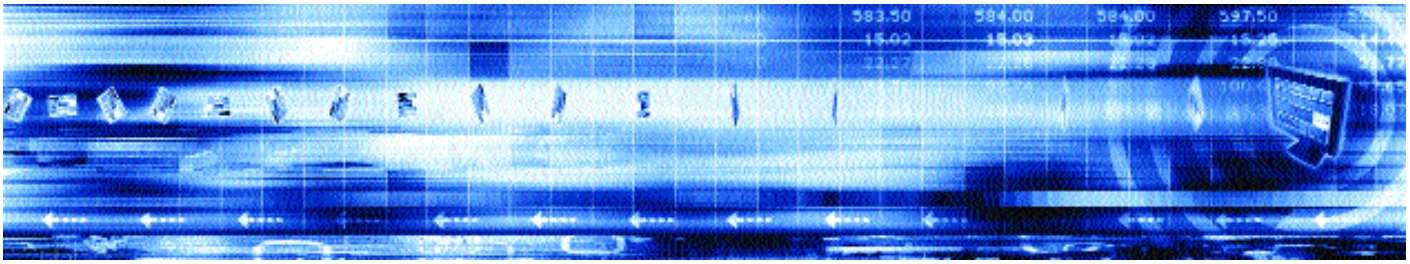


# CAPLIN STREAMLINK 4.2

## Technical Overview

**June 2006**

# 1 Preface

## 1.1 What this document contains

This document describes Caplin StreamLink and its place in the Caplin real-time data architecture, as well as RTTP and the Caplin Liberator server. It includes instructions on how to configure the Liberator server and how to subscribe to and publish data. Ths document also contains a description of the advanced debugging functionality new to version 4.0.

## 1.2 Who should read this document

This document is intended for programmers who need to add RTTP streaming capability to their applications.

## 1.3 Typographical conventions

This document uses the following typographical conventions to identify particular elements within the text.

| Type | Use |
|---|---|
| **Arial Bold** | Function names and methods.<br>Other sections and chapters within this document. |
| *Arial Italic* | Parameter names and other variables. |
| *Times Italic* | File names, folders and directories. |
| `Courier` | Program output and code examples. |
| ❖ | Information bullet point |
| ■ | Instruction |

## 1.4 Feedback

Customer feedback can only improve the quality of Caplin product documentation, and we would welcome any comments, criticisms or suggestions you may have regarding this document.

Please email your thoughts to **documentation@caplin.com**.

# 2 Overview

StreamLink is an API that provides everything programmers need to add RTTP streaming capability to their applications. Java, C++ and .NET versions of the StreamLink API are available.

It enables access to the full range of functionality of the RTTP transport mechanism, including smart tunnelling, data health checking, advanced flow control and persistent virtual connection.

StreamLink automates the functions that most applications require and provides default processing of responses from Caplin Liberator which are easy for a developer to extend. You only need to write code to perform custom processing in specific instances—everything else can be handled by StreamLink.

StreamLink allows you to send and receive real-time data to or from any application or Java applet as part of a stand-alone or browser-based trading application. When used to write Java applets, it allows the widest possible range of browsers to be supported. Java 1.1 support is all that is required, and it is compatible with Netscape 4.0 and above, and Internet Explorer 4.0 and above. StreamLink for Java does not require applets to be signed.

When used for stand-alone applications, StreamLink provides RTTP connectivity for development on any platform.

In order to use StreamLink you need access to a Caplin Liberator, and Java 1.1 or above for the Java API. StreamLink for C++ is a cross platform library that is supported and tested on the following platform/environment combinations: Linux: Redhat 9 GCC 3.2.2, Solaris: Solaris 8 Forte 8, Windows: Visual Studio .NET. Other configurations may work but are untested. StreamLink for .NET is a standard .NET managed DLL that can be used within any of the .NET environment languages.

## 2.1 Documentation

The StreamLink for Java installation includes Javadoc documentation, which gives in-depth technical descriptions of all the methods and classes within the *rtjl.jar* file.

The StreamLink for C++ installation includes Doxygen documentation, which gives in-depth technical descriptions of all the methods and classes in the library. The Unix targeted release also includes standard man pages.

The StreamLink for .NET installation includes documentation in the form of HTML pages, which gives in-depth technical descriptions of all the methods and classes in the library.

All the kits include a release-note specific to the platform. Figure 2-1 shows Caplin's platform architecture.

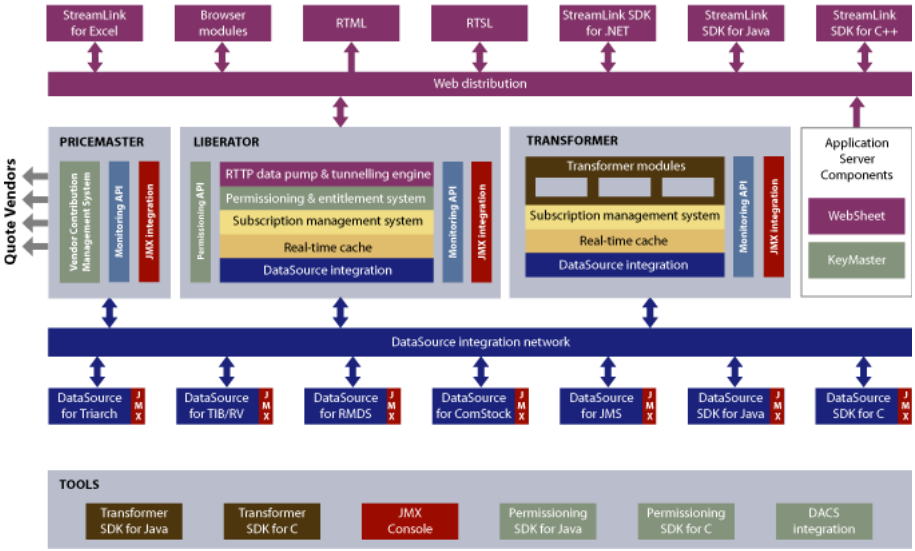



***Figure 2-1: Caplin's platform architecture***

Figure 2-2 shows a simplified view of the Caplin architecture and highlights StreamLink and its place in the platform.

