 messaging constructs, which will be standard across all deployed MDDs.

2. MDD Injector

A MDD Injector process contains multiple threads. The primary thread type of an Injector process is an Injector thread. Other threads are general purpose utility threads and are not specific to the required functioning of an Injector process.

One or more Injector threads (Inj) are configured within an Injector process. Each Injector thread will be configured to consume one specific market data stream as input.

The set of Injector threads configured within an Injector process will receive a set of market data streams which encompasses the content required to form a discrete market data set, generally corresponding to all instruments in one or more market segments of a trading venue or exchange.

In the diagram, seven Injector threads (Inj) are illustrated within an Injector process. Each is shown as supporting a specific LSE Infolect Performance channel. In this example the total set of seven channels encompasses all the content published by the LSE for a selected set of tradable securities. In this example, the content set for LSE Level 1 is configured.

Functionality

Each Injector thread will be configured to support the transport, session and messaging protocol required for one specific market data stream.

An Injector thread will verify input message stream integrity by parsing received messages and extracting source sequence numbers where provided by the market data source. An Injector thread will then process each parsed input message according to its message type as specified in a configuration table.

An Injector thread will detect and report gaps in the received message stream and will maintain a record of missed messages. If the session protocol for a market data stream supports recovery of missed messages (repair of gaps), then an Injector thread will respond to detected message gaps by requesting retransmissions from the source.

Administrative messages (such as heartbeats, acknowledgements , etc.) that are used to administer message flow within a session will be processed by the Injector thread and not forwarded for downstream processing (they are filtered out).

3. Injector Output

Messages (message types) received by an Injector thread which convey market data information (as defined by configuration table entires) are integrated into (one or more) output message streams for forwarding to a Normalizer process by way of an interprocess queue. An Injector thread supplements received message content with meta data before forwarding. This meta data will include the time of receipt (origin time) of each message with microsecond precision.

A Queue Driver thread in an Injector process manages the delivery of messages to be forwarded from an Injector to a Normalizer. [A Queue Driver thread (QD) is a utility thread, which decouples an Injector thread from the interprocess communication protocol used to transport messages from one process to the next.]

A Queue Driver thread may be configured or switched to virtual or direct-mode. These modes inform the sending thread of the manner in which to invoke queue utility functions to transport a message on an output queue. Direct mode may be used to reduce latency and increase deterministic behavior of a system.

4. Injector Logs

All messages parsed from an input market data stream by an Injector thread will be logged (by default).

- Separate log files will be configured to store messages for each input stream.
- Logging for each input stream may be disabled by configuration or by command.
- Utility programs (display) will be used to interrogate log files to analyze or audit received messages.
- Log files will be archived as part of an end-of-day processing.
- Log files may be use as emulator input in MDD test or diagnostic environments.

5. Injector Monitoring and Control

Each Injector process will include a configured Command thread. [A Command thread is a utility thread, which serves to enquiry and control requests submitted from a Monitoring subsystem, scheduled operational control scripts or command line tools. A Command thread within a process listens for and accepts connections, accepts requests, invokes embedded command functions, returns results sets to the requester and closes connections on completion.]

An Injector process will be configured to support the following command request types:

injector_stats	Control and monitor input message streams
queue_stats	Control and monitor queue objects, and return queue metrics
seq_stats	Monitor and administer missing message metrics
task_stats	Control and monitor threads, and return thread metrics
transport	Control and monitor specific transport protocols underlying queues

6. MDD Normalizer Threads

A MDD Normalizer process contains multiple threads. The primary thread type in a Normalizer process is a Normalizer thread. Other threads are general purpose utility threads, and are not specific to the required functioning of a Normalizer process.

One or more Normalizer threads (Nrm) are configured within an Injector process. Each Normalizer thread will be configured to consume messages from one specific Normalizer input queue. Messages which originate from a market data source will be sent to a Normalizer input queue by an Injector process. This will be the primary real-time message stream input to a Normalizer process.

Refresh requests, data administration requests and reference cache load requests will also result in messages being directed to Normalizer threads by way of queues. These are considered ancillary requests. These requests are executed by Normalizer threads which are peers to the primary Normalizer thread processing messages which originate from a market data source.

Functionality

A Normalizer thread implements a framework which supports plug-in functions. Plug-in functions are developed for each market data source supported. These implement the MDD logic required to process each message type from each market data source. Plug-in functions also exist to process internal requests, such as data administration requests, refresh requests or data load requests.

A Normalizer thread will process each received message (externally or internally sourced) according to meta data which is passed with the received message. The meta data for an input message includes a key (or index), which is identifies the specific plug-in function to be invoked to interpret and process the received message.

The steps included in processing market data messages vary according to the semantics of each message type from each market data source. Typically, the following steps will be executed for an externally sourced message which reports a market data event, such as a trade, a quote or an order:

- Construct the unique identification key for the tradable instrument from attribute values reported in the received message;
- Establish a lock using the identification key to prevent concurrent access;
- Read an in-memory cache record which contains current context for the instrument using the identification key
  OR
  create a new in-memory cache record if one does not already exist for the instrument;
- Verify that the received message is a valid update using the current context;
- Derive value added attributes for the market data event reported, if required;
- Construct a normalized representation of the market data event reported including added-value attributes (an ML2 message set);
- Construct the topic name on which the message set should be published;
- Publish the message set through a MDN session;
- Update the current context for the instrument;
- Commit the context back to cache; and
- Remove the lock on the identification key.

Topic Namespace

Each message published by a Normalizer is published to a destination defined by a topic name. All topic names used for publishing message which convey market data encode a domain, a market center and a symbol value.

Domains segment the universe of supported items at the highest level. Domains allow inclusion of information for the same traded instrument supplied by different sources. For example, market data for a traded security may be sourced directly from a market center as well as being sourced indirectly through a third party vendor feed. In such a case, two items in the universe may have the same market center and symbol values but are distinguished by different domain values.

For an item that represents a traded security, the market value identifies a market center on which the instrument is traded and to which information in the message payload pertains. The market center value is also used to distinguish between level one and level two (depth of market or order book) data for the same traded security. Where both are supported, two items will exist distinguished by different market values.

Most often the symbol value used to identify a traded security corresponds to the symbol or mnemonic code commonly used by the primary market on which the security is traded. Often such symbols are used to identify each security on the originating feed which acts as (input to) the message source.

