weswit
Software Made in Italy

Network Protocol Tutorial

Last updated: 13/02/2013

Table of contents

1 Introduction

If you use a Client Library (e.g. SDK for JavaScript Clients, SDK for .NET Clients, etc.) to develop a Lightstreamer Client, then you don't need to read this document.

This document specifies the Lightstreamer network protocol that can be used to implement Lightstreamer Client at socket level.

The described protocol is supported by the current version of Lightstreamer Server; see the SDK compatibility notes.
Any older versions of the protocol are still supported; they should be considered as deprecated and are not included here.

In order to receive realtime updates from **Lightstreamer Server**, a **Lightstreamer Client** should open an **HTTP**/**HTTPS** connection (called **stream connection**), through which it authenticates and waits for incoming realtime updates. The Lightstreamer Client could eventually open other **HTTP/HTTPS** connections (called **control connections**) through which it sends commands to the server in order to manage the contents of the stream connection (see section 2.3, Stream Connection, Control Connections).
Basically, there are two different communication protocols that a client can use in order to communicate with **Lightstreamer Server**:

- The ***JavaScript mode***, where *realtime updates* are packed into **JavaScript** commands (used by **HTML Web clients** and, in general, clients based on the **JavaScript Client Library**).
- The ***Text mode***, where *realtime updates* are packed in a simple pipe-separated text protocol (usually used by application clients).

The topic of this document is the **Lightstreamer Server *Text mode*** protocol. It will also be explained the general data subscription/unsubscription/reception workflow.

2 General workflow

In this chapter we will provide a quick overview of the general **Lightstreamer Server** behaviour, in order to give the reader the basic understanding of how the server works.

2.1 HTTP Requests

All the requests that a client sends to **Lightstreamer Server** use the **HTTP** protocol over **TCP**.
Lightstreamer Server also supports **HTTP** over **SSL** connections (**HTTPS** connections).
There are two different **HTTP** methods that can be used for the requests to the server: the **GET** method and the **POST** method.
In a production environment, a cluster of **Lightstreamer Server** can be placed behind a **Load Balancer**.
All these cases will be discussed in this section.

2.1.1 HTTP vs. HTTPS

The client developer has to choose between when normal **HTTP** connections should be used and when to use **secure HTTP** (**HTTPS**) connections; it is recommended to use the following policy:

- **HTTPS** should be used for **stream connections** when realtime data should be encrypted or the user authentication has critical security issues, because on these kinds of connections all user identification data (from the client to the server) and market data (from the server to the client) are sent. Otherwise **HTTP** is suitable for stream connections.
- **HTTP** should be used for **control connections**, because with these connections no sensitive data are sent. In this case it is possible to avoid the SSL server-side and client-side encryption and consequently optimize system performance.

2.1.2 GET method vs. POST method

There are two different **HTTP** methods that can be used for requests: the **GET** method and the **POST** method.
As a general rule, requests to **Lightstreamer Server** cause side effects on the Server behaviour, thus, according to HTTP specifications, they should always be issued through the **POST** method.
This also avoids any issue with querystring length limits and is compliant with the *Web Services standards*.
For testing purposes, however, it may be far easier to use the **GET** method.
For instance, this would allow you to issue manual requests through the “telnet” utility without being annoyed by the content-length requirements.
This would also allow you to issue manual requests through a browser address bar.
Using the **GET** method, though not strictly compliant with **HTTP** specifications, will, in general, work as well. Just ensure that the user agent and any intermediate node don't try to leverage the properties of the **HTTP GET** method to issue duplicated requests, for any reason.

2.1.3 Lightstreamer Server behind a Load Balancer

Every **Lightstreamer Server** maintains a list of active sessions. In the case of a cluster of **Lightstreamer Servers** behind a **Load Balancer**, all the **Control connections** coming from a specific client should be directed to the instance of **Lightstreamer Server** that is serving the **Stream Connection** of that client. In order to avoid configuration problems that arise when trying to *stick* a session on a Load Balancing appliance, Lightstreamer Server provides the following strategy:

Each running instance of **Lightstreamer Server** reads its own public address (either the hostname – for example ***ls1.lightstreamer.com*** - or the ***IP*** address of the computer on which it is running) from its configuration file (by the configuration parameter "***CONTROL_LINK_ADDRESS***") and sends it to every client that opens a valid **Stream Connection** with it. Then the client will open the **Control Connections** directly toward the hostname or ***IP*** that was specified by the server.
In order to make this method work correctly, the string written on the configuration file should correspond to a public address.

2.2 Subscriptions and unsubscriptions

In order to receive *realtime updates* from **Lightstreamer Server**, a client should tell the server:

- which **items** it is interested in.
- which collection of **fields** (a **Field Schema** or **Field List**, as explained in the **Lightstreamer Glossary**) it is interested in (for every **item** it is going to **subscribe** to).
- if needed, which of the available **Data Adapters** is responsible for supplying the **item** data.

The action of sending this information to the server is called ***subscription***.
When a client is not interested in an **item** anymore, it should make the reverse operation: telling the server to stop sending *realtime updates* on a certain **item.** This action is called ***unsubscription***.

2.3 Stream Connection, Control Connections and Polling connections

As explained in section 1, Introduction, and in the **Lightstreamer Glossary**, there are two main types of **HTTP**/**HTTPS** connections that a client can open with **Lightstreamer Server**:

- The **Stream Connection**, that is the permanent **HTTP** or **HTTPS** connection used to receive a flow of **updateEvents**. When opening this connection the client should also identify itself to the server (during this operation, the **Metadata Adapter** could *refuse* the connection). A stream connection is bound to a specific **Adapter Set**, composed of a **Metadata Adapter** and one or more **Data Adapters**.
- The **Control Connections** that are temporary **HTTP** or **HTTPS** connections used to send control commands that manage the contents of the **Stream Connection**. These connections are used to make **subscriptions** and **unsubscriptions**.

As explained in the **Lightstreamer Glossary**, **Lightstreamer Server** associates client **Stream Connections** to internal **sessions**. The **Control Connections** act on a **Stream Connection** contents by directly referencing the underlying **session**. Several **Stream Connections** can even follow each other on the same underlying **session**, by binding to the **session**; in this way, constraints on the connection total length can be overcome (see also below, 2.5, Content-Length management).

- Moreover, there is a particular kind of **Stream Connection**, that is the **Polling Connection**. A **polling connection** is a temporary connection that is meant to be part of a sequence of connections on the same **session**, controlled by the client. At the reception of a **polling connection**, the Server sends all the **updateEvents** collected since the closure of the previous **polling connection** of the sequence, then it closes the connection.

A **session** is closed by the client, by truncating the current **connection** or by interrupting the current **connection** sequence. A **session** is also closed upon an unexpected error.
A special **control connection** is also provided to asynchronously force **session** closing. It can be used on the server side, by Metadata Adapters or other back-end processes, to force closure of active sessions. It is available to the clients as well, if needed to close the current **session** or a previous one.

In particular environments, the contents of a **stream connection** may get buffered in some intermediate stage and may not reach the client in real time. To better cope with these cases, a special **control connection** is provided, to asynchronously force the Server to close the **stream connection**, so that the client can rebind to the **session** through **polling connections**.

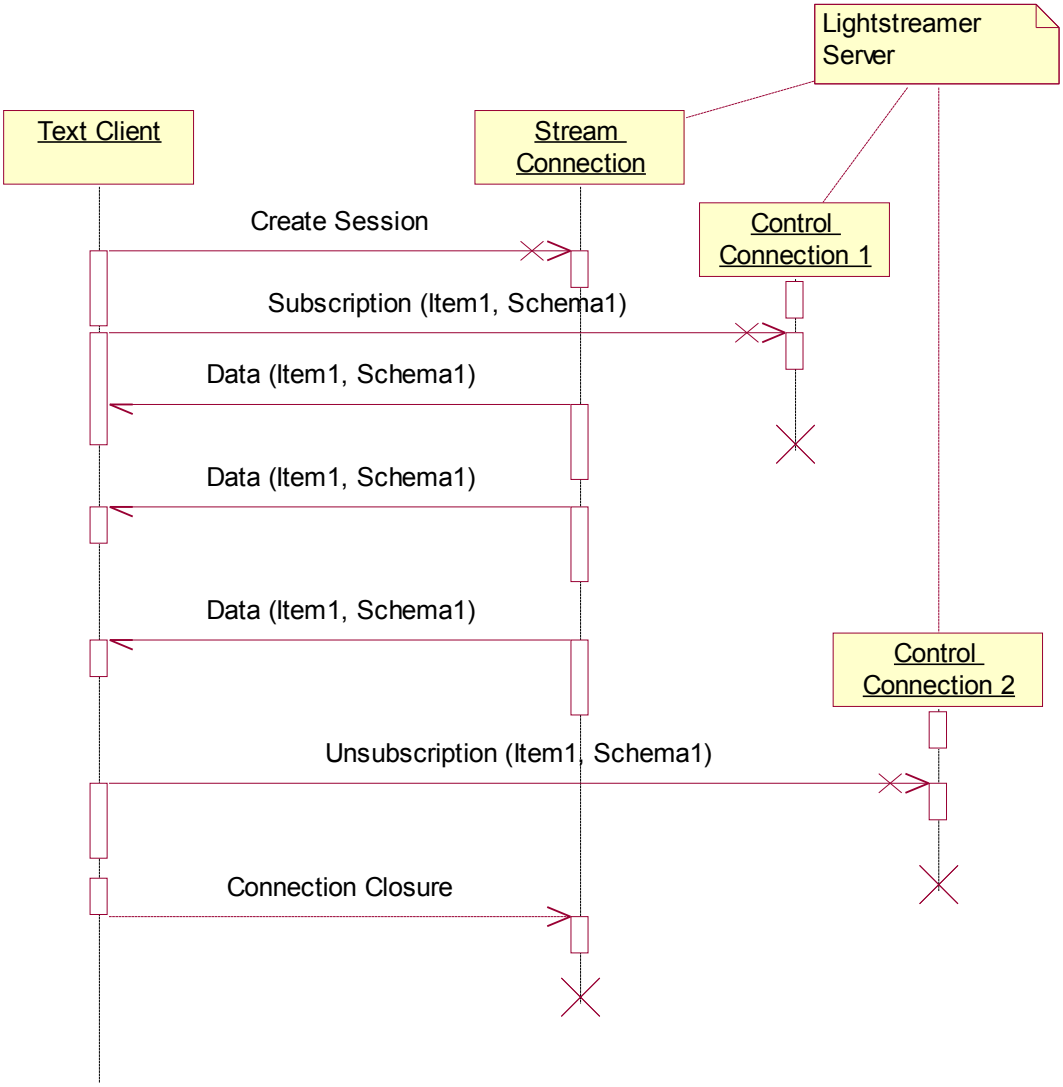
2.4 Data reception workflows

- **NOTE:** For the sake of clarity, in the following sequence diagrams the *connections* (stream and control) are represented as *objects*. The reader should understand that a formal UML representation would need a "*Lightstreamer Server*" object instead.

