GTP Market Data Adapter. Overview

Group Ticker Plant Connectivity Overview

Below is a brief overview of the Group Ticker Plant market data connectivity.

Services

The GTP market data adaptor currently supports two services.

Level 1 service provides the best bid and offers data, the trade report messages, and the group statistics information. Level 2 service provides instrument definition, instrument status, order book (market by order), opening/closing statistics, and trades data.

Market Data Groups

A market feed is load-balanced by market data groups. A market data group includes one or more instruments. Each instrument is assigned to just one market data group.

Sites

A market data group includes two sites disseminating identical data – the primary (A) and secondary (a).

Channels

A site includes up to three channels.

Real-Time UDP channel disseminates market data via a multicast group.

Replay TCP channel is used to recover from a small message loss in the Real-Time channel. The channel allows the recipient to request retransmission of a range of messages already published in the Real-Time channel.

Optional Recovery TCP channel is used to recover from a large message loss in the Real-Time channel or a late join to the Real-Time channel. The Recovery channel allows the recipient to request a snapshot of the current market data.

Accounts

The recipient must have a valid account to login to the Replay and Recovery channels.

GTP Market Data Adaptor Overview

GTP Market Data Adaptor is a module that allows communicating with the GTP market feeds.

Interfaces

Two sets of interfaces are exposed to the client.

One set abstracts the client from GTP Market Data Adaptor implementation.

- Lse::Gtp::Application
- Lse::Gtp::ApplicationListener
- Lse::Gtp::Service
- Lse::Gtp::ServiceListener

The other set abstracts GTP Market Data Adaptor from connectivity implementation. The client can supply its own connectivity implementation. If not supplied by the client the default implementation based on Boost.Asio is created and used by the GTP Market Data Adaptor.

- Lse::Gtp::ConnectionManager
- Lse::Gtp::UdpConnection
- Lse::Gtp::UdpConnectionListener
- Lse::Gtp::TcpConnection

Configuration

Group Ticker Plant Market Data Adaptor is configured via the XML configuration file that sets up the application options, the services, the service options, filtering by segments, and instruments.

Application

Application stores services and provides access to them by index and by name. The application also stores resources shared across services such as a connection manager.

It is possible to have several applications created at the same time. The client can choose to have either a single application for all the services of all the market feeds or have a separate application for each service. All the IO operations of all the services of an application are processed by the same connection manager.

Service

Service provides common tasks such as Real-Time channel management, Replay channel management, Recovery channel management, statistics management, status management, etc.

GTP Service

GTP service is a specialized service that processes tasks such as unit parsing, message replay, and message recovery.

Parser

GTP parser defines structures for the units and the messages and the general-purpose functions to construct, parse, validate, and dump the units and messages.

Sequence Processor

Sequence processor processes message losses, message reorders and message duplications that may occur in the Real-Time channel.

The sequence processor automatically arbitrates between feeds A and an of the Real-Time channel to reduce the probability of message loss.

Out of order messages are either filtered out or cached in a queue until the gap is filled from the same feed or the second feed or the Replay channel.

In the case, a message gap is detected the sequence processor starts a replay procedure. The sequence processor automatically arbitrates between the primary and secondary sites of the Replay channel.

The sequence processor processes late join and detects the inter-day and intra-day resets.

In the case of a late join or a reset detected the sequence processor executes a recovery procedure. The sequence processor automatically arbitrates between the primary and secondary sites of the Recovery channel.

Service Statistics

Two types of service statistics are maintained - connection statistics and date statistics.

The connection statistics are reset on each service connect operation.

The date statistics are persisted and is reset on each new date.

Service Status

The status of each service channel is tracked and available via the service connection statistics.

Real-Time channel is considered inactive if there are no messages received within certain time interval.

The Replay channel is considered inactive if the last replay procedure failed. If there are no gaps detected within certain time interval the Replay channel is probed to update the Replay channel status.

The recovery channel is considered inactive if the last recovery procedure failed.

Asio Connection Manager

Asio connection manager serves as a factory for UDP and TCP connections.

Asio connection manager manages a pool of threads and a completion port to process the results of asynchronous operations of an arbitrary (unlimited) number of concurrent UDP and TCP connections.