ML2 Message Sets

A Normalizer process functions to construct and publish a normalized representations of market data events using Fixnetix ML2 syntax, semantics and encoding scheme. An ML2Message is a message that contains a payload which conveys market data relating to a trade-able security, an index or other object. (The term item is used as a general term for any security, index, etc.)

Every published ML2Message is member of a message set. A message set may contain one or more messages. Certain market data events require multiple messages to be published to fully represent the event and to define the resultant state for a tradable security. These multiple messages would be published as a single message set. The large majority of market data events will be published by a Normalizer as a message set containing one message.

All messages in a set are guaranteed to be delivered to a subscribing client application in the correct sequence, with no intervening messages for the specific item. A client application may process the complete message set as if it were an atomic event for the specific item.

7. Normalizer Message Streams

A Normalizer process will be configured to support market data which encompasses all the content (instruments and attributes) required to form a discrete market data set generally corresponding to all instruments in one or more market segments of a trading venue or exchange.

The iX-Api and MDNapi are both embedded in a Normalizer process. The iX-Api is layered above the MDNapi. A Normalizer process initializes the iX-Api/MDNapi on start up and provides a configured connection name. The connection name identifies a MDN host group, which includes a virtual MDN host. The virtual MDN host is instantiated as a multicast channel.

In this way a Normalizer process performs the role of a source where the act of publishing ML2Messages through the iX-Api results in multicasting of sequenced data packets (containing the published ML2 messages). Hence all content published by a Normalizer process will be published on a single multicast channel.

A reliable multicast transport protocol will be used. Selection of the specific transport protocol is configurable.

8. Master Reference Cache

A Normalizer process contains an in-memory cache, termed a master reference cache. A record in the cache is maintained for each instrument within the market data set configured to be supported by the Normalizer process.

Multiple indices are maintained within a cache allowing direct access to each instrument record by different key values. When a market data message is processed by a Normalizer, a unique identification key value is formed from the attribute values in the received message. The formation of the identification key is dependent on the semantics of messages from each market data source. For LSE reported instruments (Level 1) for example, the identification key is formed by the concatenation of a market identifier and an ISIN code. This identification key is the primary key for accessing records within the cache.

Different cache record types are supported within a cache allowing the attributes within of a record to vary according to the type of instrument data being stored.

Following completion of initialization (start up) of a Normalizer process the content of the in-memory cache is loaded. A Normalizer thread is configured to read from a named input queue. The named queue is instantiated as a file. This file contains messages where each message defines part or all the attribute values for an instrument cache record, along with meta data that identifies the specific instrument. This file is show on the diagram as UK1 Start of Day.

The content of a master reference cache will be saved at the end of a trading day, prior to the shut down of a Normalizer process. A Flusher thread (FLS) will be started by scheduled command. The Flusher thread will read through all records within a cache, form messages based on the content contained in a record, add meta data identifying each instrument and send the resultant message set to an output queue. The output queue is instantiated as a file. This file is show on the diagram as UK1 End of Day.

The resultant end of day file will be archived, and will be copied to act as the start of day file for the next trading day.

9. Normalizer Logs

All messages published by a Normalizer process will be logged (by default).

Separate log files will be configured to store messages for each published stream (multicast channel).

Logging for each output stream may be disabled by configuration or by command.

Utility programs (display) will be used to interrogate log files to analyze or audit published messages.

Log files will be archived as part of an end-of-day processing.

10. Normalizer Processing of Client Refresh Requests

At any time there will be one Normalizer instance which is considered to “own” the current and correct values for an instrument. Subordinate downstream components subscribe to receive messages published by an owning Normalizer and maintain current values for selected instruments. All subordinate downstream components work to maintain an accurate and up-to-date copy of the instrument data held in the owning Master Reference Cache, for the set of instruments of interest.

If there is a subordinate component that needs to receive the current “state” for a specific instrument, either on initial subscription, or to repair state on detection of a missing or in-error update, then a refresh request is routed to the nearest upstream cache which holds a copy of state for that instrument.

A refresh request may be generated by a component that connects directly (joins) the multicast group to which a Normalizer publishes. In this case the refresh request must propagate to the Normalizer process that “owns” the current and correct values for the specific instrument to be refreshed. In this case a refresh request generated by a subordinate component is published (by or through the iX-Api) to a topic which identifies the content “set” that contains the instrument to be refreshed.

A Normalizer acts primarily as a publisher of messages to the iX-Api. However a Normalizer also subscribes through the iX-Api. During initialization a Normalizer process will create a subscription to receive refresh request messages published to a set of well-known topics. These topics use a syntax which identifies the content set “owned” by the Normalizer.

The Normalizer defines a callback function to be invoked on receipt of a message published to a refresh topic. The callback function is executed by a MDNapi on-message thread (OnMsg). The callback function forwards a received refresh message to a configured queue. This queue (which is instantiated as an in-memory list) is read by a Normalizer thread.

A Normalizer thread will process each received message according meta data which is passed with the received message. In this case the meta data includes a key (or index), which is identifies the specific plug-in function to be invoked to interpret and process the received refresh message.

Typically the following steps will be executed by the plug-in function configured to process a refresh request:

- Construct the unique identification key for the instrument from attribute values contained in the received refresh request message
- Establish a lock using the identification key to prevent concurrent access
- Read an in-memory cache record which contains current context for the instrument, using the identification key

OR
if the instrument does not exist, publish a message indicating the item does not exist

- Construct a normalized representation of the current state for the instrument (an ML2 message set)
- Construct the topic name on which the message set should be published in this case the topic name is a private topic, identifying the specific down stream component which issued the refresh request
- Publish the message set by way of a MDN session on the private topic
- Remove the lock on the identification key

A Normalizer process will respond to a refresh request by publishing a message set which fully represents the current state for the instrument requested. The message set will be published through the iX-Api to the configured multicast channel for the content set containing the instrument. The published message set will reflect the current state for the requested instrument, and will fully account for all messages already published for the instrument, and none that will be published after.

The refresh message set is published on a private topic, that is sent by the requesting downstream component as meta data in the refresh request. By using a private topic a response to a refresh request will be processed only by the component requesting the refresh.