2.4.1 Basic workflow

In the sequence diagram below, the basic workflow of a simple ***client-server interaction*** is shown:

- The client opens the **Stream Connection** with a **Lightstreamer Server** that accepts the connection.
- Then, the client **subscribes** to the **item** *item1* (with the related **field schema** *schema1*) and **Lightstreamer Server** starts sending *realtime updates* to the client.
- The client **unsubscribes** from the **item** *item1*, **Lightstreamer Server** stops sending *realtime updates*.
- Finally the client decides to close the **Stream Connection** (just by closing the **TCP connection**).



Note:

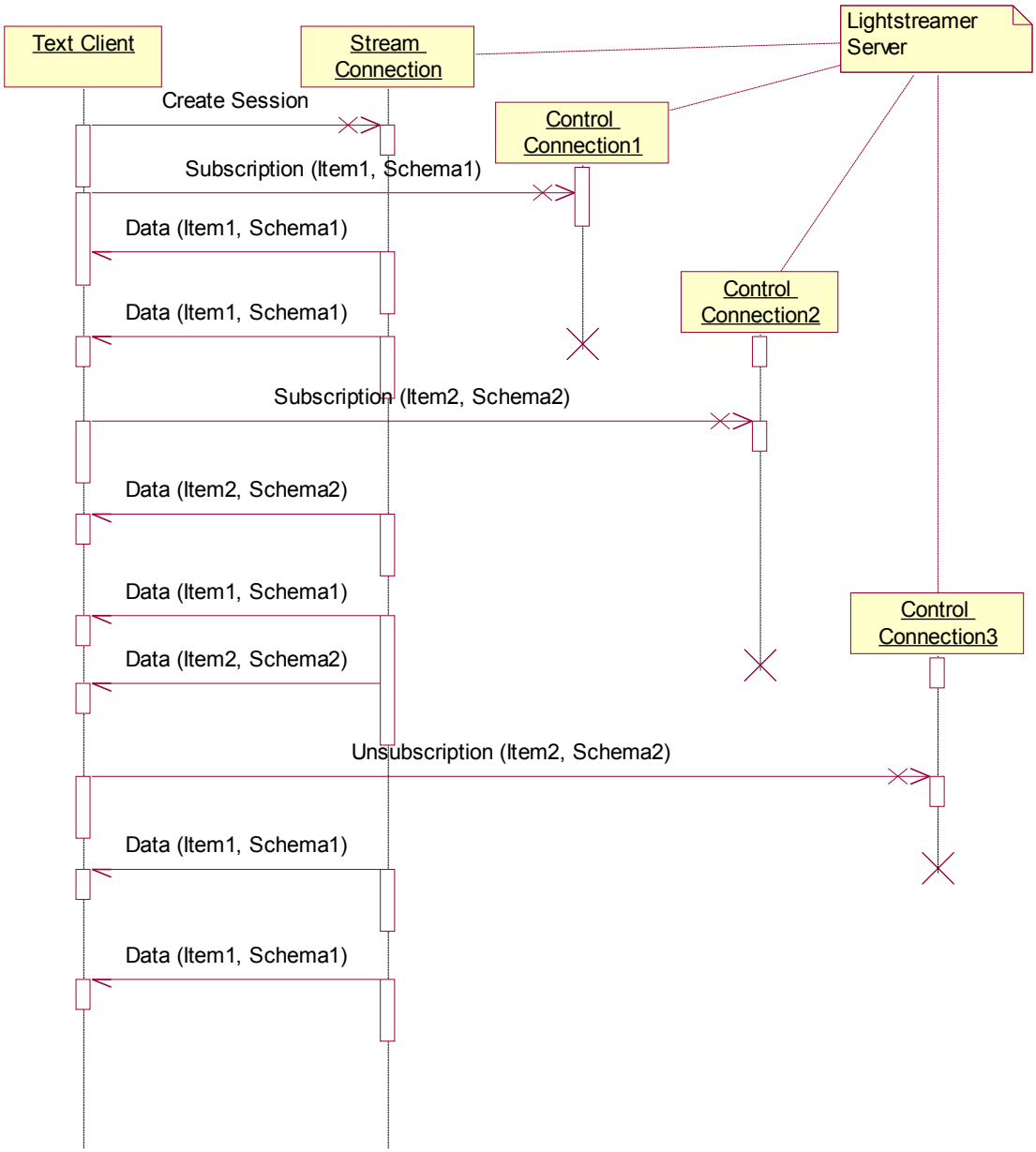
- **Control Connections** are always ***synchronous***.
- Data are delivered to the client ***asynchronously***.

2.4.2 Double subscription

Now we will discuss a more complex example, that is:

- The client opens the **Stream Connection** with a **Lightstreamer Server**.
- The client **subscribes** to the **item** *item1* (with the related **field schema** *schema1*) and **Lightstreamer Server** starts sending *realtime updates* for *item1*.
- The client **subscribes** to the **item** *item2* (with the related **field schema** *schema2*) and **Lightstreamer Server** starts sending *realtime updates* for *item2*.

- The client **unsubscribes** from the **item** *item2*, **Lightstreamer Server** stops sending *realtime updates* for *item2* but continues sending data for *item1*.

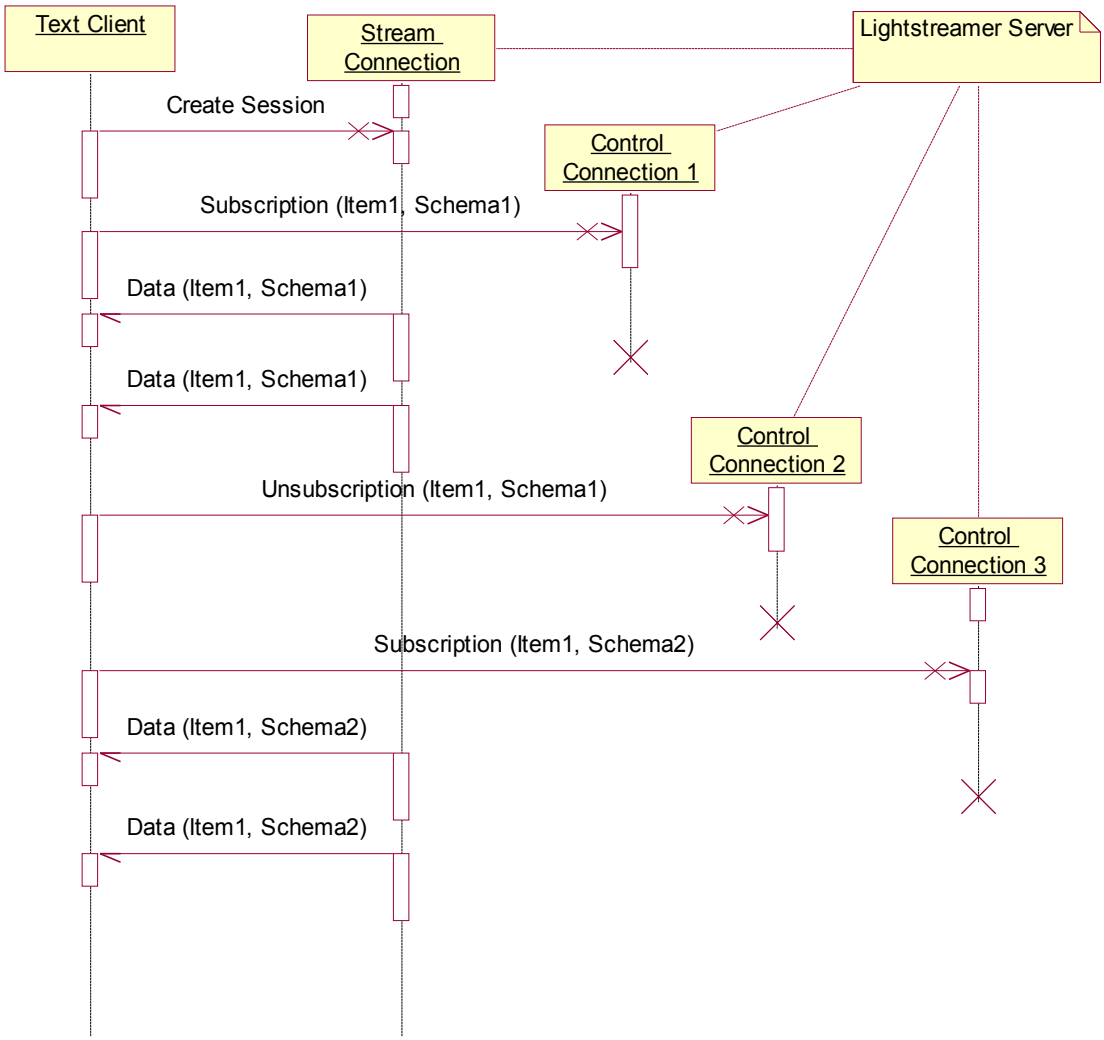


2.4.3 Field schema change

In this case, the client decides to change the collection of **fields** (**field schema**) for which it is receiving *realtime updates*:

- The client opens the **Stream Connection** with a **Lightstreamer Server**.
- Then, the client **subscribes** to the **item** *item1* (with the related **field schema** *schema1*) and **Lightstreamer Server** starts sending *realtime updates* for *item1*.