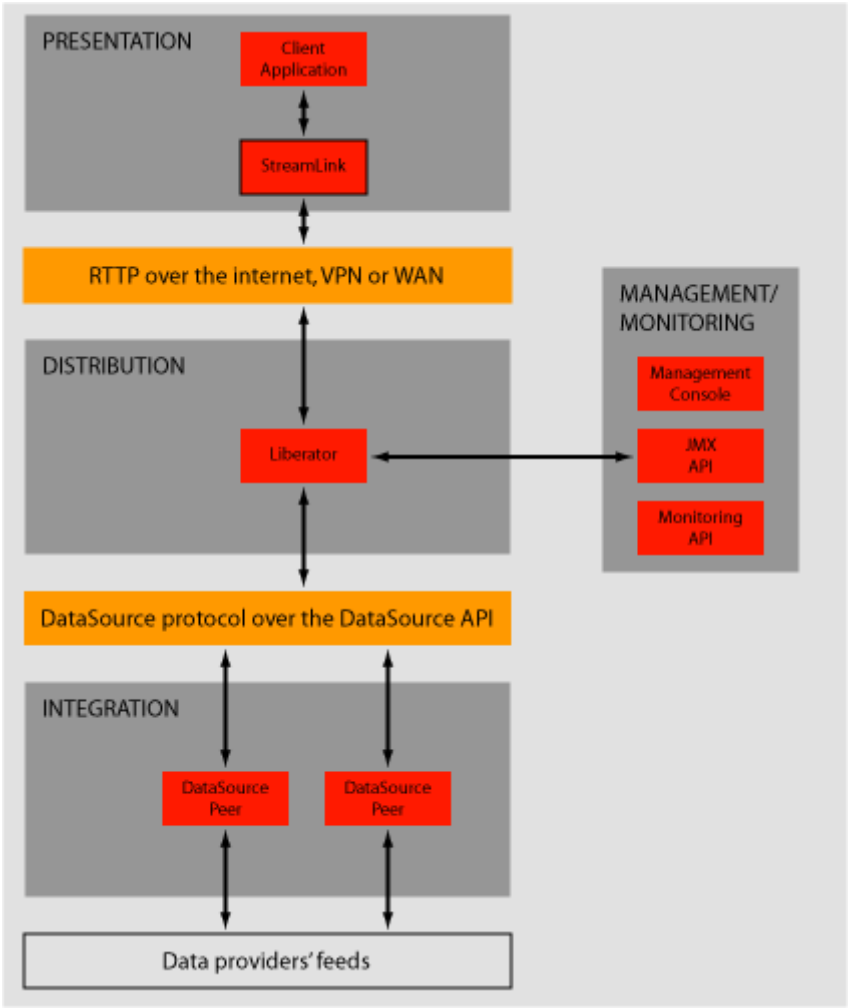



***Figure 2-2: StreamLink's place in the Caplin architecture***

# 3 How StreamLink works

Figure 3-1 below shows the main steps that make up a request for real-time data.


***Figure 3-1: Requesting data using StreamLink***

There are three possible outcomes after an initial successful call of the request function:

- Caplin Liberator returns the cached value of an object.
- Caplin Liberator returns a message indicating the object has been actively requested from Caplin Liberator's data sources.

- Caplin Liberator returns a message indicating a problem exists which means the data cannot be accessed at the moment.

If there is no problem, StreamLink will start to receive updates or be notified of problems as they occur.

StreamLink uses an abstract class which catches all RTTP-related events. Descendants provide all the functionality necessary to:

- connect to Caplin Liberator.
- route messages to the processing functions of the base class.
- receive all responses and updates from Caplin Liberator.
- provide access to functions that send messages to Caplin Liberator.
- monitor the server connection and reconnects if it is lost.
- monitor individual symbols or symbols and parameters on Caplin Liberator.

## 3.1 Sending messages to Caplin Liberator

Most messages ask for data or send data to Caplin Liberator. All access to Caplin Liberator is controlled by interfaces.

For the majority of applications there is only one method necessary to provide the main request, discard, create, delete and contribute functions: to request an object simply create a target object and pass it to the request function. As with all StreamLink functions, if an error occurs while processing the request, or if the target object doesn't exist, an exception will be thrown.

***Note:*** *An active data source is one that will keep track of which objects have been requested and send updates for those objects only. When using StreamLink to request data from an active source, the success or failure of the request may not be known immediately. The relevant method will be notified when this is known.*

If the request is successful, the initial returned value and subsequent updates will be received by a suitable update method.

# 4 About the data

RTTP (Real Time Text Protocol) is a web protocol developed by Caplin Systems Ltd that implements advanced real time streaming for almost all types of textual information, including logical records, news and free-format pages, over internet protocol networks. It supports both client-server and peer-to-peer publish/subscribe models. It can be used as a simple point-to-point protocol over a LAN, but also supports reliable publishing to thousands of simultaneous users over the public internet.

RTTP is a very robust protocol that automatically 'tunnels' through firewalls and proxy servers. RTTP ensures high data quality irrespective of most network obstacles using persistent virtual connections (providing rapid and seamless recovery from transient connection loss) with smart/secure tunnelling and data health checking to monitor the 'freshness' of the information being received.