11. Data Administration Interface

Each Normalizer process will include a configured Access thread (ACS). An Access thread is a utility thread, which serves enquiry and control requests submitted from scheduled data administration control scripts, or from command line data administration tools. An Access thread within a process listens and accepts for connections, accepts requests, invokes embedded command functions, returns results sets to the requester and closes connections on completion.

A Normalizer process will be configured to support the following data administration request types:

Load Cache	Start the Normalizer thread to read from the start-of-day queue, and load the cache
Save Cache	Start the Flusher thread to read through the cache and write to the end-of-day queue
Send Refresh	Generate and publish a message set reflecting the current state for an identified instrument or instruments
View Record	Read the cache for an identified instrument and return a message set reflecting current values
Go Active	Activate a secondary instance MDD, causing it begin publishing, and become primary

Go Standby	De-activate a primary instance MDD, causing it to stop publishing, and become secondary

12. Normalizer Monitoring and Control

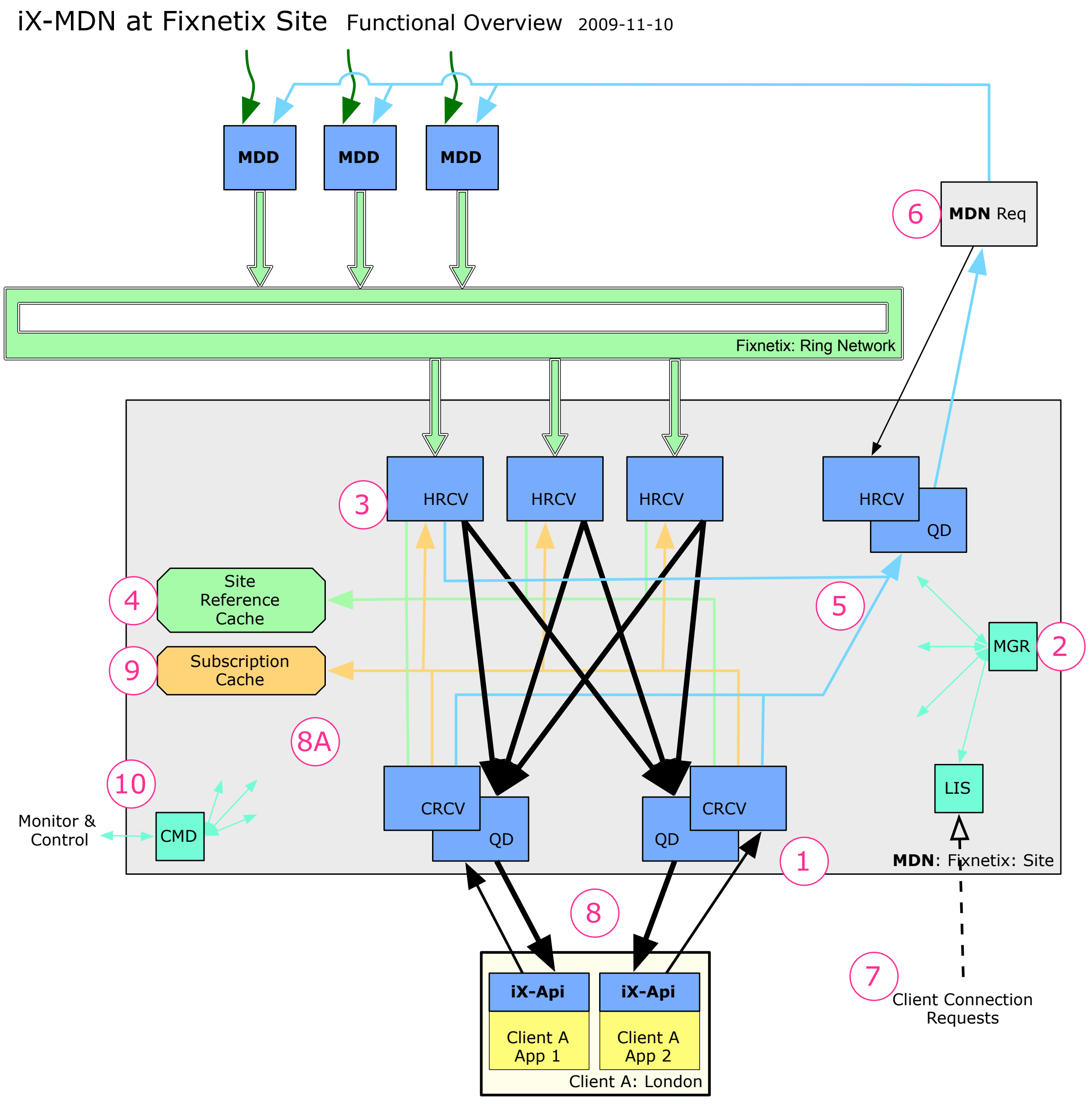
Each Normalizer process will include a configured Command thread (CMD). [A Command thread is a utility thread, which serves enquiry and control requests submitted from a Monitoring subsystem, from scheduled operational control scripts, or from command line tools. A Command thread within a process listens and accepts for connections, accepts requests, invokes embedded command functions, returns results sets to the requester and closes connections on completion.]

A Normalizer process will be configured to support the following command request types:

cache_stats	
host_stats	
injector_stats	Control and monitor input message streams
queue_stats	Control and monitor queue objects, and return queue metrics
seq_stats	Monitor and administer missing message metrics
task_stats	Control and monitor threads, and return thread metrics
transport	Control and monitor specific transport protocols underlying queues

iX-MDN

This section illustrates and describes the internal structure and functionality of an MDN component. An MDN component is deployed to receive, cache and distribute market data to client applications.



This above diagram illustrates the internal structure and functionality of an MDN component. An MDN component is deployed to receive, cache and distribute market data to client applications. This diagram illustrates primary functionality only. No attempt is made to show redundant components which address fault tolerance requirements.

Primary message data flow through this MDN instance is shown along with back channels used for requesting refresh or repair transmissions. The terms MDN node and MDN process are synonymous.

The notes below reference the numeric labels on the diagram.

1. MDN for Fixnetix Site

The diagram illustrates the internal structure and data flow for a single MDN process.

A MDN process will be deployed and configured to receive messages from one or more multicast channels, maintain an in-memory cache for each topic on which messages are published, and to selectively distribute messages to connected client applications that subscribe to receive market data for specific topics. A MDN process will be located at a Fixnetix site, typically close to the client applications which will connect to it. A MDN process may be designated as a primary instance, and will run hot, processing live market data and actively generating output to connected subscribing client applications.