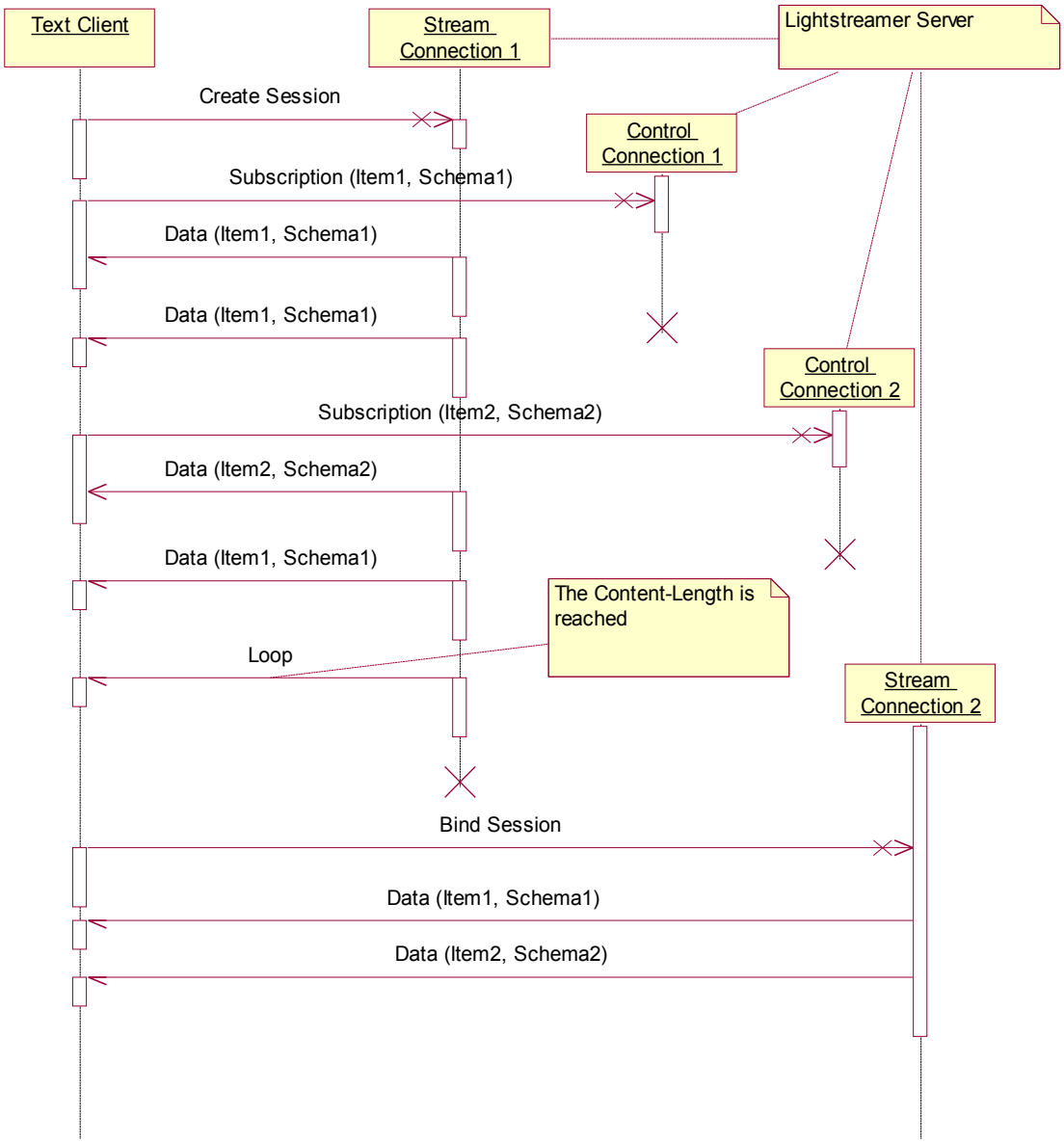
- The client decides to change the subscription **field schema**, so it **unsubscribes** from the **item** *item1* (and **Lightstreamer Server** stops sending *realtime updates* for *item1*) and then re-**subscribes** to *item1* with the new **field schema** *schema2* (and **Lightstreamer Server** re-starts sending *realtime updates* for *item1*, but with different **fields**).



2.5 Content-Length management

Like all **HTTP**/**HTTPS** connections, **Stream Connections** have a **Content-Length** that is the total number of bytes that the server will send to the client while the connection is maintained. When the **Stream Connection** reaches its **Content-Length**, the **HTTP**/**HTTPS** connection is closed and the client should be able to manage the situation.
The **Content-Length** is configured in **Lightstreamer Server** configuration; the client can also request a particular **Content-Length** at connection time, as will be shown below.
When a **Stream Connection** is open, **Lightstreamer Server** sends to the client a **SessionId** (a string uniquely identifying a **Session**). When the **Stream Connection** reaches its **Content-Length** and it is consequently closed, the client should open a new **Stream Connection** ***binding*** this new connection to the old **Session** by specifying the **SessionId** received at the beginning of the previous **Stream Connection**.

The client is advised by the server of the fact that the **Stream Connection** is reaching its *Content-Length* with a **loop** command (see section 4.5.7, Loop messages). See the sequence diagram below:

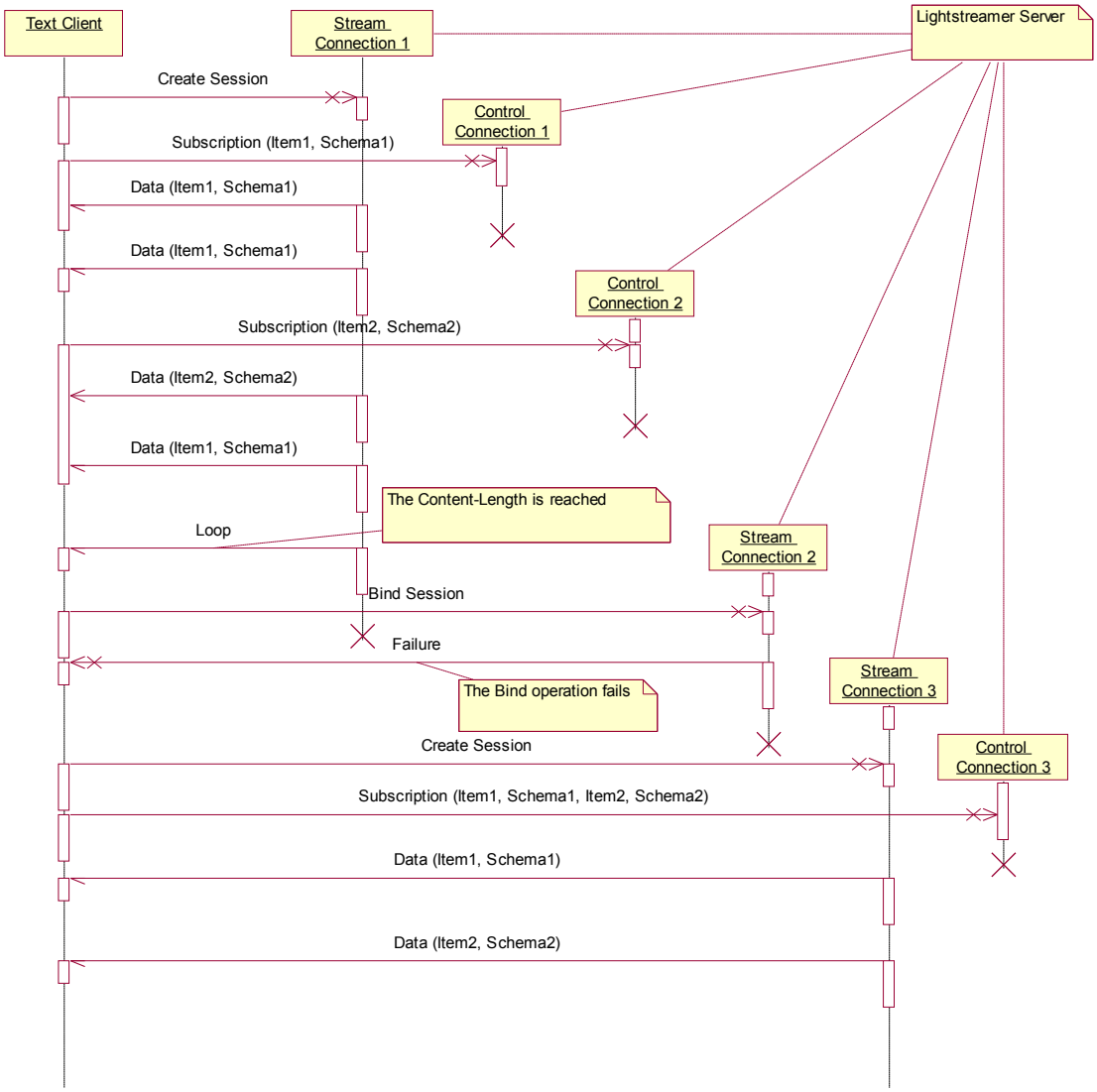


By ***binding*** the old session, the client is sure that the server *remembers* all its **subscribed items** (with the related **field schemas**), and *buffers* the updates that it receives while the first connection has been closed and the second hasn't been established yet.
The ***binding*** operation can fail, for example:

- If too long time passes since the end of the previous **Stream Connection**, **Lightstreamer Server *deletes*** the old **Session**.
- If there is a cluster of **Lightstreamer Server** behind a load balancer, it is possible that for the new **Stream Connection** the client is connected to a different **Lightstreamer Server** that will not recognize the old **Session** (but see section 2.1.3, Lightstreamer Server behind a Load

Balancer, for an explanation of how to avoid this; what was stated there about Control Connections also holds for binding Stream Connections).

In this case, the client should create a new **Stream Connection** as if it was the first time it connects to **Lightstreamer Server** and re-execute all the **subscriptions** that were active at the moment the previous **Stream Connection** reached its **Content-Length**. See the sequence diagram below:



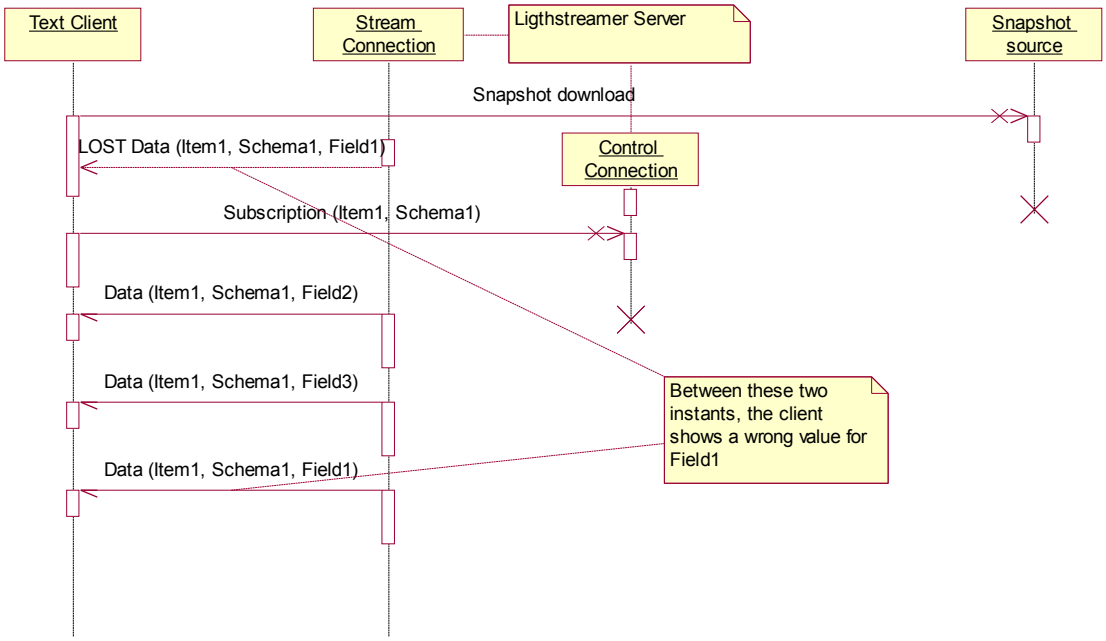
2.6 External Snapshots

In all the example cases analysed until now, it has been supposed that just after the **subscription** of an **item**, **Lightstreamer Server** was able to give the initial **Snapshot** for it.
The ability of **Lightstreamer Server** to supply initial **Snapshots** to the client strictly depends on the ability of the involved **Data Adapter** to supply initial **Snapshots** to **Lightstreamer Server**.