## 4.1 How RTTP data is organised

There are several types of RTTP object:

| | |
|---|---|
| Directory | Caplin Liberator uses directories which contain Page, Record, News headline and story, and Chat objects. Data can be held by an object in two ways: in one chunk and/or as name-value pairs. Whether a particular object stores information as a chunk and/or name-value pairs depends on the type of the object. Users of data streamed on RTTP can subscribe to a directory and receive updates when objects are created or deleted within that directory. |
| Page | A page is a free format piece of text made up of rows. RTTP supports any size of page up to 128 rows of 256 characters (typical sizes are 14 rows of 64 characters and 25 rows of 80 characters). |
| Record | A record is a means of storing and displaying information. Records are composed of fields which may not be of the same type: for example, a record containing equity data could have several price fields (e.g. the last traded prices) together with time and date fields, whereas an index record would have a price field but no bid or ask values. |

| | |
|---|---|
| News story | News stories do not need to be streamed on RTTP since they do not benefit from being real-time enabled (although a client might want to stream stories if the content can be changed or appended to as more news becomes available, and readers want to see updates to the story immediately). |
| News headline | The news headline, however, must be RTTP-enabled, so that if the user wants to read the story they can select that particular news item and use a more standard subscription mechanism to request the story. A request for a news headline object may contain a filter string which allows a client to limit the updates it receives based on a simple logical syntax. |
| Chat | RTTP chat objects allow users logged into Caplin Liberator to chat in real-time. Each chat object represents a virtual chat room for 2 or more users. To send a message to the channel, users contribute to chat objects. |

## 4.2 Symbols and fields

Most real time data handled by RTTP is identified by combinations of symbols and fields. The symbol is stored as the name of an object on Caplin Liberator.

- A symbol is a letter or sequence of letters used to identify a security. Symbols should always start with a "/". For example, "/DCX" is used for Daimler Chrysler Corporation, "/LO/VOD" for Vodafone trading on the London Stock Exchange, and "/MSFT" for Microsoft.
  The symbol you choose depends on the "symbology" being used by the data source. If you are running your own Caplin Liberator, this will by default be the same as the symbology of the data source to which it is connected. If you are using a third-party RTTP source, you should obtain a symbol directory from its owner.

- A field is a certain piece of information relating to the symbol. Typical fields are "Bid" (the bid price), "Ask" (the asking price) or "Cls" (for the previous day's closing price).
  As with symbols, the range of fields available for a particular financial instrument depends on the data source to which you are connected.

## 4.3 How StreamLink uses RTTP data

StreamLink provides a set of objects which enable you to subscribe to and modify the contents of Caplin Liberator's directories. These functions perform the following tasks:

| | |
|---|---|
| requesting | allows you to get the current value of an object (i.e. its data and/or name-value pairs) on Caplin Liberator, and also receive future changes to that value |
| discarding | allows you to stop receiving future updates to a previously requested item |
| creating | allows you to create new objects on Caplin Liberator in order to contribute new values to them |
| deleting | allows you to delete objects from Caplin Liberator |
| contributing | allows you to add or modify the data and/or name-value pairs associated with an object on Caplin Liberator |
| throttling | allows you to change the throttle level (frequency of updating) for each object |

# 5 About Caplin Liberator

Caplin Liberator is a complete RTTP publishing and subscription management system which features an advanced push server for streaming real-time data to any RTTP-aware client.

Contributing applications send market data to Caplin Liberator, which then aggregates the data and publishes it over the internet using RTTP, where it can be integrated into web pages or custom applications. Systems can be set up with multiple Liberators for load balancing and fault-tolerance purposes.

The StreamLink library is designed to connect to a single Liberator at a time. It is able to fail over from one server to another, but cannot hold a connection to more than one server. If your application requires connections to more than one server, create a new instance of the connecting class or a subclass for each server using properties that point to Server 1, and another instance using properties pointing to Server 2. If the updates from all Liberators are to be displayed in a single place you will need to write an object to collect the updates from the various different StreamLink instances.

User permissioning and usage monitoring can be carried out in a variety of ways:

- using Caplin Liberator's authentication API to write your own authentication module in C or Java for interfacing to a proprietary database or entitlements system;
- using the simple built-in file-based authentication, the quickest way to get started with your Caplin Liberator;
- using Caplin's XML Auth Module, which uses XML to create permissioning structures and control the entitlement of objects held on Caplin Liberator.

## 5.1 Data sources

Caplin Liberator is capable of retrieving data from any source connected to the DataSource Integration Network, an interface that enables most Caplin and RTTP-related products to communicate with each other. These can be configured to be "active", which means the source will keep track of which objects have been requested and send updates for those objects only. This improves performance by reducing network bandwidth requirements.

DataSource handles data from the following sources:

- Triarch;

- RMDS
- ComStock ;
- TIB/RV;
- Caplin Transformer, which can receive large volumes of raw real-time market data and republish it as value-added data;
- Reuters MarketLink feed;
- Microsoft Excel via Caplin's DataSource for Excel software;
- Caplin's PriceMaster;
- Any client application written in C or Java which uses the DataSource API toolkit.
- Any client application for which an adaptor has been written in C or Java using the DataSource API toolkit.

## 5.2 Permissioning and security features

Caplin Liberator supports a modular system for handling authentication and authorisation. Each "Auth" module allows users to be authenticated, objects to have permissions loaded, a user's read and write permissions for an object to be checked and object name mappings to be performed.

## 5.3 Clustering

Caplin Liberators can be arranged as a cluster and act as one server, in order to monitor licence use and numbers of users logged on. The cluster can be configured to use a global cache, which means on failover each clustered Liberator can provide data from the same cache without having to rerequest it from the data source.

## 5.4 Reconnecting

Caplin Liberator will maintain a session following a disconnection for a configurable period, to enable the user to reconnect and continue to receive updates for objects they have subscribed to without having to request them again. Update messages are stored in an output queue which

can be resent when reconnecting, so that the client is less likely to miss messages while disconnected.