The size of the thread pool is configurable.

Asio UDP Connection

Asio UDP connection asynchronously receives UDP packets from a multicast group.

Asio TCP Connection

Asio TCP connection synchronously sends and receives data to and from a TCP endpoint.

Logging

All the data sent and received can be optionally logged at three levels for later analysis and troubleshooting.

- logClientMessages – log client messages to text log files
- logMessages – log UDP/TCP connection messages to text log files
- logData – log UDP/TCP connection data to binary log files

Error, warning, and trace events are logged to the FIXAntenna log subsystem.

Appendix A. Interfaces

Lse::Gtp::Application

```
// Releasing
virtual void release() const = 0;

// Name
virtual std::string getName() const = 0;

// Opening / closing
virtual bool opened() const = 0;

// Initializes the application, creates and adds services from the configuration file.
// The services are created in the disconnected state and should be connected explicitly.
// applicationListener should stay alive until close() is called.
virtual void open(ApplicationListener* applicationListener) = 0;

// Removes the services and finalizes the application.
// Should only be called if open() succeeded.
virtual void close() = 0;

// Services

// Returns the number of existing services.
virtual size_t getServiceCount() const = 0;

// Returns a service by index.
virtual Service* getService(size_t index) = 0;

// Finds a service by name. Returns 0 if the service does not exist.
virtual Service* findService(const std::string& name) = 0;

// Creates and adds a new service to the application.
// The service is created in the disconnected state and should be connected explicitly.
virtual Lse::Gtp::Service* addService
(
    const std::string& name,
    const ServiceOptions& serviceOptions
) = 0;

// Removes the service from the application and destroys it.
virtual void removeService(const std::string& name) = 0;

// Removes all the services from the application and destroys them.
virtual void removeAllServices() = 0;
```

Lse::Gtp::ApplicationListener

```
// Services

// Called after a service is created and added to the application
virtual void onServiceAdded(Application* application, Service* service) = 0;

// Called before a service is removed from the application and destroyed
virtual void onServiceRemoving(Application* application, Service* service) = 0;
```

Lse::Gtp::Service

```
// Application
virtual Application* getApplication() const = 0;

// Name
virtual std::string getName() const = 0;

// Returns the state of the service
virtual bool connected() const = 0;

// Connects the service to the mutlicast group.
// serviceListener should stay alive until disconnect() is called.
virtual void connect(ServiceListener* serviceListener) = 0;

// Disconnects the service from the mutlicast group.
// Should only be called if connect() succeeded.
virtual void disconnect() = 0;

// Statistics

// Returns service statistics snapshot.
// connectionStatistics can be 0.
// dateStatistics can be 0.
virtual void getStatistics
(
    ServiceConnectionStatistics* connectionStatistics,
    ServiceDateStatistics* dateStatistics
) const = 0;
```

Lse::Gtp::ServiceListener

```
// Messages

// Called on service reset. The client should reset all its message derived data.
virtual void onReset(Service* service, ResetReason resetReason) = 0;

// Called on GTP message received.
virtual void onGtpMessage(Service* service, const Lse::Itch::MessageHeader* messageHeader) {}

// Called when either the recovery or the replay request has been completed.
virtual void onRequestComplete(RequestType request, bool success) {}

// Called on a service resynchronization. Informs the client about an out-of-order message
virtual void onResync(Service* service) {}
```

Lse::Gtp::ConnectionManager

```
// Releasing
virtual void release() const = 0;

// Returns the state of the connection manager
virtual bool opened() const = 0;

// Initializes connection manager
virtual void open() = 0;

// Finalizes connection manager
virtual void close() = 0;

// Connections

// Creates a UDP connection.
// The connection is created in the disconnected state and should be connected explicitly.
// The connection should be destroyed with UdpConnection::release().
// It is possible to have several connections created at the same time.
virtual UdpConnection* createUdpConnection(const std::string& name, bool logData) = 0;

// Creates a TCP connection.
// The connection is created in the disconnected state and should be connected explicitly.
// The connection should be destroyed with TcpConnection::release().
// It is possible to have several connections created at the same time.
virtual TcpConnection* createTcpConnection(const std::string& name, bool logData) = 0;

// Tasks

// Posts the task for asynchronous execution in the context of another thread.
// connectionManagerListener should stay alive until ConnectionManagerListener::onTask() is called.
virtual void postTask
(
    const std::string& name,
    void* parameters,
    ConnectionManagerListener* connectionManagerListener
) = 0;
```