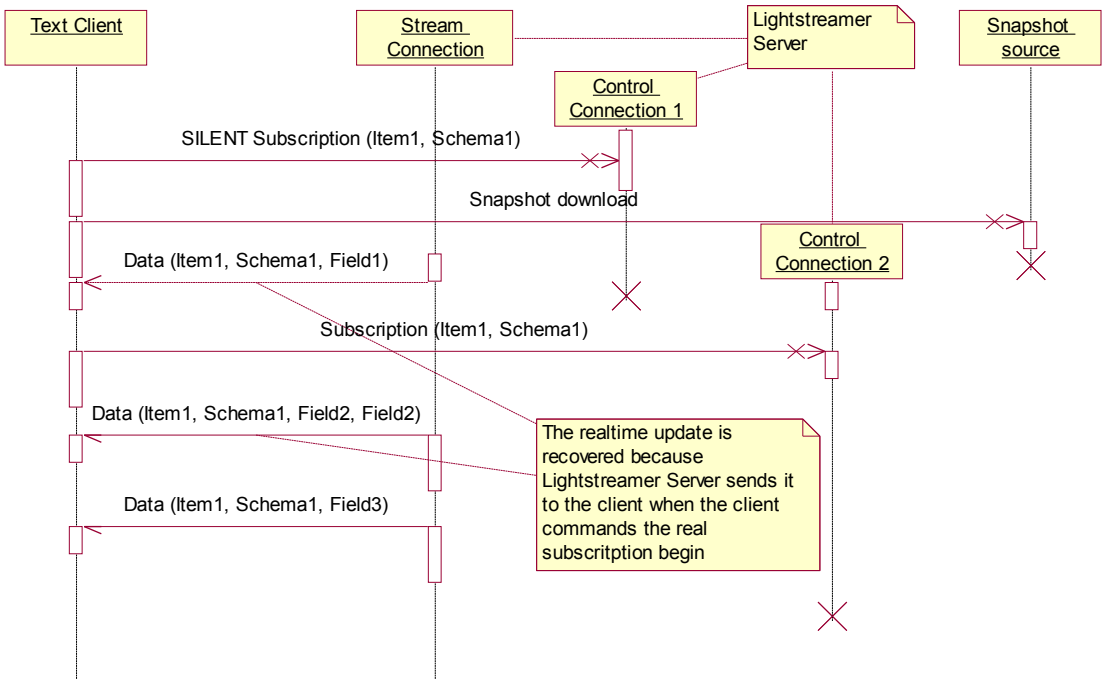
If the **Data Adapter** is not able to supply initial **Snapshots**, Lightstreamer Server tries to recreate snapshot information from the updates flow, but this may result in sending incomplete snapshot to the clients for long time. If this is the case, the client may choose to receive the initial **Snapshot** from another source (for instance, from a proprietary *Web Service*). In these cases, the client should be careful in the **subscription** operation, because the following situation could occur:

- The client requests the **Snapshot** of a certain **item** to the **Snapshot source**.
- Once received the **Snapshot**, the client opens the **Control Connection** to **Lightstreamer Server** in order to **subscribe** to the **item** (without asking for snapshot information).
- While the **HTTP** request *is arriving* to the server, an **item field** changes its **value**; this change is not received by **Lightstreamer Server**, because the request has not been elaborated by the server yet.
- When **Lightstreamer Server** receives the **Control Connection**, it starts sending *realtime updates*, but the **field value** change has been definitively lost and the **field** will be shown on the client with a wrong **value** until the next *update* on it (that will be received by **Lightstreamer Server** and sent to the client).

The sequence diagram below explains the problem:



How to solve this problem? Lightstreamer Server has a proper feature for **Snapshot**-*realtime updates* ***synchronization***. It consists of the possibility to **subscribe** to an **item** in ***silent mode*** (that is, gathering *realtime updates* on **Lightstreamer Server** without sending them to the client).
If the client tells **Lightstreamer Server** to gather data concerning a certain **item** and then requests the snapshot to the **Snapshot source**, once received the **Snapshot** it is sufficient that the client asks **Lightstreamer Server** to send all the actually gathered data in order to avoid the loss of information (in a worst case, there will be duplicated *realtime updates*, but these cases are easily recoverable at the client).
See the sequence diagram below:



As snapshot management has been improved on the Server side, since Lightstreamer Server 3.0 release update synchronization is no longer supported by Lightstreamer client libraries.

3 Subscription Management

In this chapter, we will see the description of the **Lightstreamer Server** logical ***table-group-item*** division.

3.1 Basic concepts

For the complete definitions of the terms **Item**,**Table/Subscription** and **Item Group**, refer to the **Lightstreamer Glossary**. For now, it is important to understand the following concepts:

- A client **subscribes** to the **items** by grouping them into **Tables**, a.k.a. **Subscriptions**. A client can **subscribe** to several **Tables** and the same **item** could be **subscribed** to within several **Tables**.
- When **subscribing** to a **Table** (i.e. **Subscription**) the client specifies an ***id***, that is, the name of an **Item Group**, a string that the **Metadata Adapter** is able to transform into a set of **items**. The client also specifies a **field schema**, that is the same for all the **items** contained in the **Item Group**, and a **mode** (see Lightstreamer Glossary).
- **Field schemas** and **modes** could be different for each **Table** (i.e. **Subscription**). So, if the same **item** is contained in two different **Tables**, it can be **subscribed** to with two different **field schemas** and **modes**.
- When **subscribing** to a **Table** (i.e. **Subscription**), the client must assign it a progressive integer (1, 2, 3, …). This integer will become the univocal reference to the **Table**.
- When **Lightstreamer Server** sends *realtime updates* to the client, for each sent **Update Event** it specifies the number of the **item** within its **Table** (i.e. **Subscription**) and the **Subscription** number. Note that the **items** enumeration is decided by the **Metadata Adapter** and should be known by the client.
- If multiple **Data Adapters** are available in the **Adapter Set** associated to the **session**, the client has to specify the configured name of a specific **Data Adapter**. All the items in the **Item Group** must be supplied by the chosen **Data Adapter**.

4 The Text Mode protocol

In this chapter, we describe the ***Text Mode*** protocol. All the messages (**subscriptions**, **unsubscriptions**, received *realtime updates*, etc.) will be explained in a detailed way.
All actions are initiated by the client, which sends requests detailed through URL querystrings, i.e. in the general form *name1=value1&name2=value2&….* where all values should be protected by URL encoding as described by RFC 3986. Note that the encoding described by the html specification for the *application/x-www-form-urlencoded* MIME format is also supported.

4.1 Creating the Stream Connection

- **Method**: **POST**.
- **Protocol**: **HTTP** | **HTTPS**.
- **URI**: "**/lightstreamer/create_session.txt**".
- **Objective**: Session management; opening a new streaming Session.
- **Request parameters:**
  - **LS_user** = user name (used for authentication). This string should be interpreted and verified by the **Metadata Adapter**, so the developer is free to decide his own meaning. In simplified scenarios, the argument can be omitted, but authentication is still requested to the **Metadata Adapter** and a null user name is specified.
  - **LS_password** = (**optional**) user password (used for authentication). This string should be interpreted and verified by the **Metadata Adapter**, so the developer is free to decide his own meaning.
  - **LS_adapter_set** = logical name that identifies the **Adapter Set** (i.e. the **Metadata Adapter** and the related **Data Adapters**) that will serve and provide data for this **stream connection**.
  - **LS_requested_max_bandwidth** = (**optional**) max bandwidth requested by the client, expressed in kbps (it can be a decimal value). See the **Lightstreamer Glossary** for more information.
  - **LS_content_length** = (**optional**) Content-Length to be used for the connection contents. If too low or not present, the Content-Length is assigned by Lightstreamer Server, based on its own configuration.
  - **LS_keepalive_millis** = (**only if LS_polling is not "true"; optional**) longest inactivity time allowed for the connection. If such a long inactivity occurs, Lightstreamer Server sends a keepalive message. If too low, the Server may apply a configured minimum time. If too high, the Server will apply a configured maximum time. If not present, the keepalive time is decided by Lightstreamer Server, based on its own configuration. Anyway, the keepalive time used is notified to the client in the response header.
  - **LS_report_info** = (**optional**) if set to "true", asks the Server to notify, inside the response header, some limits and capabilities, specified in the current configuration, which can affect the interaction.
  - **LS_polling** = (**optional**) requests a "polling" connection. If set to "true", the Server will send only the updates that are ready at connection time and will exit immediately, keeping the session active for subsequent rebind requests.

- **LS_polling_millis** = (**only if LS_polling is "true"**) expected time between the closing of the connection and the next polling connection. Required by the Server in order to ensure that the underlying session is kept active across polling connections. If too high, the Server may apply a configured maximum time. Anyway, the timeout used is notified to the client at the end of the response.
- **LS_idle_millis** = (**only if LS_polling is "true"; optional**) time the Server is allowed to wait for an update to return, if none is present at request time. If zero or not specified, the Server response will be synchronous and might be empty. If positive, the Server response will be asynchronous and, if the specified timeout expires, might be empty. If too high, the Server may apply a configured maximum time.

Response:

- If an error occurred:
  - "ERROR\r\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a formal error occurred) or by the **Metadata Adapter** (if a notation error occurred). In particular, **err_code** can be:
    - 1 - user/password check failed
    - 2 - requested **Adapter Set** not available
    - 7 - licensed maximum number of sessions reached (this can only happen with some licenses)
    - 8 - configured maximum number of sessions reached
    - 9 - configured maximum server load reached
    - 10 - new sessions temporarily blocked
    - 11 - streaming is not available because of Server license restrictions (this can only happen with special licenses)
    - 14 - request for internal monitoring data not allowed with the current Server edition
    - <= 0 - the **Metadata Adapter** has refused the user connection; the code value is dependent on the specific **Metadata Adapter** implementation
  - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
  - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
- If no error occurred:
  - "OK\r\n"
+ "SessionId:" + **SessionId** + "\r\n"
[+ "ControlAddress:" + **ControlLink** + "\r\n"]
+ "KeepaliveMillis:" + **KeepaliveTime** + "\r\n"
+ "MaxBandwidth:" + **max_bandwidth** + "\r\n"
[+ "RequestLimit:" + **RequestLimit** + "\r\n"]
[+ "ServerName:" + **ServerName** + "\r\n"]
+ "\r\n"
    - **SessionId**: the **Lightstreamer Server** internal string representing the **Session**, this string must be sent with every following **Control Connection**.