# 6 Runtime configuration

All configuration for StreamLink is done via name-value pairs passed into a constructor or a subclass.

## 6.1 RTTP Client Library properties

These properties configure the RTTP Client Library (the communication layer on top of which StreamLink is built), including:

- How to identify the client application.
- The connection types to attempt (RTTP, HTTP or refresh).
- The host names and ports of the RTTP servers; the protocol to use to contact a proxy and the host and port on which the proxy resides.
- The XML file containing the failover algorithm and servers information.
- The user name and associated password used to login to the server. This will be used for all servers listed in the XML file.

## 6.2 Reconnection properties

These properties configure how StreamLink handles reconnections to Liberators, including:

- The number of times to attempt reconnection after each disconnection before giving up the attempt and how long to wait before retrying to connect after a reconnection attempt has failed.
- The size of the batches that will be used to rerequest data after a disconnection.
- The filtering to occur on data received from the server. If a field is received from the server that was not specified in a request, it will be discarded. This only applies to objects which have been requested with fields.

## 6.3 RTTP Monitoring properties

Monitoring ensures that objects are receiving regular updates. These properties configure what items should be monitored:

- Any RTTP object name, beginning with "/" or similar. Only this property is required to monitor an object.

- Time after the last update for the object before an error is signalled.

# 7 Subscribing to data

## 7.1 Requesting an object

In order for StreamLink to receive data, the client application must subscribe to the objects whose data it is interested in. This is achieved by requesting the objects from the server.

To make a request, you must create a target using a factory class. When the object has been successfully created, the object can be requested from the server by passing the object that has just been created into the request method. This will request the object with the specified object name from the server, and the client will receive all the updates for that object provided it exists on the server and is updating. If there is a problem with the request, an exception will be thrown.

## 7.2 Requesting specific fields

You can request updates for a subset of the record's fields. For example, the client may want to request only the Bid and Ask fields for the record. This can be achieved using a method which includes a list of the fields that should be subscribed to for the specified object.

## 7.3 Requesting Type 2 and Type 3 records

There are also several factory methods which create specialised types of target in order to request more specific object types such as Type 2 and Type 3 record data.

***Note:*** *although the client can request a Type 2 or Type 3 object using the factory methods which create the targets, the Type 2 or Type 3 callback methods will only be called if the object is a record, and has fields that are of Type 2/Type 3. For instance, if the object requested is actually a page, StreamLink will handle updates as pages rather than Type 2 or 3 updates.*

## 7.4 Batched requests

If the client application needs to request many objects from the server, it can do so by using a method that batches requests together. This method has the advantage of using only one message to request several objects instead of many messages requesting individual objects.

## 7.5 Discarding an object

After an object has been requested, the server will continue to send updates for that object to StreamLink until the client application is closed. If the client wants to stop receiving the updates, it can do so by discarding the object.

## 7.6 Discarding specific fields

If an object has been requested with specific fields, you can discard some of these fields and keep receiving updates for others. For example, if an object has been requested with the Bid and Ask fields, the client can discard the Ask field for that object, in which case it would still receive updates for the Bid field, but would stop receiving updates for the Ask field.

## 7.7 Batched discards

As with requests, it is also possible to discard many objects from the server with one message.

# 8 Receiving data

Once an object has been requested from the server, the client receive an initial image for that object and then subsequent updates for it until the object is discarded. The object's image is its current state when the client first requests it, which will come from the server's cache.

When StreamLink receives an update, it determines the object's type (e.g. record, directory or chat) and then calls the associated update method. The client is able to override these methods to process the updates in the way required by the application.

# 9 Publishing data

## 9.1 Contributing to an existing object

A client can contribute to any object it has write access to, regardless of whether it created the object itself, or whether the object was created by another of the server's data sources. If created by another source, the contributed values will be passed on to that data source for processing.

(At present, Liberator only accepts StreamLink contributions to record and chat objects.)

***Note:*** *If the client contributes values for fields that are not configured on the server, the contribution will appear to succeed; however values for any fields that do not exist will be ignored. For example, if the client contributes values for the fields Bid and BidSize to a server that does not have the BidSize field configured, only the Bidvalue will be contributed to the object.*

## 9.2 Creating a new object

To create an object, the client must have write access to the directory in which the object is to be created, and that directory must already exist. For example, the client may have write access to the /DEMO directory, but it can only create the object /DEMO/EQUITY/TEST if the /DEMO/EQUITY folder exists. If the client attempts to create an object in a directory that does not exist, an exception will be thrown.

When an object is created, the client must specify the data type (i.e. whether it is a record, page, news headline etc).

## 9.3 Deleting an object

To delete an object, the client must have write access for that object. If the client attempts to delete an object that does not exist, an exception will be thrown.

# 10 Handling events

RTTP is based on responses and events.

| | |
|---|---|
| Responses | sent to the client by Liberator after a message is sent to the server. |
| Events | received at any time; they may be produced as a side-effect of a previous message sent to the server (such as an update). Most events are generated by the server and delivered to the client but some connection events are generated by the client directly. |

Due to the design of StreamLink the client only has to process the events types that they are interested in. All events that can be received are represented by callback methods. Amongst the events that StreamLink handles are:

- when a response or event containing data has been received. Most clients will want to override more specific methods based on the type of data they expect to receive (i.e. when a response or event containing data relating to a directory object or a Type 2 record object has been received).
- when a subscription to an object either failed or became invalid. If this was a new subscription it may be necessary to inform the user.
- when a query has been made to the source as a result of a subscription request. Further events will be generated when the result of the subscription is known.
- when the server has disconnected this client because another client has logged in with the same username. As there is no automatic reconnection in this circumstance, the login method must be called to reconnect the session.
- when an error has been detected with the connection to the server and the client has been disconnected. If configured, reconnection attempts will be automatic.
- when an attempt is being made to connect to a server.
- when a connection to the server has been established.
- when the status (i.e. validity/freshness) of a subscribed-to object has changed.