A secondary or duplicate instance of a primary MDN process may be configured to run in parallel. The secondary MDN process will provide a secondary or fail-over connection point for client applications in the event of failure of the primary MDN process. A secondary MDN process will be configured to support the same content set as the primary MDN process for which it is a duplicate. A secondary MDN process may be located at the same Fixnetix site as the primary MDN mode, or it may run at an alternative Fixnetix site.

2. MDN Process

A MDN process executes multiple threads that cooperate to implement the required functions. Each MDN process is defined by a configuration specified in configuration table files. A MDN process is data agnostic, in that it is designed to function independently of the meaning or interpretation of the message payloads that it distributes.

A Manager thread (MGR) is responsible for overall control of the process. A manager thread is created as part of process initialization. It manages and coordinates asynchronous events such as client connection requests and subscription requests from clients, host or client connection failures, etc. Events are reported to the manager by other threads including the client receiver and host receiver threads which signal changes in the status of client and host connections. The manager thread takes actions depending upon the event reported and the current state of the process. Actions might include reestablishing host connections, terminating failed client connections, etc.

3. Host Receiver Threads

A MDN process will be configured to support one or more discrete market data sets. A market data set will generally include all instruments reported for one or more market segments of a trading venue or exchange, as processed and distributed by a single MDD instance over a single multicast channel.

A MDN process will use a named connection identified in configuration tables. Within the configuration tables, a connection name identifies a MDN host group. A MDN host group will identify one or more virtual MDN hosts. Each virtual MDN host is instantiated as a specific multicast channel. Hence during initialization, a MDN process joins one or more multicast groups and each host receiver thread takes the role of a receiver on a multicast channel.

The diagram illustrates a MDN process which is configured with 3 host receiver threads, each receiving message streams published by three MDD instances, over three reliable multicast channels. The specific transport protocol to be used is configurable. A reliable multicast protocol will be used.

Functionality

A host receiver thread receives and processes all messages published by a MDD on an input multicast channel.

The following steps will be executed by a host receiver thread on receipt of a published message:

- Read the subscription cache entry for the topic on which the received message was published;
- Identify each connected client that has a current subscription for the topic;
- Forward the message for output to each subscribing client;
- Read the site reference cache record for the topic
  OR
  create a new record if one does not already exist;
- Verify that the received message is consistent with the current context held for the topic;
- Update the current context held for the topic to reflect the message received;
- Commit the context back to the site reference cache; and
- If loss of integrity was detected:
    - Initiate recovery of current state for the topic;
    - Construct an “Instrument Status” message with a status of “Indicative” for the topic; and
    - Forward the message for output to each subscribing client.

4. Site Reference Cache

A MDN process is configured to contain an in-memory cache, termed a Site Reference Cache. The primary function of a MDN Site Reference Cache is to allow a full representation of current state to be sent to a client application when a new subscription for a topic is created by that client application. Delivery of current state provides a subscribing client application with information relating to all market data events already published on the topic, allowing a late joining client application to build the correct context for interpretation and processing of subsequently published messages. [Full representation of current state is published as a message set. The message set is delivered to a client application in the correct sequence relative to messages (message sets) already published on the topic, and to be published on the topic.]

A MDN process is data agnostic, in that it is designed to function independently of the meaning or interpretation of the message payloads that it distributes. In this case, a MDN process is specifically configured with a Site Reference Cache and with a plug-in library that supports the ML2 message semantics. The configured plug-in library implements update, verification, recovery and reference functions specifically for ML2 message streams.

Maintenance

A record in the cache is maintained for each topic on which received messages have been published, regardless of whether any client application has subscribed to a topic. [In general each topic identifies a discrete tradable security reported by a market center or trading venue.]

On initialization of a MDN process, the cache is empty. A record is created automatically in the cache on receipt by a host receiver thread of the first message published to a topic. [A record is also created automatically on receipt of a new subscription for a topic, for which no published messages have yet been received.]

A record is updated in the cache on receipt of each subsequent message published to a topic. A record contains the "last" instance of each ML2 message type published for the topic. An update includes evaluation of the integrity of the message stream received for the topic. This is based on the current context held in the cache record for the topic and meta-data contained in the message received.

The meta data used for verification of the message stream includes the following message attributes:

Revision	A revision number is maintained for each item by the MDD message source which "owns" that item. Note that messages for an item (e.g. a tradable security reported by a market center) are published to a topic specific to that item, hence in this context item and topic are synonymous. An item revision number is reset to zero before the start of each trading session. This is performed as part of an initialization or roll-forward activity performed by the message source (MDD) for all items on a given market (or segment of a market). The revision value conveyed by the message set (and contained in each message within the message set) reflects the revision for the item at the point at which the message set is published. When a market data event (such as a new quote or trade) triggers a change to values for an item then the revision number is incremented to reflect that change. Revision numbers in message sets reporting new events will increment by one. Revision numbers in message sets which refresh or recap current state will carry an event number reflecting the latest reported revision.
Set Size	Every ML2Message is member of a message set. A message set may contain one or more messages. The set size in a message defines how many messages are in the set of which this message is a member.

Set Identity	If a message set contains more than one message, a set identity value will be carried by all messages in the set.
Set Position	If a message set contains more than one message, a set position value identifies the position of each message in the set.

States

A record in the cache is maintained for each topic on which received messages have been published. Each record is managed independently based on interpretation of the received message stream. At any time, a record (the content within the record held for a specific item or topic) is considered to be in one of the following states:

Initial	The item record has been created as a result of receipt of a subscription but is empty, no messages have been received.
Firm	The record fully represents the current status for the item/topic based on the received message stream.
Indicative	An error has been detected in the received message stream. On transition to this state, a refresh request is published upstream to solicit a repair transmission for the topic. Also, an “Instrument Status” message is constructed for the topic with a status of “Indicative”. This message is forwarded for output to each client subscribing to the topic as an alert to the loss of integrity.
Recovering	One or more (but not all) messages have been received in a message set that is in the process of conveying a full representation of state for the item. Once completed successfully the state will transition to Firm.
Building	One or more (but not all) messages have been received in a message set that is conveying an update event. Once completed successfully the state will transition to Firm.