- **ControlLink** (**optional**): if a **Link Control** (see section 2.1.3, Lightstreamer Server behind a Load Balancer) is specified in the **Lightstreamer Server** configuration file, this is the Internet address (or IP address) to which every following **Control Connection** must be opened. If this parameter is not present, it means that the client should open all the **Control Connections** to the same address to which it opened this **Stream Connection**.
- **KeepaliveTime**: the longest inactivity time (in milliseconds) guaranteed throughout connection life. For a stream connection, when no updates have been sent for this time, a probe signal is sent to the client. On the other hand, for a polling connection, if the response has not been supplied for this time, an empty response is issued. Not receiving any message for longer than this time may be the signal of a problem.
- **max_bandwidth**: the server-side bandwidth constraint on the connection data flow; the special value of 0 means that no limitation is applied; the special value of -1 means that no limitation is applied and no client-side limitation is allowed.
  Note that, but for the -1 case, the real bandwidth allowed to the connection may be less, because of the client-side constraint declared with the **LS_requested_max_bandwidth** parameter.
- **RequestLimit** (**optional**): the maximum length allowed by the Server for a client request. It is configured through the **request_limit** element in the Server configuration file. It is returned only if **LS_report_info=true** is specified in the request and the Server version is 3.4.4 or greater. Note that this limit should be set long enough that any single request can always be accepted; however, the limit may affect the batching of control requests (see section 4.3.7).
- **ServerName** (**optional**): the name assigned to the server socket which is handling the request. The name is assigned to a server socket through the <http_server> or <https_server> element in the Server configuration file. It is returned only if **LS_report_info=true** is specified in the request and the Server version is 3.4.6 or greater. However, the socket name is not returned upon a bind request (see the next session), though, with some particular configurations, it is possible that bind requests are handled by a server socket which is different than the one which served the session creation request.

▶ After this header, the **push content phase** starts; see section 4.5, for details.

4.2 Binding to an existing Session

- **Method**: **POST**.
- **Protocol**: **HTTP** | **HTTPS**.
- **URI**: "**/lightstreamer/bind_session.txt**".
- **Objective**: Session management; replacing a completely consumed connection in listening for an active Session.
- **Request parameters:**
  - **LS_session**= the **Lightstreamer Server** internal string representing the **Session** the client wants to ***bind*** to.

- **LS_requested_max_bandwidth**= (**optional**) max bandwidth requested by the client expressed in kbps (it can be a decimal value). See the **Lightstreamer Glossary** for more information.
  If missing, the default request not to limit the bandwidth is applied; this may cause a change of the current value.
- **LS_content_length** = (**optional**) Content-Length to be used for the connection contents. If too low or not present, the Content-Length is assigned by Lightstreamer Server, based on its own configuration.
- **LS_keepalive_millis** = (**only if LS_polling is not "true"; optional**) longest inactivity time allowed for the connection. If such a long inactivity occurs, Lightstreamer Server sends a keepalive message. If too low, the Server may apply a configured minimum time. If too high, the Server will apply a configured maximum time. If not present, the keepalive time is decided by Lightstreamer Server, based on its own configuration. Anyway, the keepalive time used is notified to the client in the response header.
- **LS_report_info** = (**optional**) if set to "true", asks the Server to notify, inside the response header, some limits and capabilities, specified in the current configuration, which can affect the interaction.
- **LS_polling** = (**optional**) requests a "polling" connection. If set to "true", the Server will send only the updates that are ready at connection time and will exit immediately, keeping the session active for subsequent rebind requests.
- **LS_polling_millis** = (**only if LS_polling is "true"**) expected time between the closing of the connection and the next polling connection. Required by the Server in order to ensure that the underlying session is kept active across polling connections. If too high, the Server may apply a configured maximum time. Anyway, the timeout used is notified to the client at the end of the response.
- **LS_idle_millis** = (**only if LS_polling is "true"; optional**) time the Server is allowed to wait for an update to return, if none is present at request time. If zero or not specified, the Server response will be synchronous and might be empty. If positive, the Server response will be asynchronous and, if the specified timeout expires, might be empty. If too high, the Server may apply a configured maximum time.

Response:

- If an error occurred:
  - "ERROR\r\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a formal error occurred) or by the **Metadata Adapter** (if a notation error occurred). In particular, **err_code** can be:
    - 3 - the session was initiated with a different and incompatible communication protocol
    - 11 - streaming is not available because of Server license restrictions (this can only happen with special licenses)
    - 14 - request for internal monitoring data not allowed with the current Server edition

    Only upon the first rebind on a session that has not been used yet, **err_code** can also be:
    - 7 - licensed maximum number of sessions reached (this can only happen with some licenses)
    - 8 - configured maximum number of sessions reached
  - "SYNC ERROR\r\n": when a ***synchronization*** error occurs, it means that the server does not recognize the received **SessionId**. In this case, the client should

create a new **Stream Connection** and re-**subscribe** to all the old **items** and related **field schemas** (see section 2.5, Content-Length management).

- "END\r\n" or "END\r\n" + **cause_code** + "\r\n": the requested session has just been forcibly closed on the Server side. See the asynchronous version of this notification, in section 4.5.6, End messages, for a general description of the case and for a reference on the **cause_code** possible values.
- A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
- **HTTP** error 500 (if an **HTTP** or protocol level error occurred).

- If no error occurred, the workflow is the same explained for the creation of a Stream Connection (section 4.1, Creating the Stream Connection).

4.3 Control connections

4.3.1 Subscription Control Connections

- **Method**: **POST**.
- **Protocol**: **HTTP** (Recommended) | **HTTPS**.
- **URI**: "**/lightstreamer/control.txt**".
- **Objective**: table (i.e. subscription) management (creation, activation, deletion).
- **Request parameters:**
  - **LS_session** = **SessionId** received from **Lightstreamer Server** at the beginning of the **Stream Connection**.
  - **LS_table** = progressive identification number of the **Table** (i.e. **Subscription**) to which the operation specified in **LS_op** parameter applies.
  - **LS_op** = **add** | **add_silent** | **start** | **delete**
    - **add** = creates and activate a new **table**. The **item group** specified in the **LS_id** parameter will be **subscribed** to and **Lightstreamer Server** will start sending *realtime updates* to the client immediately.
    - **add_silent** = creates a new **table**. The **item group** specified in the **LS_id** parameter will be **subscribed** to but **Lightstreamer Server** will not start sending *realtime updates* to the client immediately.
    - **start** = activate a **table** previously created with an "**add_silent**" operation. **Lightstreamer Server** will start sending *realtime updates* to the client immediately. When **LS_op=start**, none of the following request parameters must be sent, as the **table** content has been specified in the previous "**add_silent**" operation.
    - **delete** = deletes the specified **table**. All the related items will be unsubscribed from and **Lightstreamer Server** will stop sending *realtime updates* to the client immediately. When **LS_op=delete**, none of the following request parameters must be sent, as the **table** content has been specified in a previous "**add**" or "**add_silent**" operation.

Only if either "LS_op=add", "LS_op=add_silent" or "LS_op=start" the following parameters must be added:

- **LS_data_adapter** = (**optional**) configured name of one of the **Data Adapters** available in the **Adapter Set**. This **Data Adapter** must supply all the requested **items**. If not specified, then the default **Data Adapter** configured for the **Adapter Set** is requested.
- **LS_id** = identification name of the **item group** that the **Table** (i.e. **Subscription**) contains; this name is interpreted by the **Metadata Adapter**.
- **LS_schema** = identification name of the **field schema** related to the **items** in the **Table**; this name is interpreted by the **Metadata Adapter**.
- **LS_selector** = (**optional**) identification name of a **selector** related to the **items** in the **Table**; this name is interpreted by the **Metadata Adapter**.
- **LS_mode** = ***subscription mode*** of all the **items** in **Table**:
  - Values= **RAW** | **MERGE** | **DISTINCT** | **COMMAND;**
  - See the **Lightstreamer Glossary** for details about ***subscription modes***.
- **LS_requested_buffer_size** = (**optional**) dimension (expressed in number of update events) of the **buffers** related the **items** in the **Table**; see the **Lightstreamer Glossary** for more details.
  If not specified, the default buffer size is 1 if **LS_mode** is **MERGE** and unlimited if **LS_mode** is **DISTINCT**. Specify 0 to request an unlimited buffer (the Server will probably limit the buffer size, however).
  - Considered only if **LS_mode** is **MERGE** or **DISTINCT** and **LS_requested_max_frequency** is not set to **unfiltered**.
- **LS_requested_max_frequency** = (**optional**) **unfiltered** | maximum **update frequency** (expressed in updates/sec) for the **items** in the **Table**; see the **Lightstreamer Glossary** for more details.
  - **unfiltered**: **Lightstreamer Server** should forward each **update** as soon as possible (as in the unlimited frequency case), but also without losses.
    Considered only if **LS_mode** is **MERGE**, **DISTINCT** or **COMMAND**.
  - **Update frequency**: A decimal number can be supplied; the decimal separator should be a dot. A **0** value means no frequency limit.
    Considered only if **LS_mode** is **MERGE**, **DISTINCT** or **COMMAND** (in COMMAND mode, the maximum frequency applies to the UPDATE commands sent for each *key*).
  - Default value **= 0**.
- **LS_snapshot** = (**optional**) **true** | **false** | requested **snapshot length** (expressed in number of events) for the **items** in the ***Table***.
  - **true**: **Lightstreamer Server** should send the **snapshot** (if available) for the **items** contained in the **Table**.
    Considered only if **LS_mode** is **MERGE**, **DISTINCT** or **COMMAND**.
  - **false**: **Lightstreamer Server** must not send the **snapshot** for the **items** contained in the **Table**.
  - **Snapshot length**: Admitted only if **LS_mode** is **DISTINCT**. **Lightstreamer Server** should send the **snapshot** (if available) for the **items** contained in the **Table**, limiting the length to the requested value.
  - Default value **= false**.

Response:

- If an error occurred:
  - "ERROR\r\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a

formal error occurred) or by the **Metadata Adapter** (if a notation error occurred). In particular, **err_code** can be:

- 17 - bad **Data Adapter** name or default **Data Adapter** not defined
- 19 - request processing failed, possibly because of a race condition
- 21 - bad **Item Group** name
- 22 - bad **Item Group** name for this **Field schema**
- 23 - bad **Field schema** name
- 24 - **subscription mode** not allowed for an **Item**
- 25 - bad **Selector** name
- 26 - unfiltered dispatching not allowed for an **Item**, because a frequency limit is associated to the **Item**
- 27 - unfiltered dispatching not supported for an **Item**, because a frequency prefiltering is applied for the **Item**
- 28 - unfiltered dispatching is not allowed by the current license terms (for special licenses only)
- 29 - **RAW** mode is not allowed by the current license terms (for special licenses only)
- <= 0 - the **Metadata Adapter** has refused the subscription or unsubscription request; the code value is dependent on the specific **Metadata Adapter** implementation

- "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
- A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
- **HTTP** error 500 (if an **HTTP** or protocol level error occurred).