- when a source attached to Liberator has gone down or back up. If a source goes down then data for all symbols supplied by that source may be stale. When the source comes back up again, the data may not be stale.

- when the status of a source attached to the server has changed.

## 10.1 Advanced event processing

All messages received by the client (both responses and events) have an RTTP code which indicates how the message should be interpreted. There is a large number of RTTP codes, but StreamLink removes the need for a developer to be familiar with them. However it is sometimes necessary to access these codes when coding advanced functionality.

StreamLink automatically processes responses—failure responses are converted to exceptions and thrown. Failure responses contain a reference to the original response from which the client can access the code. All other responses are silently discarded (with the exception of responses to subscription requests which contain data) as in these cases the data is passed to a callback function; there is no way for the client to access the RTTP code for these responses.

The RTTP codes can be retrieved from a class, but this class is part of the RTTP Client Library and does not have public documentation. The RTTP Client Library also provides methods which allow comparisons to be made with the value returned.

# 11 Requesting different fields for the same symbol

When a subscription is first made for a symbol the server allocates a sub-channel within the RTTP stream and delivers all data associated with the symbol over this channel. If further subscriptions using the same symbol name are made in the future (e.g. if the symbol VOD.L with field "Bid" was initially requested, and subsequently VOD.L with fields "Ask" is requested) then this data will come down the same subchannel which has been allocated for that symbol name. This behaviour is useful when data is being distributed to a number of consumers as it minimises the bandwidth required to deliver the data.

There are situations where this behaviour is not desirable, for instance when a request is made using a filter or when the same symbol is required at two different throttle levels. In these cases, although the data the server sends out may be for the same symbol name and fields, the actual data will differ. A new sub-channel can be allocated by the server for each subscription made, even if other sub-channels currently exist with the same symbol name.

# 12 Re-requesting symbols

Streamlink does not maintain a cache of current values or images. As described above, the server allocates a sub-channel for each distinct symbols request. If a symbol is requested once then subsequent requests will **not** result in an image being retrieved from the server (with the exception of additional fields as previously discussed). Therefore the application using StreamLink must cache existing values if they are required later by a subsequent request. A unique sub-channel can be created using a UniqueRTTPTarget, this will result in a new complete image being retrieved but also a corresponding increase in required bandwidth as each update will be sent down both sub-channels.

# 13 Handling incoming messages using listeners

Listeners provide the client with callbacks which enable you to keep tab on events regarding failed requests, connection status and miscellaneous messages such as source status. All RTTP messages delivered to the client application by StreamLink arrive through callback methods, except for some messages relating to chains.

For non-RTTP events (i.e. events generated on the client side not related to the server connection) StreamLink allows listeners to be added by a client application:

- as part of the shutdown sequence, to allow clients to release any resources they may be holding and terminate internal threads.

- after an object has been passed to the request method, the request has been sent to the server, a positive response received and a "requested" method called. (Currently this is used internally by the chain functionality, but is available for client applications to use if appropriate.)

- when an object has been successfully discarded as a result of being passed to the discard method. Note: if a subscription to an object ends for a reason other than a discard (e.g. loss of connection, delayed failure message from a data source) no call will be made to the discard listener.

# 14 Handling chain data

StreamLink provides support for requesting objects as chains, maintaining chains and accessing the values of elements within a chain.

Chains are used to group together records to form meaningful sequences: chains are often used to represent indices and "most active" lists, for example. They provide a sequential list of record names that may grow, shrink and update in real-time. An application may request these record names to access data within the records that form the chain.

Chains are often represented on Liberator by multiple linked symbols, but StreamLink hides this implementation and provides the entire chain in one list associated with a single symbol name.

## 14.1 Chain listeners

Listeners can be added to a chain object to listen for the following:

- core chain update events such as additions or changes to chain elements;
- any errors that occur either during chain loading or subsequently;
- any change to the status of the chain such as a part becoming stale.

## 14.2 Requesting chains

A chain object, created using the factory class, is requested using the usual StreamLink request mechanism.

## 14.3 Receiving chain updates

Updates to the chain, such as it growing, shrinking or the value of a chain element changing are notified to the client. An event is fired as each chain element is received from the server, which allows any required processing to occur whilst the chain is loading. (This can prove more efficient than waiting for the complete chain to be received.) Once the chain is complete, an event is fired to the registered listener.

After the chain is loaded, it may grow, shrink or an element in the chain may change value—in each case an event may be fired to the registered listener.

Typically an application will use these events to request and discard the records named by the methods which retrieve the update value of a chain element and compare them to the previous value.

## 14.4 Querying a chain

At any time, the chain can be inspected to check its state, namely:

- the current size of the chain.
- whether the chain is complete.
- whether the chain is known to be invalid.

Also a specific element, all elements or an enumeration of all the current elements can be retrieved.

## 14.5 Errors with chains

When an error occurs whilst retrieving a chain an event is fired to the registered listener.

Two types of errors can occur when requesting a chain: when the object is not available as a chain, or the chain is invalid. Both events contain a string description detailing the error.

When one or more parts of the chain are unavailable for some reason, an event is fired. In addition to a string description, an object is available detailing the specifics of the error.

## 14.6 Status Messages

When any part of the chain goes stale or has another status change (such as not being stale anymore as a new update has occurred), an event is fired to the registered listener. The event contains an object that indicates the new state of all or part of the chain.

It is possible for only part of the chain to become invalid in some implementations.

## 14.7 Discarding chain objects

When a chain is no longer required, the standard StreamLink discard mechanism is used to discard the chain.