Lse::Gtp::UdpConnection

```
// Releasing
virtual void release() const = 0;

// Connection manager
virtual ConnectionManager* getConnectionManager() const = 0;

// Name
virtual std::string getName() const = 0;

// Returns the state of the connection
virtual bool connected() const = 0;

// Connects to the multicast group.
// connectionListener should stay alive until disconnect() is called.
virtual void connect
(
    const std::string& localAddress,
    const std::string& groupAddress,
    System::u16 port,
    UdpConnectionListener* connectionListener
) = 0;

// Disconnects from the multicast group.
// Should only be called if connect() succeeded.
virtual void disconnect() = 0;
```

Lse::Gtp::UdpConnectionListener

```
// Data

// Returns the size of data to receive and pass to onDataReceived() or 0.
virtual size_t getDataSize(UdpConnection* connection, const void* data, size_t size) = 0;

// Called on data received.
virtual void onDataReceived(UdpConnection* connection, const void* data, size_t size) = 0;

// Called to notify no more data is available.
virtual void onNoData(UdpConnection* connection) = 0;
```

Lse::Gtp::TcpConnection

```
// Releasing
virtual void release() const = 0;

// Connection manager
virtual ConnectionManager* getConnectionManager() const = 0;

// Name
virtual std::string getName() const = 0;

// Returns the state of the connection
virtual bool connected() const = 0;

// Connects to the remote endpoint.
virtual void connect(const std::string& localAddress, const std::string& remoteAddress, System::u16 port) = 0;

// Disconnects from the remote endpoint.
// Should only be called if connect() succeeded.
virtual void disconnect() = 0;

// Sending / receiving

// Sends the data synchronously.
virtual void sendData(const void* data, size_t size) = 0;

// Receives the size of data synchronously.
virtual void receiveData(void* data, size_t size) = 0;
```

Appendix B. Configuration file example

```
<?xml version="1.0"?>
<Application Name="GTP" LocalAddress="10.4.233.34" LogMessages="true" LogClientMessages="true" LogData="false"
ThreadCount="8">
  <Services>
    <Service Name="Level 1" UserName="INEL01" ConnectInterval="3000" ConnectAttempts="3" MaxConnectAttempts="6"
ReplayDelay="0">
      <RealTime>
        <Primary Address="224.0.112.9" Port="21679"/>
        <Secondary Address="224.0.112.10" Port="21680"/>
      </RealTime>
      <Replay DateLoginLimit="1000" DateRequestLimit="1000">
        <Primary Address="196.216.153.13" Port="21681"/>
        <Secondary Address="196.216.153.19" Port="21681"/>
      </Replay>
      <Recovery DateLoginLimit="500" DateRequestLimit="500">
        <Primary Address="196.216.153.13" Port="21682"/>
        <Secondary Address="196.216.153.19" Port="21682"/>
      </Recovery>
      <Instruments>
        <Instrument Id="6656671337" />
      </Instruments>
    </Service>
    <Service Name="Level 2" UserName="INEL01" ConnectInterval="3000" ConnectAttempts="3" MaxConnectAttempts="6"
ReplayDelay="0">
      <RealTime>
        <Primary Address="224.0.112.5" Port="21651"/>
        <Secondary Address="224.0.112.6" Port="21652"/>
      </RealTime>
      <Replay DateLoginLimit="1000" DateRequestLimit="1000">
        <Primary Address="196.216.153.19" Port="21653"/>
        <Secondary Address="196.216.153.12" Port="21653"/>
      </Replay>
      <Recovery DateLoginLimit="500" DateRequestLimit="500">
        <Primary Address="196.216.153.19" Port="21654"/>
        <Secondary Address="196.216.153.12" Port="21654"/>
      </Recovery>
      <Segments>
        <Segment Code="TEST" />
      </Segments>
    </Service>
  </Services>
</Application>
```