- If no error occurred:
  - "OK\r\n"

4.3.2 Item Groups and Field Schemas

As explained previously, the **Item Group** and the **Field Schema** specified in **Control Connections** are interpreted by the **Metadata Adapter**, and so the developer is free to decide his/her own meaning for these parameters.
It is suggested to use the following notation (because it is the clearest and most *debuggable* one):

- Specify **Item Groups** by their contents. For example a **pipe-separated item name list** could be used as the ***id***.
  - Example: LS_id=NASDAQ.MSFT|NASDAQ.YHOO|AFF.F|AFF.STM
- Specify **Field Schemas** by their contents as well. For example a **pipe-separated field name list** could be used as a **field schema**.
  - Example: LS_schema=LAST_PRICE|BID|ASK|VOLUME|TIME

4.3.3 Subscription reconfiguration Control Connections

- **Method**: **POST**.
- **Protocol**: **HTTP** (Recommended) | **HTTPS**.
- **URI**: "**/lightstreamer/control.txt**".

- **Objective**: changing subscription parameters for a currently subscribed table (i.e. subscription)
- **Request parameters:**
  - **LS_session** = **SessionId** received from **Lightstreamer Server** at the beginning of the **Stream Connection**.
  - **LS_table** = progressive identification number of a **Table** (i.e. **Subscription**) whose subscription parameters should be changed dynamically.
  - **LS_op** = **reconf**

  Actually, the only parameter that can be changed in this way is the maximum **update frequency**. Moreover, this parameter cannot be used to switch between filtered and unfiltered dispatching.
  - **LS_requested_max_frequency** = (**optional**) maximum **update frequency** (expressed in updates/sec) for the **items** in the **Table**; see the **Lightstreamer Glossary** for more details.
    - Admitted only if the Table is not **subscribed** to with **unfiltered** dispatching.
      A decimal number can be supplied; the decimal separator should be a dot. A **0** value means no frequency limit.
      Considered only if the **subscription mode** of the **Table** is **MERGE**, **DISTINCT** or **COMMAND** (in COMMAND mode the maximum frequency applies to the UPDATE commands sent for each *key*).
- **Response**:
  - If an error occurred:
    - "ERROR\r\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a formal error occurred) or by the **Metadata Adapter** (if a notation error occurred). In particular, **err_code** can be:
      - 13 - modification not allowed because the **Table** is configured for **unfiltered** dispatching
      - 19 - request processing failed, possibly because of a race condition
    - "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
    - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
    - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
  - If no error occurred:
    - "OK\r\n"

4.3.4 Session constraints Control Connections

- **Method**: **POST**.
- **Protocol**: **HTTP** (Recommended) | **HTTPS**.
- **URI**: "**/lightstreamer/control.txt**".
- **Objective**: session constraints management (bandwidth modification).
- **Request parameters:**
  - **LS_session** = **SessionId** received from **Lightstreamer Server** at the beginning of the **Stream Connection**.

- **LS_op** = **constrain**
- **LS_requested_max_bandwidth** = (**optional**) max bandwidth requested by the client expressed in kbps (it can be a decimal value). See the **Lightstreamer Glossary** for more information.
  If missing, the default request not to limit the bandwidth is applied; this may cause a change of the current value.

Response:

- If an error occurred:
  - "ERROR\n\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a formal error occurred) or by the **Metadata Adapter** (if a notation error occurred). No error case is possible at the current stage.
  - "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
  - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
  - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
- If no error occurred:
  - "OK\r\n"

4.3.5 Asynchronous request for session rebind Control Connections

Method: **POST**.

Protocol: **HTTP** (Recommended) | **HTTPS**.

URI: "**/lightstreamer/control.txt**".

Objective: Session management; forcing the Server to close the current connection related to an existing session and ask for a rebind.

Request parameters:

- **LS_session** = **SessionId** associated by **Lightstreamer Server** to the session; received by the client at the beginning of the **Stream Connection**.
- **LS_op** = **force_rebind**

Response:

- If an error occurred:
  - "ERROR\n\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** are specified by the **Lightstreamer Server Kernel**.
    No error case is possible at the current stage.
  - "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
  - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
  - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
- If no error occurred:
  - "OK\r\n"

4.3.6 Session asynchronous destroy Control Connections

- **Method**: **POST**.
- **Protocol**: **HTTP** (Recommended) | **HTTPS**.
- **URI**: "**/lightstreamer/control.txt**".
- **Objective**: Session management; forcing closure of an existing session.
- **Request parameters:**
  - **LS_session** = **SessionId** associated by **Lightstreamer Server** to the session; received by the client at the beginning of the **Stream Connection**; received by a Metadata Adapter through specific notifications.
  - **LS_op** = **destroy**
- **Response**:
  - If an error occurred:
    - "ERROR\n\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** are specified by the **Lightstreamer Server Kernel**.
      No error case is possible at the current stage.
    - "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
    - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
    - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
  - If no error occurred:
    - "OK\r\n"

4.3.7 Batching of Control Requests

Batching of multiple different **Control Requests** in a single **HTTP** or **HTTPS Connection** is possible. The **Requests** have to be written in the content part of the **HTTP** or **HTTPS** message on different lines of text (i.e. the **Requests** have to be separated by "\r\n", while the last **Request** should not be terminated by "\r\n"). **Control Requests** of all kinds can be mixed together, though this is especially meaningful for subscription requests. The management of the various **requests** can be performed in parallel.

The **Requests** are independent from one another and are processed in sequence. The output lines generated by the different **Requests** are joined and returned in the same order. However, if one of the **Requests** should cause a static **HTML** error page or an **HTTP** error 500 to be issued, then the processing would be stopped and only that answer would be returned.

Note: if, for testing purpose, the **HTTP GET** method is used, this batching syntax is not available.

4.4 Sending Messages

4.4.1 Synchronous version

- **Method**: **POST**.
- **Protocol**: **HTTP** (Recommended) | **HTTPS**.

- **URI**: "**/lightstreamer/send_message.txt**".
- **Objective**: Sending a custom message to **Lightstreamer Server** and waiting for the elaboration outcome. A message is a pure block of text to be interpreted and processed by the **Metadata Adapter**. For this reason, a message can only be sent in the context of an already established push **session**, for which specific user credentials have been supplied and a specific **Adapter Set** has been associated.
- **Request parameters:**
  - **LS_session** = **SessionId** received from **Lightstreamer Server** at the beginning of the **Stream Connection**.
  - **LS_message** = any text string. This string should be interpreted and verified by the **Metadata Adapter**, so the developer is free to decide his own meaning.
- **Response**:
  - If an error occurred:
    - "ERROR\r\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a formal error occurred) or by the **Metadata Adapter** (if a notation error occurred). In particular, **err_code** can be:
      - <= 0 - the **Metadata Adapter** has refused the message; the code value is dependent on the specific **Metadata Adapter** implementation
    - "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
    - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
    - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
  - If no error occurred:
    - "OK\r\n"

4.4.2 Asynchronous version

- **Method**: **POST**.
- **Protocol**: **HTTP** (Recommended) | **HTTPS**.
- **URI**: "**/lightstreamer/send_message.txt**".
- **Objective**: Sending a custom message to **Lightstreamer Server** to be processed asynchronously and freeing the connection as soon as possible; any elaboration outcome will be received through the push contents. A message is a pure block of text to be interpreted and processed by the **Metadata Adapter**. For this reason, a message can only be sent in the context of an already established push **session**, for which specific user credentials have been supplied and a specific **Adapter Set** has been associated.
- **Request parameters:**
  - **LS_session** = **SessionId** received from **Lightstreamer Server** at the beginning of the **Stream Connection**.
  - **LS_sequence** = an alphanumeric identifier (the underscore character is also allowed), used to identify a subset of messages to be managed in sequence, based on the assigned progressive numbers.

All messages associated with the same **sequence** name that have successfully been sent to the Server are guaranteed to be processed sequentially, according to the associated progressive numbers.
In case a message is received and the messages for all the previous numbers expected haven't been received yet, the latter numbers can be skipped according to the timeout specified with the message.
If the special “UNORDERED_MESSAGES” **sequence** name is used, then the associated messages are processed immediately, possibly concurrently, with no ordering constraint. Yet, progressive numbers for which no messages have been received after a server-side timeout may be notified by **Lightstreamer Server** as skipped.

- **LS_msg_prog** = the progressive number of the message within the specified sequence, starting from 1.
- **LS_max_wait** = (**optional**) the maximum time the Server can wait before processing the message if one or more of the preceding messages for the same **sequence** have not been received.
  If too high or not specified, the timeout is assigned by **Lightstreamer Server**, based on its own configuration.
  If the special “UNORDERED_MESSAGES” **sequence** name is used, the setting is ignored, as a server-side timeout is applied.
- **LS_message** = any text string. This string should be interpreted and verified by the **Metadata Adapter**, so the developer is free to decide his own meaning.

Response:

- If an error occurred:
  - "ERROR\r\n" + **err_code** + "\r\n" + **err_msg +** "\r\n", where **err_code** and **err_msg** are specified by the **Lightstreamer Server Kernel**.
    In particular, **err_code** can be:
    - 32 - The specified progressive number is too low; either a message with this number has already been enqueued (and possibly processed) or the number has already been skipped by timeout (the exact case cannot be determined).
    - 33 - The specified progressive number is too low; a message with this number has already been enqueued (and possibly processed).
  - "SYNC ERROR\r\n", when a ***synchronization*** error occurred (normally, the server does not recognize the **SessionId**);
  - A static **HTML** error page (if the current Server configuration does not support the **Text mode** protocol).
  - **HTTP** error 500 (if an **HTTP** or protocol level error occurred).
- If no error occurred:
  - "OK\r\n", which only means that the message has been enqueued for processing.

4.5 Push Contents

The data sent on the **Stream Connections** consist of 8 different types of messages:

- **Update** messages.
- **Overflow** messages.
- **End-of-Snapshot** messages.

- **Asynchronous Send Message outcome** messages.
- **Probe** messages.
- **End** messages.
- **Loop** messages.
- **Push error** messages.

4.5.1 Update messages

Update messages contain *snapshot values* or *realtime updates* for a certain **item**. The **update** message syntax is the following:

- **table** + "," + **item** + "|" + **values** + "\r\n"

where:

- **table** is the number of the **Table**;
- **item** is the number of the item within the **Table**, assigned by the **Metadata Adapter** during the **subscription** operation;
- **values** is the string containing the **pipe-separated encoded field** values.
  The number of field values received is the same as the number of fields in the **field schema** specified in the subscription operation.
  See below for details on the encoding/decoding rules.

Here is an example:

- **table** = 1;
- for the **items** in the **Table** the **field schema** = `LAST_PRICE|PERCENTAGE_CHANGE|LAST_CHANGE_TIME` is specified;
- the **Table** has only one **item**.