This causes the underlying implementation to perform the appropriate discard action to the chain, which might involve sending multiple messages to the server.

# 15 Throttling updates—Advanced Flow Control

Throttling (also known as conflation) is the process of capping the rate at which updates are sent to a client. This is done by sending out one update every throttle period at most, an update that contains only the most recent value received from the data source.

"Advanced Flow Control" allows dynamic throttling to be performed where the throttle level for a particular symbol can be changed by the client at any time.

Throttling can be used for a number of reasons:

- to reduce network usage levels, both leaving Caplin Liberator and entering the client;
- to reduce load on the client, for instance when displaying updates on screen at a high rate is overloading the client machine;
- to reduce load on Caplin Liberator, by reducing the number of updates it sends out.

Throttling examples:

1) If a symbol updates twice every second, but has a throttle time of one second, the client will only receive an update once per second. Note that because Caplin Liberator must wait until the end of the throttle period before sending the update(in case further values arrive), throttled values will be sent to the client with a delay of up to one throttle period.

| Seconds | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **Incoming Updates** | **A** | | | **B** | | | **C** | |
| **Throttled Updates** | **A** | | | **B** | | | **C** | |

2) If a symbol receives an update once every three seconds and the throttle time is one second, the client will receive one update every three seconds. When Caplin Liberator identifies that a symbol is updating less frequently than the throttle time it does not activate throttling and sends

values out immediately. Updates would be sent out as soon as they were received and not at the end of the throttle period.

| Seconds | 0 | ➢ | 1 | ➢ | 2 | ➢ | 3 | |
|---|---|---|---|---|---|---|---|---|
| **Incoming Updates** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | |
| **Throttled Updates** | | **B** | | **D** | | **F** | | |

## 15.1 Throttling configuration

Throttling is set up by specifying a number of global and per-symbol throttle times in Caplin Liberator's configuration.

When a client logs in and requests symbols they will be throttled according to these configuration settings. A client may then alter the throttle levels for any symbols it is subscribed to or subscribes to in the future, within the throttle levels specified in the Liberator configuration.

These throttle levels are controlled using the commands shown below. The client is not aware of the actual throttle times as these are configured in Liberator, but is able to switch from one throttle level to another. Within StreamLink a class provides access to the valid throttle levels.

The methods on the throttling interface send commands to Liberator to perform the following:

- adjust the throttle level on all existing subscriptions currently at the 'default' level and all future subscriptions.
- adjust the throttle level on a specific object.
- adjust the throttle level for one or more objects identified by the targets array. All objects will be set to the same throttle level.

# 16 Filtering requests and updates

## 16.1 Client-side filtering

In certain situations Caplin Liberator sends unrequested fields to reduce server-side processing.

Because StreamLink maintains a list of requested symbols and fields for reconnection purposes, it is possible to filter out any fields a Caplin Liberator sends which the client has not requested. For example, if the client requests LO/VOD Bid Caplin Liberator may send an update containing LO/VOD Bid, Ask and Time. If client-side filtering is enabled, the additional fields will be removed from the update before it reaches the client application.

## 16.2 Server-side filtering

Liberator can accept requests for objects with a user-defined filter—only updates matching the expression given by the user will be sent to that user. The expression is based around the values of the fields in the update and can contain most standard logical operators.

For example:

- a user might only request news headlines which are sport headlines or contain the word "golf", or updates to a stock where the value of the Volume is greater than 10000.

- two requests could be made, both for /VOD.L;Bid. The first has a filter "Bid>100" and the second "Bid>200". Liberator will see these as entirely separate requests and send separate sets of updates for each. If an update of Bid = 250 occurs, two updates will be sent to the client. Because both subscriptions have the same object and field names it is necessary to add a differentiator to identify which update is for which subscription.

Every time a subscription is made a new object number is allocated on Liberator. This means that a client can receive a field value several times when multiple subscriptions were made with the same symbol name.

Currently there are two types of filtered subscription supported: record filtering and news filtering.

## 16.3 Record filtering

Records are filtered by specifying a filter when creating an object. The filter string consists of checks for the presence of fields and/or their values. Record filters are specified using a filter string with the operators shown below.

| Character | Meaning |
|---|---|
| ¦ | or |
| & | and |
| = | equals |
| ! | not |
| < | less than |
| > | greater than |
| ~ | contains |
| ( ) | perform these filters first |

It is not possible to perform queries where these characters form part of the field name or the value to be compared.

## 16.4 News filtering

News is filtered by specifying a news filter when creating an object. The filter string consists of news codes and whole words to search for. To create a news filter specify a filter string using the operators shown below.

| Character | Meaning |
|---|---|
| [space] \| | or |
| + & - | and |
| = | equals |

| Character | Meaning |
|---|---|
| ! ~ | not |
| ' | start or end of free text search string |
| " | start or end of free text search string |
| ( ) | perform these filters first |

It is not possible to perform queries where these characters form part of the value to be compared.

As well as serving up cached headlines previously broadcast to Liberator, Liberator can actively collect historic news using a suitably-configured DataSource such as DataSource for HNAS. This enables clients to request news from a certain date without being limited by Liberator's cache size.

## 16.5 Receiving filtered updates

The objects used in a request are passed back into overloaded callback methods when updates are received. These methods include the object used in the original request to allow the application to identify which requested object the update belongs to.

# 17 Decoding binary data

DataSource provides the ability to pass binary data into Caplin Liberators, encoded and stored as a string. Various types of binary data are predefined in DataSource— please refer to the DataSource documentation for more information.

Binary data is received by the client as a normal value in a name-value pair. StreamLink provides a helper class which decodes these binary types. When expecting binary-encoded values in a particular field you can simply call a class and an object representing the type will be returned. You can override the method to perform your own processing of the data.