Usage

Support for caching for “Level 2” items which use the ML2 Book Order and Level messages (published by a MDD) is consistent with Level 1 items. An additional cache data structure is configured to hold discrete Orders/Levels separately from the cache that holds all other ML2 message images for items.

This additional cache has a primary (default) index of the order tag (e.g. L2.ZJ00CI9RJC). The primary index allows orders to be accessed directly for change or delete actions. A secondary index uses the item “ident” as a key (e.g. E\I\L2\VOD). This second index allows all orders for an item to be accessed for full item refresh or delete-all actions.

Implementation of caches within MDN nodes and within the MDNapi embedded in client applications supports the following functionality consistently for both Level 1 and Level 2 items:

- Automatic response to a client application creating a new subscriptions by publication of full state for the selected item.
    - A response to a valid subscription request will be a message set conveying full state for the requested item.
    - Full state for the is retrieved from nearest upstream cache or from the owning MDD instance if no cache is located with current firm copy.
    - Publication of full state is a directed response sent only to the new subscriber and is not seen by any other subscribers.
    - Publication of full state is correctly synchronized with previously published messages for the item, hence a subscribing client application does not need to juggle received messages pending reception of full state for an item, but can process all messages serially as they are received.
- Subscription requests for an “invalid” item (a non-existing topic) results in publication of a subscription error response.
    - A subscription error response is published as a ResponseStatus message carrying an indication that the item requested was not found.
    - A subscription is created and maintained even for a non-existing item. If the requested item is subsequently created, then messages published for the item are automatically distributed to the subscribing client.
    - This allows a client application to pre-subscribe for items that may be subsequently created and quoted for example.
    - An initial subscription request for a nonexistent item will result in an enquiry being sent upstream to an owning MDD instance.
    - However any subsequent subscription requests for the same item will be resolved from the nearest upstream cache, avoiding further requests to the owning MDD.
- Snapshot requests retrieve current state for an item.
    - A response to a valid snapshot request will be a message set conveying full state for the requested item.
    - Full state for the item is retrieved from nearest upstream cache or from the owning MDD instance if no cache is located with a firm copy.
    - A snapshot response is a directed response going only to client application issuing the snapshot request.
    - If a client application issues a snapshot request for an item that is already being subscribed to by that client application, then full state is retrieved from the client application's own local cache.

- Snapshot requests for an “invalid” item (a non-existing topic) results in publication of a snapshot error response.
  - A snapshot error is published as a ResponseStatus message carrying an indication that the item requested was not found.
  - An initial snapshot request for a nonexistent item will result in an enquiry being sent to the owning MDD instance.
  - However the “invalid” item snapshot request is remembered in the nearest upstream (MDN) cache.
  - Any subsequent snapshot requests for the same item will be resolved from that nearest upstream cache, avoiding further requests being sent to the owning MDD.
  - This promotes efficient handling of repeat submission of poorly formed snapshot requests and localizes resolution of those requests avoiding impact on an owning MDD.
- Automatic notification of loss of integrity of the received message stream for each item.
  - Maintenance of a cache includes verification of the message stream being received for each item.
  - Notification of loss of integrity is by publication of an Instrument Status message for the item to current subscribers.
- Automatic solicitation of repair from the nearest upstream cache (or from the owning MDD instance).
  - On detection of loss of integrity of the message stream for an item, a repair request is sent upstream.
  - A receiving upstream cache responds by publishing a full expression of state for the item to be repaired.
  - Duplicate repair requests received from subordinate clients are eliminated to avoid NAK implosion.
  - Repair messages are sent only to those clients subscribing to the item being repaired.
  - Notification of completion of a repair is by publication of an Instrument Status message for the recovered item to subscribers.

5. MDN Generation of Refresh Requests

If loss of integrity of a message stream is detected for a topic during an update of the Site Reference Cache, a refresh request is automatically generated and published. This request is published to a MDN host specifically configured to manage and route requests to “owning” MDD instances (specifically to the Normalizer process that owns the Master Reference Cache for the required item/topic).

6. Routing of Refresh Requests

A configured connection name identifies a MDN host group to be used by a MDN process. The MDN host group will identify one or more virtual MDN hosts, where each virtual MDN host is instantiated as a specific multicast channel. In addition, the host group will identify a “real” MDN host specifically configured to manage and route requests published by a MDN process. This real MDN host is instantiated as a connection to a MDN process (MDN Req). The connection will use TCP/IP as the configured transport protocol.

Request messages are published to a set of well-known topics. These topics use a syntax which identifies the content set “owned” by a Normalizer (as published on a specific multicast channel). This allows each Normalizer process to subscribe to receive requests for content sets that it owns.

A Normalizer process responds to a refresh request by publishing a message set representing the current state for the item.

7. Listener Thread

A MDN process Listener thread will be configured to listen for client application connection requests on a configured(well-know) address and port. When a Listener thread accepts a connection, it will spawn a Client Receiver thread and pass the connection to the new thread.

The iX-Api will be embedded in any client application connecting to a MDN process. Configuration tables distributed as part of the iX-Api include specification of target connection points. The transport protocol for client application connections will be TCP/IP.

8. Client Receiver Thread

A Client Receiver thread [CRCV] is created by a Listener thread to manage a specific client application session. As part of initialization, the Client Receiver thread will open an interprocess queue which inherits the socket connection passed by the Listener thread.

As part of the opening of the queue, a Queue Driver thread [QD] is also created. The Queue Driver manages delivery of messages to be forwarded from a MDN process to a client application. [A Queue Driver thread (QD) is a utility thread, which decouples an Injector thread from the interprocess communication protocol used to transport messages from one process to the next.]

8.A. Functionality

A Client Receiver thread receives messages sent by a client application. The following messages may be sent by a client application and result in execution of the steps described by the Client Receiver thread:

Logon	• Client application submits a logon request including a user identification string and a password. • Client receiver thread creates an entry in the Subscription Cache for the client. • An acknowledgement is returned to the client application.
Logoff	• Client application submits a logoff request to terminate a session. • Client receiver thread deletes entries in the Subscription Cache for the client and topics subscribed to be the client. • An acknowledgement is returned to the client application.
Subscribe	• Client application submits a subscription request specifying the topic required. • Client receiver thread creates an entry in the Subscription Cache for the client and topic. • An acknowledgement is returned to the client application. • The Site Reference Cache is accessed for the topic being subscribed. • The concluding action taken by the client receiver thread is dependent upon the record state (see below).
Unsubscribe	• Client application submits a request to remove a subscription the topic to be removed. • Client receiver thread deletes entries in the Subscription Cache for the client and topics subscribed to be the client. • An acknowledgement is returned to the client application.
Refresh	• Client application submits a request to refresh a current subscription specifying the topic required. • The Site Reference Cache is accessed for the topic being subscribed. • The concluding action taken by the client receiver thread is dependent upon the record state (see below).