The updates commands for the **Table** could be like the following:

- `1,1|10.5|+12.5%|12:55:56`
- `1,1|10.4|+12.4%|12:57:00`

If **snapshot** information has been requested for an **item**, the very first **updates** for the **item** may carry the current state of the **item** (the **snapshot**) rather than a state update. In particular, if the subscription mode is MERGE, the first **update** carries **snapshot** information, while if the subscription mode is DISTINCT or COMMAND then all the **updates** before the **End-of-Snapshot** message carry **snapshot** information.

Value Encoding

Each **field value** can be any UNICODE text.

All the **values** to be sent by the Server to the clients undergo a two phase encoding.
In the first phase, a simple compression mechanism is applied to the values in order to reduce the amount of data sent to the Client.

- If a **field** of an **item** is unchanged with respect to the previous delivery to the Client, then it is transformed into an empty string.

- At the same time, real empty strings and null strings are considered and transformed into **"$"** and **"#"**, respectively, in order to be sent to the Client.
- On the other hand, data strings starting with **"$"** or with **"#"** are encoded by doubling the first character.

In the second phase, an escaping mechanism is applied on the values, so that data can be sent to the Client as ASCII text. Alphanumeric characters and most ASCII symbols are transmitted in ASCII code. All other characters are transmitted as UTF-16 escape sequences of the form "\uXXXX" or "\uXXXX\uYYYY".
This escaping phase is also needed to escape those characters that will be used as field delimiters in update packaging ("|" in **Text mode** and "" in **JavaScript mode**).

The Client should decode the received values by using the following algorithm:

- if the value is equal to "$", then it should be transformed into an empty string ("");
- otherwise, if the value is equal to "#", then it should be transformed into a null value;
- otherwise, if the value is equal to "" (empty string), then the Client should consider it unchanged with respect to the previous value of the same **field** of the same **item** (note that this case will never be possible on the first update for an **item**);
- otherwise (hence the value is an actual string), the Client should check for a leading "$" or "#" and remove it; then it should decode any "\uXXXX" UTF-16 escape sequence inside the value and recognize possible UTF-16 surrogate pairs, to come up with a string made of UNICODE characters.
  NOTE: actually, unmatched high surrogates and low surrogates may be found, as they are not excluded by the Java Adapter Interface, which is based on java String objects. The management of such code units is left to the application.

Alternatively, the decoding of the UTF-16 escape sequences can be performed before decompression (i.e. before the evaluation of the "$", "#" and "" values), rather than after it. This is possible, because "\" characters in data values are escaped, while "$" and "#" characters in data values are not escaped. Note that this is how the values would be decoded in **JavaScript mode**, where the UTF-16 escape sequences would be directly handled by the JavaScript parser.

Here is an example involving encoded data:

- **table** = 1;
- for the **items** in the **Table** the **field schema** = `CATEGORY|HEADLINE|BODY` is specified;
- the **Table** has only one **item**.

The updates commands for the **Table** could be like the following:

- `1,1|SPORT|ITALY WINS SOCCER WORLD CHAMPIONSHIPS|$`
- `1,1||ZIDANE ELECTED \u0022MVP\u0022|`

Note that the BODY field is empty for both updates.

4.5.2 End-of-Snapshot messages

End-of-Snapshot messages notify that the subsequent **updates** for an item will no longer carry **snapshot** information, but rather state updates. The **End-of-Snapshot** message syntax is the following:

- **table** + "," + **item** + "," + "EOS" + "\r\n"

where:

- **table** is the number of the **Table**;
- **item** is the number of the item within the **Table**, assigned by the **Metadata Adapter** during the subscription operation;

End-of-Snapshot notification are received only for **items** subscribed to in DISTINCT or COMMAND mode with a request for **snapshot** informations.

4.5.3 Overflow messages

An **Overflow** message notifies that one or more **updates** for an **item** have been dropped because of internal buffer limitations in Lightstreamer Server. The **Overflow** message syntax is the following:

- **table** + "," + **item** + "," + "OV" + **overflowSize** + "\r\n"

where:

- **table** is the number of the **Table**;
- **item** is the number of the item within the **Table**, assigned by the **Metadata Adapter** during the **subscription** operation;
- **overflowSize** is the number of consecutive events dropped for the indicated **item**.

This notification can only be sent if the **item** was **subscribed** to in RAW or COMMAND mode, or if it was **subscribed** to in MERGE or DISTINCT mode and unfiltered dispatching was requested. These are the modes for which events dropping is not allowed (for filtered COMMAND mode, this applies to "ADD" and "DELETE" events only). So, in all these cases, whenever the Server has to drop an event because of resource limits, it notifies the client in this way.

4.5.4 Asynchronous Send Message outcome messages

Corresponding with any successful **Asynchronous Send Message** request, a notification of the elaboration outcome is received after the elaboration has terminated, unless the current **session** terminates first.
Within each requested **sequence**, the notifications are guaranteed to be received ordered by the message progressive numbers. The notifications are guaranteed to start from 1 and to leave no holes. Hence, all message numbers skipped by timeout are also notified.
As an exception to the above, for all messages for which the "UNORDERED_MESSAGES" **sequence** was specified, the notifications are only guaranteed to be unique for each progressive number.

The **Asynchronous Send Message** outcome message syntax is the following:

- **"**MSG**"** + "," + **sequence** + "," + **prog** + "," + "DONE**"** + "\r\n"
- **"**MSG**"** + "," + **sequence** + "," + **prog** + "," + "ERR" + "," + **err_code** + "," + **err_msg** + "\r\n"

The first syntax notifies successful elaboration, where:

- **sequence** is the sequence identifier specified in the related request;
- **prog** is the message progressive number specified in the related request.

The second syntax notifies either unsuccessful elaboration or skipped elaboration. In the former case, the <**sequence**, **prog**> pair is also related to a specific request; in the latter one the pair is inferred from other requests. **err_code** and **err_msg** could be specified either by the **Lightstreamer Server Kernel** (if a formal error occurred) or by the **Metadata Adapter** (if a notation error occurred).
In particular, **err_code** can be:

- 34 - the request has been unexpectedly refused as illegal by the **Metadata Adapter** (i.e. a NotificationException was thrown);
- 35 - the elaboration has unexpectedly failed for any reason;
- 38 - the specified progressive number has been skipped by timeout;
- 39 - the specified progressive number and some consecutive preceding numbers have been skipped by timeout; only in this case, **err_code** is an integer and specifies how many progressive numbers have been skipped; the notification pertains to all those progressive numbers;
- <= 0 - the **Metadata Adapter** has refused the message; the code value is dependent on the specific **Metadata Adapter** implementation.

4.5.5 Probe messages

When no *realtime updates* are available, **probe** messages are sent in order to keep the **HTTP** connection alive through proxies and firewalls. Probe messages can be ignored by the client or they can be handled as heartbeat signals.
The **probe** message syntax is the following:

- "PROBE\r\n"

4.5.6 End messages

A **Stream Connection** might be closed explicitly on the Server side.
In such a case, the Server sends an **end** message and closes the **Stream Connection**. The client cannot rebind again to the **session**; moreover, it is advised not to try to recover the problem by opening a new **session** immediately.
The **end** message syntax is the following:

- "END " + **cause_code** + "\r\n"
- "END\r\n"

where **cause_code** indicate the original cause of the interruption; it can be:

- 31 The session was closed by the administrator through a "**destroy**" request;
- 32 The session was closed by the administrator through JMX;
- 33, 34 An unexpected error occurred on the Server while the **session** was in activity;
- 35 The **Metadata Adapter** does not allow more than one **session** for the current **user** and has requested the closure of the current **session** upon opening of a new **session** for the same **user** by some client;
- 40 A manual rebind to the same **session** has been performed by some client;
- a different code value is possible and signals an unexpected cause;
- a missing code (i.e. the second syntax case) signals that the cause could not be determined.

The following codes can only be received at the moment a **session** is used for the first time:

- 7 Licensed maximum number of sessions reached (this can only happen with some licenses);
- 8 Configured maximum number of sessions reached.

Any trailing lines received after this message should be ignored.

4.5.7 Loop messages

When the **Stream Connection** is reaching its natural completion, **Lightstreamer Server** sends a **loop** message in order to tell the client to open a new **Stream Connection** in order to rebind to the same **session**.
This may happen because the connection has exhausted its **Content-Length** (see section 2.5, Content-Length management) or because a polling connection was requested.
The **loop** message syntax is the following:

- "LOOP " + **holding_time** + "\r\n"
- "LOOP\r\n"

where:

- **holding_time** is the time (in milliseconds) the **session** is guaranteed to remain active inside **Lightstreamer Server** (useful for polling requests).
  Note: **Lightstreamer Server** also applies a supplementary holding time to take computation and connection delays into account.
- The first form is used only if polling mode was requested on the connection.
- If the second form is used, the implied holding time is 0 and the client should rebind to the **session** as soon as possible.

Any trailing lines received after this message should be ignored.

4.5.8 Push error messages

Actually, no asynchronous error messages are used.
If any abnormal condition ever occurred such that it would prevent streaming from continuing correctly, then this would just cause the connection to be closed.
If closing the connection gracefully were possible, an END message with a 33 or 34 code would be issued.

5 Tests and Examples

5.1 Test environment

For testing the examples described in this chapter, it is possible to use the test installation of **Lightstreamer Server** included in the distribution package (follow the instructions in GETTING_STARTED.TXT to run the Server). We assume in the examples below that the Server instance is available at the "**localhost**" address.

This **Lightstreamer Server** test installation is configured as follows:

- The Server listens for **HTTP** requests on a configured port. See the <port> element in the Server configuration file. Let's assume in the examples below that the port is 8080.
- The SSL server is not active. The tester could use **HTTPS** connections instead of **HTTP** connections when testing via Web Browser, provided that the SSL server is activated.
- The Server runs the "**DEMO**" **Adapter Set**, which, in turn, contains several **Data Adapters** to manage all the demo-related items. In the examples below, we will be using the "**QUOTE_ADAPTER**" **Data Adapter**.
  Hence, the **LS_adapter_set** parameter should be set to "**DEMO**" and the **LS_data_adapter** parameter should be set to "**QUOTE_ADAPTER**" (see section 4.1, Creating the Stream Connection).

- The chosen **Data Adapter** supplies the following **items**:
  - item1
  - .....
  - item30

  corresponding to 30 simulated stock quotes.

  For each item, the **Data Adapter** supplies the following fields:
  - stock_name
  - time
  - last_price
  - pct_change
  - min
  - max
  - ask
  - bid
  - bid_quantity
  - ask_quantity
  - ref_price
  - open_price

The **Data Adapter** generates random values and is able to supply **Snapshots**.