# 18 Server failover

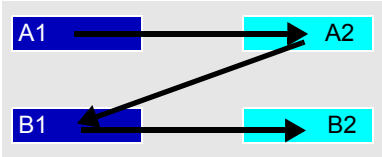
Caplin has designed a number of algorithms to select a backup server from a list when a connection to a server fails. These algorithms balance the load across the servers and minimise use of the backups. There are three algorithms, named after the pattern of selection they produce.

The examples below use two primary servers with two backup servers. A1 is paired with backup server B1 and A2 is paired with B2. The client is assumed to attempt to A1 first.

## 18.1 Z failover algorithm

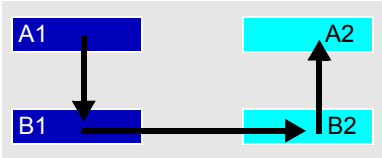
Always tries to failover to a primary server. Backup servers will only be attempted if StreamLink has failed to connect to all of the primary servers.

Example sequence: A1 > A2 > B1 > B2




## 18.2 U failover algorithm

If the client cannot connect to a primary server, it attempts to connect to the backup server. If there is no backup server for the primary server, then a server is selected from the list of other backup servers.

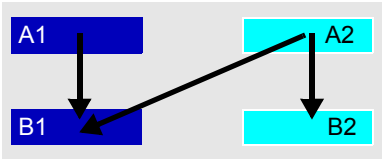



If the client cannot connect to the backup server, then it will attempt to connect to another backup server. It should only attempt to connect back to a primary server once the list of backup servers has been exhausted.

Example sequence: A1 > B1 > B2 > A2

## 18.3 N failover algorithm

If the client cannot connect to a primary server, it attempts to connect to the backup server. If there is no backup server for the primary server the client is connected to, then a server is selected from the list of primary servers.




If the client cannot connect to the backup server, then it should attempt to connect to a primary server.

Example sequence: A1 > B1 > A2 > B2

User-defined algorithms A user-defined algorithm can decide whether StreamLink should connect to a paired backup server (if there is one) or to a random primary or backup server.

## 18.4 Choosing a failover type

Client applications must download an XML file (rttp.service.url) and associated DTD file in order to use algorithm-based failover.

The XML file must contain a list of servers available for the RTTP Service (a set of Caplin Liberators which all accept the same clients and provide the same set of data), and the preferred failover algorithm. If no XML file is specified, the application will repeatedly try to connect to the servers identified by its configuration properties.

If the XML file URL is invalid, the XML does not conform to the DTD or a value within the XML is invalid, an exception will be thrown.

The required failover type can be selected by using a property which identifies the location of the XML file containing the list of servers. If an algorithm is included in the XML file it cannot be overridden.

If no algorithm is set in the XML file, it can be set in the client application using a different property, which must contain the fully qualified class name of the algorithm.

If no algorithm is set by these means, the application will use the Z method of failover by default.

# 19 Connection interfaces

## 19.1 Connection options

When an initial connection to the server is created, it can continue to exist until the application is shutdown, the server is shutdown, or network problems cause a break in the connection. Alternatively this connection will continue to be maintained, even if the server is restarted or network problems cause a break in the connection.

It is important that the functions to perform this should only be called at the appropriate time, for instance if a message has been received indicating the connection has been lost. In addition if using the reconnecting class neither a connect or login method should be called if a disconnect or logout method have not been called previously.

An interface is provided to notify a client when the StreamLink library shuts down. A notification is made after the logout message has been sent to the server.

## 19.2 Noop messages

The noop interface has a method which sends a message to the server to ensure that the connection is functioning correctly. This method can also be used to give an indication of the round trip time of sending a message to the server and receiving the response. The client can call this method on a regular basis or at any time while connected in order to test the connection.

## 19.3 Statistics

The statistics interface provides various methods which return given statistics about the connection to the server, such as bytes sent and received, connection type and latency measured as half their round-trip no-op time.

One method returns the size of the update queue, i.e. the number of events that have been received from the server but have yet to be passed on to the client application. Normally this queue will be very small or empty; if this is not the case it indicates that the client application is not processing updates quickly enough and the data being processed by the client is not up-to-date. This situation can be improved by adjusting the throttling to the data or by reducing the amount of processing that is done per update.

# 20 Service up and down notifications

## 20.1 Data services

In version 4.0 of the Caplin Liberator, the source mapping functionality was enhanced to explicitly define which data sources a particular object will be requested from. Such a collection of data sources is called a data service. Each source that makes up the service must either be defined as being required or non-required. A required source is one that is providing critical data and must be up for the service to be considered being OK (e.g. a source providing the current market prices for an object). A non-required source is one that is providing non-critical data (e.g. a source providing fundamental data for an object).

## 20.2 Service status

The state of the service is dependent on the states of its constituent sources. The possible states are:

- OK - All of the data sources for the service are up.
- DOWN - One or more of the required data sources are down.
- LIMITED - One or more of the non-required data sources are down, but all of the required data sources are up.

StreamLink passes on notifications about the change of a data service's state using the **serviceStatusUpdated** method.

## 20.3 Object status updates

The state of an object is dependent on the state of the data service that provides it. It is also possible for a data source to send out individual object status messages. The possible states are:

- OK - All of the data sources for the service that is providing the object are up.
- STALE - One or more of the required data sources providing the object are down, or have sent a specific stale message for that object.
- LIMITED - One or more of the non-required data sources providing the object are down, or have sent a specific stale message for that object.

- INFO - A informational message about the status of the object.
- REMOVED - The client's subscription for the object had been cancelled on the Liberator. The client will not receive any more updates for that object.

These object states replace the states that were used prior to 4.0. The REMOVED state replaces the old BAD state, whilst OK replaces NOTSTALE. The only state that doesn't have a corresponding value in 4.0 is NOTSTALEWAIT. This state no longer exists - when the OK state is received the Liberator guarantees that the data the client currently has for the object is up-to-date.