When a client connection terminates, either by request or because of transport error or because a logon request was rejected, the Client Receiver thread and partner Queue Driver will exit. All subscriptions for that client connection will be removed from the Subscription Cache.

A client application (specifically, the iX-Api embedded within the client application) will automatically recover a dropped connection by resubmitting a connection request and reestablishing subscriptions. A reestablished client application connection is handled as a new connection by the MDN process. No context is saved or passed forward from the previous connection within the MDN process.

Responding To Subscription Or Refresh Requests

When a Client Receiver thread processes either a subscription request or a refresh request, a full representation of the current state for the selected topic is to be returned to the client. The Client Receiver thread accesses the Site Reference Cache for the topic to be subscribed or refreshed.

The steps then taken are dependent upon the current state of the record in the cache as follows:

Initial	The item record has been created as a result of receipt of the new subscription but is empty, no messages have been received. The Client Receiver initiates publication of a Refresh Request which is forwarded to the Normalizer owning the topic. A Normalizer process responds to a refresh request by publishing a message set representing the current state for the item. The message set is forwarded to the client application.
Firm	The record fully represents the current status for the item/topic based on the received message stream. A message set is constructed from the record. The message set is forwarded to the client application.
Indicative	An error has been detected in the received message stream. Repair has already been initiated. A message set is constructed from the record, including an Instrument Status message with a status of Indicative The message set is forwarded to the client application. A Normalizer process responds to a previously published refresh request by publishing a message set representing the current state for the item. The message set is forwarded to the client application.
Recovering	One or more (but not all) messages have been received in a message set that is in the process of conveying a full representation of state for the item. A marker is set on the record indicating that a refresh message set is pending for the requesting client. Once recovery is completed successfully the state will transition to Firm. A message set is constructed from the record and forwarded to the client application.

Building	One or more (but not all) messages have been received in a message set that is conveying an update event. A marker is set on the record indicating that a refresh message set is pending for the requesting client. Once the set is completed successfully the state will transition to Firm. A message set is constructed from the record including an “Instrument Status” message with a status of “Indicative.” The message set is forwarded to the client application

9. Subscription Cache

A MDN process is configured to contain an in-memory cache termed a “Subscription Cache.” The primary function of a MDN Subscription Cache is to maintain a record of all subscriptions established for current client sessions.

10. MDN Monitoring and Control

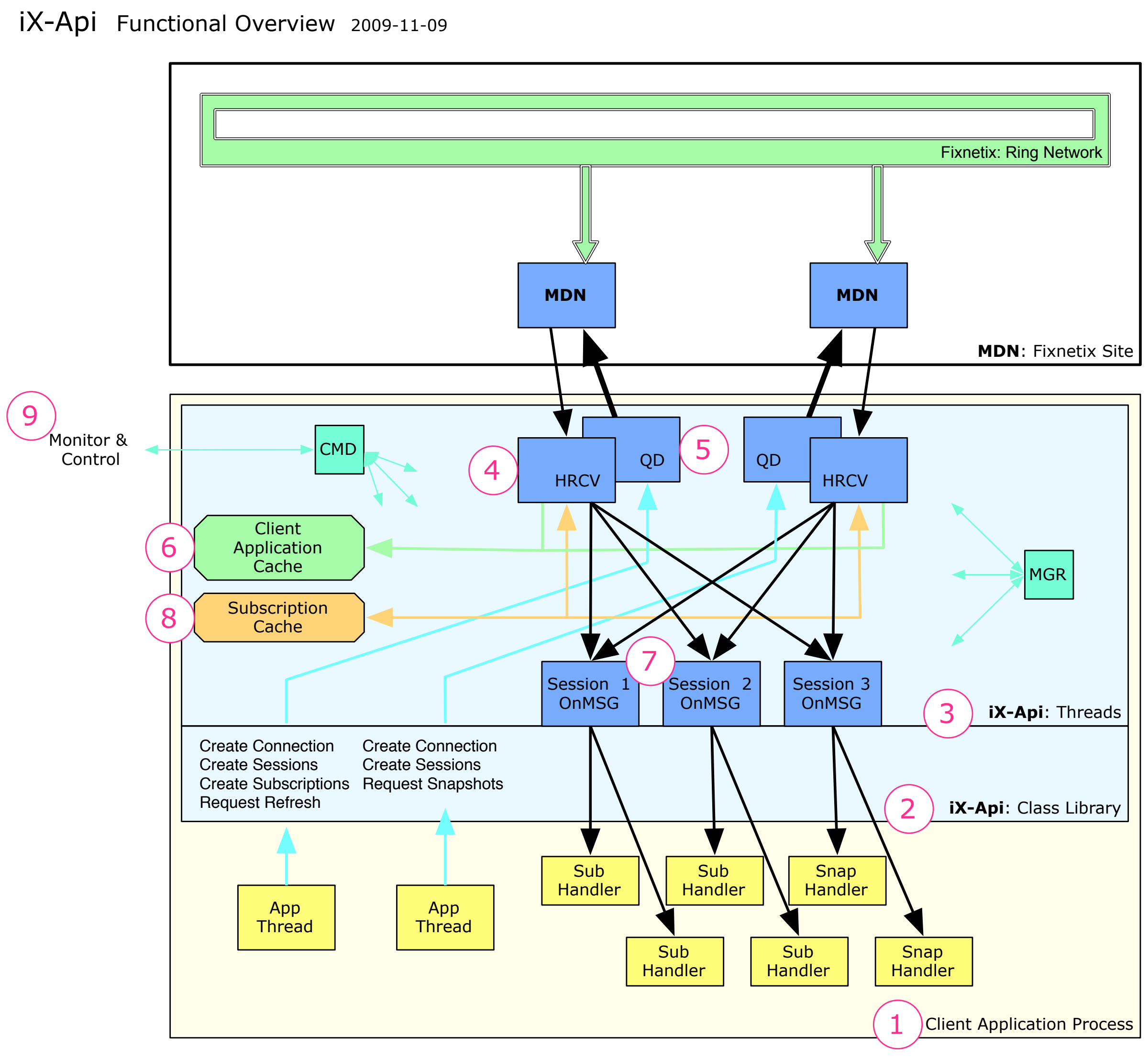
Each MDN process will include a configured Command thread (CMD). [A Command thread is a utility thread which serves to enquiry and control requests submitted from a Monitoring subsystem, from scheduled operational control scripts or from command line tools. A Command thread within a process listens and accepts for connections, accepts requests, invokes embedded command functions, returns results sets to the requester and closes connections on completion.]