- The demo **Metadata Adapter** is implemented as follows:
  - No authentication is performed, so the **LS_user** parameter **can be blank**;
  - No custom ***id***s for **Item Groups** are available. Any set of **items** can be requested by specifying the space-separated list of **item names** (to fit them into HTTP commands, the spaces will be represented through '+' characters);
  - No custom **field schema** names are available. Any set of **fields** can be requested by specifying the space-separated list of **field names** (to fit them into HTTP commands, the spaces will be represented through '+' characters).

The following examples can be tested in two different ways:

- **By telnet**: commands can be sent to the test **Lightstreamer Server** by executing:
  `telnet localhost 8080`
  and then by sending **HTTP** commands (**remember to send a blank line after every command**).
- **By Web Browser**: commands are sent by putting the specified command URL in the browser address bar.

  Important: when making these tests with a web browser, it is possible that (while creating the **Stream Connection**) the user should wait a few minutes before the browser shows the SessionId. This is due to web browsers buffering (we are using a non-HTML mode in a HTML browser just for didactic purpose).

Please note that, since the examples discussed below have been written in order to be tested via **telnet** or via a **Web Browser**, the **HTTP** method used is always the **GET** one, though only the **POST** one is officially supported.

For every test, the user will need:

- A **Stream Connection**, that is:
  - a permanent telnet connection through which *realtime updates* will appear;
  - a browser window in which *realtime updates* will appear.
- One or more **Control Connections**, that is:
  - auxiliary telnet connections (made from different shells) through which commands are sent to the server;
  - an auxiliary browser window through which commands are sent to the server.

Remember that at the beginning of the **Stream Connection**, **Lightstreamer Server** sends the **SessionId** to the client (see section 2.3, Stream Connection, Control Connections and Polling connections), so the user has to copy the received value and to fill with it the **LS_session** parameter of all the **Control Connections** of the test.

5.2 Table (i.e. Subscription) management

In this section, some examples of subscription and unsubscription operations are listed.

5.2.1 Basic workflow

In this test one **item** will be **subscribed** to in one **Table**.

- **Stream Connection Creation**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user= HTTP/1.0`
    - Send a blank line
    - Server command: `OK`
    - Server command: SessionId:S9cb4758037a95c01T0439915 (**copy the received value**)
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user=`
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
- **Subscription**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2+item11+item18&LS_schema=last_price+time+pct_change+bid_quantity+bid+ask+ask_quantity+min+max+ref_price+open_price&LS_mode=MERGE&LS_snapshot=true HTTP/1.0`
    - Send a blank line
    - The server starts sending realtime updates on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2+item11+item18&LS_schema=last_price+time+pct_change+bid_quantity+bid+ask+ask_quantity+min+max+ref_price+open_price&LS_mode=MERGE&LS_snapshot=true`
    - The server starts sending realtime updates on the **Stream Connection** browser.
- **Unsubscription**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete HTTP/1.0`
    - Send a blank line
    - The server stops sending realtime updates on the **Stream Connection** shell.
  - **Via browser**:

- Open the URL: http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete
- The server stops sending *realtime updates* on the **Stream Connection** browser.

- ***Note***:
  - before the first **Control Connection** on the **Stream Connection** shell/browser the server sends some **probe** messages;
  - *realtime updates* concern 15 **items** (1..15):
  - *realtime updates* are in the following format (table = 2 is the assigned **table** number):
    - `2,2|19,63|13:22:40|22,0|30500|19,61|19,63|88000||||`
  - the **length** of the received **pipe-separated** string is the same as the **field schema length**.

5.2.2 Double subscription

In this test two **items** will be **subscribed** to in different **tables (**table 1 and table 2).

- **Stream Connection Creation**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user= HTTP/1.0`
    - Send a blank line
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
  - **Via browser**:
    - Open the URL: http://localhost:8080/lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user=
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)

- **Subscription#1**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true HTTP/1.0`
    - Send a blank line
    - The server starts sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true

- The server starts sending *realtime updates* on the **Stream Connection** browser.

- **Subscription#2**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item18&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true HTTP/1.0`
    - Send a blank line
    - The server starts sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item18&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true`
    - The server starts sending *realtime updates* on the **Stream Connection** browser.

- **Unsubscription#1**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete HTTP/1.0`
    - Send a blank line
    - The server stops sending realtime updates on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete`
    - The server stops sending *realtime updates* on the **Stream Connection** browser.

- **Unsubscription#2**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete HTTP/1.0`
    - Send a blank line
    - The server stops sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete`
    - The server stops sending *realtime updates* on the **Stream Connection** browser.

- ***Note***:
  - *realtime updates* concern 1 **item**:
  - *realtime updates* are in the following format (table = 1 or 2):

- `2,1|6,55|-8,9|13:34:59`
- `1,1|15,81|-1,74|13:34:59`

- the **length** of the received **pipe-separated** string is the same as the **field schema length**.

5.2.3 Field schema change

In this test, one **item** will be **subscribed** to and then it will be re-**subscribed** to (within a different **Table**) with a different **field schema**.

- **Stream Connection Creation**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user= HTTP/1.0`
    - Send a blank line
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user=`
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
- **Subscription#1**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true HTTP/1.0`
    - Send a blank line
    - The server starts sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true`
    - The server starts sending realtime updates on the **Stream Connection** browser.
- **Unsubscription#1**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**

- Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete HTTP/1.0`
- Send a blank line
- The server stops sending *realtime updates* on the **Stream Connection** shell.

- **Via browser**:
  - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete`
  - The server stops sending realtime updates on the **Stream Connection** browser.

- **Subscription#2**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: GET `/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+time+pct_change+bid_quantity+bid+ask+ask_quantity+min+max+ref_price+open_price&LS_mode=MERGE&LS_snapshot=true HTTP/1.0`
    - Send a blank line
    - The server starts sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+time+pct_change+bid_quantity+bid+ask+ask_quantity+min+max+ref_price+open_price&LS_mode=MERGE&LS_snapshot=true`
    - The server starts sending *realtime updates* on the **Stream Connection** browser.

- **Unsubscription#2**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete HTTP/1.0`
    - Send a blank line
    - The server stops sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete`
    - The server stops sending *realtime updates* on the **Stream Connection** browser.

- ***Note***:
  - *realtime updates* concern 1 **item**:
  - the first time, *realtime updates* are in the following format (table = 1):
    - `1,1|15,81|-1,74|13:34:59`
  - the first time, the **length** of the received **pipe-separated** string is the same as the "last_price pct_change time" **field schema length**.

- the second time, *realtime updates* are in the following format (table = 2):
  - `2,2|19,63|13:22:40|22,0|30500|19,61|19,63|88000||||`
- the second time, the **length** of the received **pipe-separated** string is the same as the "last_price time pct_change bid_quantity bid ask ask_quantity min max ref_price open_price" **field schema length**.

5.2.4 Snapshot synchronization

In this test one **item** will be **subscribed** to in one **table** with the **Snapshot *synchronization*** mode (see section 2.6, External Snapshots).

- **Stream Connection Creation**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user= HTTP/1.0`
    - Send a blank line
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user=`
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
- **Subscription**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add_silent&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE HTTP/1.0`
    - Send a blank line
    - The server only sends probe messages on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add_silent&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE`
    - The server only sends probe messages on the **Stream Connection** browser.
- **Reception start**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**

- Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=start HTTP/1.0`
- Send a blank line
- The server starts sending *realtime updates* on the **Stream Connection** shell.

- **Via browser**:
  - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=start`
  - The server starts sending *realtime updates* on the **Stream Connection** browser.

- **Unsubscription**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete HTTP/1.0`
    - Send a blank line
    - The server stops sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete`
    - The server stops sending *realtime updates* on the **Stream Connection** browser.

- ***Note***:
  - no *realtime update* is sent before the start command;
  - *realtime updates* concern 1 **item**;
  - *realtime updates* are in the following format (table = 1):
    - `1,1|15,81|-1,74|13:34:59`
  - the **length** of the received **pipe-separated** string is the same as the “last_price pct_change time” **field schema length**.

5.2.5 Multiple subscriptions of the same item

In this test we will **subscribe** to the same **item** in two different **Tables** (with different **field schemas**).

- **Stream Connection Creation**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user= HTTP/1.0`
    - Send a blank line
    - Server command: `OK`
    - Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)
  - **Via browser**:

- Open the URL: `http://localhost:8080/lightstreamer/create_session.txt?LS_adapter_set=DEMO&LS_user=`
- Server command: `OK`
- Server command: `SessionId:S9cb4758037a95c01T0439915` (**copy the received value**)

- **Subscription#1**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command:
      ```
      GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true HTTP/1.0
      ```
    - Send a blank line
    - The server starts sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+pct_change+time&LS_mode=MERGE&LS_snapshot=true`
    - The server starts sending *realtime updates* on the **Stream Connection** browser.
- **Subscription#2**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command:
      ```
      GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+time+pct_change+bid_quantity+bid+ask+ask_quantity+min+max+ref_price+open_price&LS_mode=MERGE&LS_snapshot=true HTTP/1.0
      ```
    - Send a blank line
    - The server starts sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=add&LS_data_adapter=QUOTE_ADAPTER&LS_id=item2&LS_schema=last_price+time+pct_change+bid_quantity+bid+ask+ask_quantity+min+max+ref_price+open_price&LS_mode=MERGE&LS_snapshot=true`
    - The server starts sending *realtime updates* on the **Stream Connection** browser.
- **Unsubscription#1**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command:
      ```
      GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete HTTP/1.0
      ```
    - Send a blank line

- The server stops sending *realtime updates* on the **Stream Connection** shell.

- **Via browser**:
  - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=1&LS_op=delete`
  - The server stops sending *realtime updates* on the **Stream Connection** browser.

- **Unsubscription#2**:
  - **Via telnet**:
    - Execute: **`telnet localhost 8080`**
    - Send command: `GET /lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete HTTP/1.0`
    - Send a blank line
    - The server stops sending *realtime updates* on the **Stream Connection** shell.
  - **Via browser**:
    - Open the URL: `http://localhost:8080/lightstreamer/control.txt?LS_session=S9cb4758037a95c01T0439915&LS_table=2&LS_op=delete`
    - The server stops sending *realtime updates* on the **Stream Connection** browser.

- ***Note***:
  - *realtime updates* concern 1 **item in three different tables**:
  - *realtime updates* are in the following format (table = 1 or 2):
    - `1,1|16,72|3,91|13:55:21`
    - `2,1|16,72|13:55:21|3,91|5500|16,71|16,72|98000||16,72||`
    - `2,1|16,8|13:55:22|4,41|88000|16,77|16,8|52500||16,8||`
  - the **length** of the received **pipe-separated** string for **Table 1** is the same as the related **field schema** (“last_price pct_change time”) **length**.
  - the **length** of the received **pipe-separated** string for **Table 2** is the same as the related **field schema** (“last_price time pct_change bid_quantity bid ask ask_quantity min max ref_price open_price”) **length**.
  - *realtime updates* are duplicated.