Object status updates are passed on using the **objectStatusUpdated** method.

## 20.4 Example

The following diagram demonstrates how two data services might be set up. The first, Service A, is used for all objects that match the pattern /LO/*. This service is dependent on one source, Source 1, which has been defined as being required. The second service, Service B, is used for objects that match the pattern /NA/*. It uses two sources, Source 1 which is required, and Source 2 which is non-required.

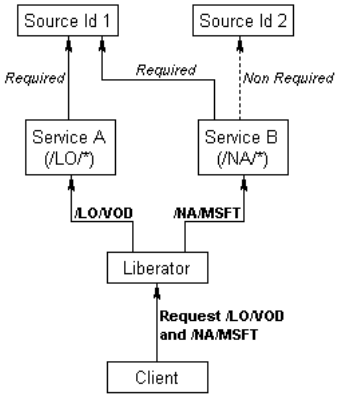



***Figure 20-1: Data services example***

In this example, the client requests two objects: /LO/VOD and /NA/MSFT. /LO/VOD matches the pattern for Service A, and is requested from Source 1. /NA/MSFT matches the pattern for Service B, and is requested from both Source 1 and Source 2.

If Source 2 goes down, a LIMITED status message will be sent for /NA/MSFT (and for Service B). /LO/VOD and Service A are not affected.

If Source 1 then goes down, a STALE status message will be sent for /NA/MSFT and /LO/VOD, and a DOWN status message will be sent for both Service A and Service B.

If Source 1 now comes back up, an OK status message will be sent for /LO/VOD and Service A, whilst a LIMITED status message will be sent for both /NA/MSFT and Service B (since the non-required source Source 2 is still down).

When Source 2 comes up, an OK status message will be sent for /NA/MSFT and Service B. /LO/VOD and Service A are not affected.

# 21 Using StreamLink for Java in applets

## 21.1 Writing an applet

StreamLink for Java can be used within any applet running in a browser which supports JDK 1.1 or above.

It must also abide by the standard applet security rules, such as connecting to the server from which it was downloaded, and not attempting to connect to multiple servers. It is possible to work around these rules if the applet is signed.

## 21.2 Obfuscation

Obfuscation has two advantages—it prevents end-users from being able to decompile and reuse any parts of an application easily, and reduces the size of the code that has to be downloaded to the end user.

Caplin Systems Ltd suggests the use of yguard (http://www.yworks.com/en/products_yguard_about.htm), a reliable bytecode obfuscator which Caplin uses to obfuscate other Java products. It is open source, free for commercial or non-commercial use and written in Java.

***Note:*** *When obfuscating, avoid the use of string-based reflection within the code, as this will cause a runtime failure unless classes referenced in this way are excluded from obfuscation.*

In StreamLink for Java the following classes are loaded by reflection and so should not be obfuscated:

## 21.3 Compression

The size of download for Java code can be significantly reduced by compressing code. In general files are compressed by adding them to a single archive file and compressing them in the process. The Microsoft version of Java supports archives in the JAR (Java Archive) format as specified by Sun, which is the same format as a .zip file and Microsoft's proprietary CAB format.

Caplin has found that the compression used in CAB files produced by Microsoft's Cabarc tool (available in the MS SDK for Java) is significantly more efficient than that used in JAR/zip files;

however non-Microsoft JVMs — including Sun's Java plugin — only support JAR format archives. Therefore Caplin recommends that if both Microsoft and non-Microsoft JVMs are to be used, both JAR and CAB files should be made available to the applet. CABs are specified in the Microsoft specific *cabbase* parameter of an <APPLET> tag.

# 22 Debugging

## 22.1 Setting the debug level

StreamLink initialises the DebugLevel it uses to the value of the *streamlink.debug.level* property. This determines which debug messages are output by the system and which are ignored. If this property is not set, or has an invalid value, StreamLink will default to using *DebugLevel.DEBUG_LEVEL*.

After StreamLink has been initialised the debug level can be modified using the **setCurrentDebugLevel(DebugLevel)** method.

## 22.2 Redirecting debug output

The location to which debug messages are output is determined by the **DebugMessageListener** that has been registered with StreamLink. All debug messages generated by StreamLink and the RTTP client library will be passed onto this listener. If a listener is not explicitly set, a default one (**StandardDebugMessageListener**) will be used that outputs the messages to standard out.

The DebugMessageListener used by StreamLink for Java can be configured using the *streamlink.debug.message.listener* property. This must specify the fully qualified name of a class that implements the **DebugMessageListener** interface.

***Note:*** *This functionality differs significantly from versions prior to 4.0. In these versions the **outputMessage(String)** and **outputMessage(String,Object[])** methods were meant to be overridden to redirect debug messages. Unfortunately this was incompatible with changes in 4.0 where the RTTP client library and StreamLink debug message outputs were unified. Any application that overrides either of these methods should be modified to use a **DebugMessageListener** instead. The **outputMessage(String,Object[])** and **outputMessage(DebugLevel,String,Object[])** methods should still be used to log debug messages however.*

# 23 Troubleshooting

If your program does not work as expected it is often useful to use the debug version of StreamLink. This can output various types of debugging information, including:

- size of response and update queues:
  [DebugLevel.RTTP_FINE_LEVEL];
- RTTP messages sent by client and received from Caplin Liberator:
  [DebugLevel.RTTP_FINER_LEVEL];
- HTTP headers on HTTP communication with Caplin Liberator:
  [DebugLevel.RTTP_FINEST_LEVEL].



## Contact Us

Triton Court
14 Finsbury Square
London EC2A 1BR
UK
*Telephone: +44 20 7826 9600*
*Fax: +44 20 7826 9610*

**www.caplin.com**

**info@caplin.com**