A MDN process will be configured to support the following command request types:

cache_stats	Monitor cache metrics
client_stats	Control and monitor client application connections
hash_stats	Monitor cache index metrics
host_stats	Control and monitor host connections
queue_stats	Control and monitor queue objects, and return queue metrics
task_stats	Control and monitor threads, and return thread metrics
transport	Control and monitor specific transport protocols underlying queues
Send Refresh	Generate and publish a message set reflecting the current state for an identified instrument or instruments
View Record	Read the cache for an identified instrument and return a message set reflecting current values

iX-Api

This section illustrates and describes the internal structure and functionality of a client application process with an embedded iX-Api. The iX-Api provides a high-level interface allowing client applications to establish connections, maintain sessions, maintain subscriptions, issue snapshot requests, issue subscription refresh requests and set message handler methods for processing received messages.



The above diagram illustrates primary functionality only. No attempt is made to show redundant components which address fault tolerance requirements.

The notes below reference the numeric labels on the diagram.

1. Client Application Process

The diagram illustrates the internal structure and data flow for a single client application which consumes real-time market data using the iX-Api. A client application is a process which executes on a single server either at a client site or co-located at a Fixnetix site.

Each client application will connect to one or more MDN processes located at Fixnetix sites. Each MDN provides generic publish/subscribe messaging services and implements distribution of real-time market data to connected client applications.

In general client applications using the iX-Api can be classified as subscribing clients. A subscribing client will typically maintain an “interest set” of subscriptions that identify the population of tradable instruments for which security pricing data is required. The act of establishing a new subscription results in an initial “state” message being delivered to the client application defining the current pricing data for the subscribed instrument. Update messages are then published and distributed to the client application in real-time to report market events for the item as they occur. Subscribing clients typically use asynchronous subscriptions and specify callback handler methods to be invoked on receipt of messages for the subscribed to instruments.

2. iX-Api Class Library

The iX-Api is implemented as a C++ or Java class library. Subscribing client applications use either the Java or C++ versions of the iX-Api. There are equivalent classes and methods within both versions.

Installation of the iX-Api shared object libraries and associated configuration tables is required on the target client platform. At runtime, the required library is loaded into the client application process context.

iX-Api::Connection

A Connection object is the abstraction of the iX-Api as seen by client applications that manages communication between a client application and a message source and/or message destination. A client application creates a connection identifying a user name, a target connection name and the location of required iX-Api configuration tables. These parameters may be provided explicitly, or may be configured and referenced implicitly by means of external environment variable settings (see the iX-Api Release Notes document).

Multiple threads within a client process may share a single connection object. This may be accomplished by having each client thread construct a connection object by specifying the same argument values when invoking a constructor. Although no built-in rules are implemented to enforce this restriction, the generation of multiple connections using different argument values is not supported within a single client process and the results are not predicted.

iX-Api::Session and iX-Api::Subscriber

A Session object defines a serial order for messages consumed by the client application. Multiple sessions may be created per connection.

iX-Api::Subscriber

Multiple Subscriber objects can be created on a session. Creation of a subscriber (a.k.a subscription) includes identification of an item of interest (a topic e.g. a tradable security) and identification of a handler function to be invoked when messages are received which were published for that item. Specification of a handler function defines a subscription as being an asynchronous subscription. Within the iX-Api, one thread is responsible for invoking the handler methods for all asynchronous subscriptions created on a session.

The order of processing of messages received for subscriptions on a single session is consistent with the order of publication of those messages by a message source. The order of processing of messages between multiple sessions is not guaranteed due to the independent execution of each sessions call back thread.

A client application can enable concurrent processing of received messages by creating multiple sessions on a connection and distributing subscriptions with specified call back handlers across those sessions. Therefore, the client application is responsible for ensuring concurrently invoked call back functions do not conflict.

iX-Api::Snapshot

A client application can create a snapshot handler on a session. All snapshot requests issued on the session for which a handler has been specified will cause snapshot response messages to be delivered to the specific handler function. This allows snapshot response processing to be performed in an asynchronous mode rather than in a synchronous mode.

Successive snapshot responses may be processed concurrently. This is controlled when creating the snapshot handler function. Note that all messages in a snapshot response message set will be processed serially and in order even if multiple snapshot handler instances are specified on a session.

3. iX-Api Threads

The first client application thread which creates a connection object will trigger the initialization of the iX-Api runtime environment within the context of the client process. The iX-Api executes multiple threads that cooperate to implement the required functions. Initialization of the iX-Api includes starting a number of iX-Api threads and the initialization of one or more TCP/IP socket connections from the client process to target MDN connection points.

Each iX-Api instance is defined by a configuration specified in configuration table files. The index into these configuration files is the target connection name specified when constructing the connection object.

Manager Thread

A Manager thread (MGR) is responsible for overall control of the iX-Api. A manager thread is created as part of initialization. It manages and coordinates asynchronous events such as subscription requests issued by the client application, host connection failures, etc.

Events are reported to the manager by other threads and iX-Api methods, such as when a subscriber is created. The manager thread takes actions depending upon the event reported and the current state of the iX-Api. Actions might include reestablishing host connections, forwarding subscription notifications, forwarding refresh requests, etc.

4. Host Receiver Threads

A client application process will require access to one or more discrete market data sets. A market data set will generally include all instruments reported for one or more market segments of a trading venue or exchange.

When a client application creates a connection, it will specify a connection name value and will identify required configuration tables (which are part of the iX-Api distribution for a client). Within the configuration tables, a connection name identifies a MDN host group. A MDN host group will identify one or more real MDN hosts. Each identified MDN host is located at a Fixnetix site and will itself be configured to support one or more market data sets. During iX-Api initialization, a client application process will establish a TCP/IP socket connection to one or more MDN hosts which are defined in the host group identified by the connection name used to create the connection.

The diagram illustrates a client application process with a iX-Api which is configured with 2 host receiver threads, each connecting over a TCP/IP socket connection to a MDN host.

Functionality

A host receiver thread receives and processes messages routed by a MDN for topics subscribed to by a client application. The following steps will be executed by a host receiver thread on receipt of a published message which is received for a currently active subscription:

- Read the subscription cache entry for the topic on which the received message was published;
- Identify each session that has a current subscription for the topic;
- Enqueue the message to each identified session (for processing by the sessions onMessage thread);
- Read the client application cache record for the topic
  OR
  create a new record if one does not already exist;
- Verify that the received message is consistent with the current context held for that topic;

- Update the current context held for the topic to reflect the message received;
- Commit the context back to the site reference cache; and
- If loss of integrity was detected:
    - Initiate a refresh request for current state for the topic to be repaired;
    - Construct an “Instrument Status” message with a status of “Indicative” for the topic; and
    - Enqueue the Instrument Status message to each session that has a current subscription for the topic.

5. Host Queue Driver

A host queue driver thread is created as a result of a host receiver thread opening the host queue. A host queue driver thread is a utility thread which manages the transmission of messages to a MDN host. Messages to be sent to the MDN host may be generated by the manager thread. These are administration messages such as subscription requests or connection “logon” or logoff requests.

A client application can publish messages on a session by calling the send function. Such messages are forwarded to the host queue driver for transmission to the MDN host.

6. Client Application Cache

A client application process’ iX-Api is configured to contain an in-memory cache, termed a Client Application Cache. The primary function of a Client Application Cache is to support verification of the integrity of the message stream received for each subscribed topic (item).

If loss of integrity of a message stream is detected then a refresh request is automatically generated and published. A refresh request is published to the MDN host that “owns” the topic.

Client Reference Cache

A client application cache may be selectively enabled to hold a full representation of current state for each active topic (i.e. for each topic for which a subscription currently exists). If this is enabled, the client application cache becomes a reference cache. It is referenced on any new subscription, on any subscription refresh request, and on any snapshot request generated by the client application (using iX-Api methods).

If the topic requested is present in the client reference cache (and is current / firm), the current state for the item is “published.” Full representation of current state is published as a message set. The message set is delivered to the client application session in the correct sequence relative to messages (message sets) already published on the topic and to be published on the topic.

Certain client applications may require ad-hoc snapshots of current values for selected items. In this case subscriptions can be established for the required items, but message delivery to a specified callback handler function can be disabled. In this mode, the subscription will result in the receipt of messages published to the topic and maintenance of current state within the client reference cache. When required, the client application may issue snapshot requests to retrieve current state for an item. In this case state is retrieved from the client reference cache.

Client Application Cache Maintenance

A record in the cache is maintained for each topic for which a subscription currently exists. A record contains meta data defining a current context for the topic. Optionally, a record also contains the “last” instance of each ML2Message received for the topic if the client application cache has been enabled to hold state.

A record for a topic is updated in the cache on receipt of each message published to the topic. An update includes evaluation of the integrity of the message stream received for the topic. This is based on the current context held in the cache record for the topic and meta-data contained in the message received.

The meta data used for verification of the message stream includes the following message attributes:

Revision	A revision number is maintained for each item by the MDD message source which “owns” that item.
Set Size	Every ML2Message is member of a message set. A message set may contain one or more messages. The set size in a message defines how many messages are in the set of which this message is a member.
Set Identity	If a message set contains more than one message then a set identity value will be carried by all messages in the set.
Set Position	If a message set contains more than one message then set position value identifies the position of each message in the set.

Client Application Cache States

A record in the cache is maintained for each topic on which received messages have been published. Each record is managed independently based on interpretation of the message stream received. At any time a record (that is the information content within the record held for a specific item or topic) is considered to be in one of the following states:

Initial	The item record has been created as a result of receipt of a subscription, but is empty, no messages have been received.
Firm	The record fully represents current status for the item/topic, based on the received message stream.

Indicative	An error has been detected in the received message stream. On transition to this state a refresh request is published upstream to solicit a repair transmission for the topic. Also an "Instrument Status" message is constructed for the topic with a status of "Indicative" This message is forwarded the for output to each client subscribing to the topic. In this way all subscribing clients are alerted to the loss of integrity.
Recovering	One or more (but not all) messages have been received in a message set that is in the process of conveying a full representation of state for the item. Once completed successfully the state will transition to Firm.
Building	One or more (but not all) messages have been received in a message set that is conveying an update event. Once completed successfully the state will transition to Firm.

7. Session OnMessage Thread

An onMessage thread is created when the first asynchronous subscriber is created on a session. Creation of the subscriber includes specification of the handler function (or onMessage method) to be invoked to process messages received for the subscription.

An onMessage thread is specific to each session. A session's onMessage thread receives messages which the host receiver puts on that session's received message queue. This queue acts to serialize processing of all messages received on all asynchronous subscriptions on the session.

Such messages are held pending processing by the onMessage thread associated with the session. The onMessage thread is responsible for consuming messages on the sessions queue and invoking the handler function defined for each asynchronous subscription.

8. Subscription Cache

A client application process iX-Api is configured to contain an in-memory cache, termed a Subscription Cache. The primary function of a Subscription Cache is to maintain a record of all subscriptions established for current client sessions.

9. MDN Monitoring and Control

Each client application process iX-Api may include a configured Command thread (CMD). A Command thread is a utility thread, which serves enquiry and control requests submitted from a Monitoring subsystem, from scheduled operational control scripts, or from command line tools. A Command thread within a process listens and accepts for connections, accepts requests, invokes embedded command functions, returns results sets to the requester and closes connections on completion.

A client application iX-Api may be configured to support the following command request types:

cache_stats	Monitor cache metrics
hash_stats	Monitor cache index metrics
host_stats	Control and monitor host connections
queue_stats	Control and monitor queue objects, and return queue metrics
task_stats	Control and monitor threads, and return thread metrics
transport	Control and monitor specific transport protocols underlying queues
Send Refresh	Generate and publish a message set reflecting the current state for an identified instrument or instruments
View Record	Read the cache for an identified instrument and return a message set reflecting current values