



TLCP Specification

Text Lightstreamer Client Protocol

Version: 2.5.0
Target: Lightstreamer Server v. 7.4.0 or greater
Last updated: 19/3/2024

Table of contents

1	INTRODUCTION.....	5
	Protocol Versions.....	5
	Main Concepts.....	5
	Transports.....	6
	Session Life Cycle.....	6
	<i>Session Recovery</i>	8
	Subscription Data Model.....	9
	Requests, Responses, and Notifications.....	10
	Request Syntax.....	11
	<i>HTTP Transport</i>	11
	<i>WS Transport</i>	12
	Common Response and Notification Syntax.....	13
	WORKFLOW EXAMPLES.....	15
	Basic Subscription.....	15
	Double Subscription.....	16
	Field Schema Change.....	17
	Session Rebinding.....	18
	<i>HTTP Loop and Rebind</i>	18
	<i>HTTP Loop and Failed Rebind</i>	19
	<i>WS Long Polling</i>	21
2	REQUEST/RESPONSE REFERENCE.....	22
	Session Creation Request.....	22
	Session Binding Request.....	24
	Session Creation and Binding Responses.....	26
	<i>Successful Session Creation or Binding Response</i>	26
	<i>Other Session Creation or Binding Responses</i>	28
	Control Requests.....	28
	<i>Common Parameters</i>	29
	<i>Subscription Request</i>	29
	<i>Unsubscription Request</i>	32
	<i>Subscription Reconfiguration Request</i>	32
	<i>Session Constrain Request</i>	33
	<i>Force Session Rebind Request</i>	34
	<i>Session Destroy Request</i>	35
	MPN Control Requests.....	36
	<i>MPN Device Registration Request</i>	36
	<i>MPN Device Badge Reset</i>	37
	<i>MPN Subscription Activation</i>	38
	<i>MPN Subscription Reconfiguration</i>	40
	<i>MPN Subscription Deactivation</i>	42

Message Send Request.....	42
Control Responses.....	44
<i>Successful Control Response</i>	44
<i>Other Control Responses</i>	45
Client Heartbeat.....	45
<i>Successful Response</i>	46
<i>Other Responses</i>	47
WebSocket establishment check.....	47
<i>Successful Response</i>	48
<i>Other Responses</i>	48
3 NOTIFICATION REFERENCE.....	49
Real-Time Update.....	49
<i>Decoding the Pipe-Separated List of Values</i>	49
<i>Snapshot vs Real-Time Updates</i>	51
<i>Notification Examples</i>	51
<i>Notification Examples with the use of “diff” values</i>	52
Other Subscription-Related Notifications.....	53
<i>Successful Subscription</i>	53
<i>Successful Subscription with Command Mode</i>	54
<i>Successful Unsubscription</i>	54
<i>End of Snapshot</i>	55
<i>Snapshot Clearing</i>	55
<i>Overflow</i>	56
<i>Subscription Reconfiguration</i>	56
MPN-Related Notifications.....	57
<i>MPN Device Registered</i>	57
<i>MPN Device Badge Reset</i>	58
<i>MPN Subscription Activated</i>	58
<i>MPN Subscription Reconfigured</i>	59
<i>MPN Subscription Deactivated</i>	59
Message-Related Notifications.....	59
<i>Message Successfully Sent</i>	60
<i>Message Send Failed</i>	60
Session-Related Notifications.....	61
<i>Session Constraints Changed</i>	61
<i>Time Synchronization</i>	61
<i>Client IP Address</i>	62
<i>Server Name</i>	62
<i>Progressive of Last Data Notification</i>	63
<i>No Operation</i>	63
<i>Probe (Keep-Alive)</i>	63
<i>Stream Connection Loop</i>	64
<i>Stream Connection End</i>	64
4 SPECIAL USE CASES.....	66

Session Creation and Control Combo Request.....	66
<i>Session Creation and Multiple Control Combo Request</i>	67
Use of HTTP GET in Place of HTTP POST.....	68
5 HANDS ON.....	69
About cURL.....	69
Server Setup.....	69
Basic Subscription.....	70
<i>Connection and Session Creation</i>	70
<i>Subscription to an Item</i>	71
<i>Receive Some Data</i>	72
<i>Unsubscription from an Item</i>	72
<i>Disconnection</i>	73
Session Rebinding.....	74
<i>Session Creation and Subscription to Items</i>	74
<i>Receive Data Until the Content-Length Is Reached</i>	75
<i>Rebind the Session on a New Stream Connection</i>	75
Session Recovery.....	76
<i>Session Creation and Subscription to an Item</i>	76
<i>Receive Data and Interrupt the Connection</i>	77
<i>Recover the Session on a New Stream Connection</i>	77
Sending And Receiving Messages.....	78
<i>Session Creation and Subscription to an Item</i>	78
<i>Send a Message and Receive its Echo</i>	79
Mobile Push Notifications.....	80
<i>Additional Configuration</i>	81
<i>Session Creation</i>	82
<i>MPN Device Registration</i>	83
<i>MPN Subscription Activation</i>	84
<i>Disconnection and Reconnection</i>	85
<i>MPN Subscription Deactivation</i>	86
APPENDIX A: SESSION ERROR CODES.....	88
APPENDIX B: CONTROL ERROR CODES.....	91
APPENDIX C: SERVER HTTP HEADERS.....	95
HTTP Request Headers.....	95
<i>General Header Fields</i>	95
<i>Request Header Fields</i>	95
<i>Entity Header Fields</i>	96
HTTP Response Headers.....	96
<i>General Header Fields</i>	96
<i>Response Header Fields</i>	97
<i>Entity Header Fields</i>	97
APPENDIX D: THE CUSTOM “TLCP-DIFF” FORMAT.....	98

1 Introduction

The Text Lightstreamer Client Protocol, or TLCP for short, is the main protocol used by Lightstreamer clients to connect, receive and send data to a Lightstreamer Server.

This document specifies version **2.5.0** of the protocol.

NOTE: To fully understand how TLCP works, a general understanding of the main Lightstreamer concepts is a prerequisite. Terms like Adapter Set, Metadata Adapter, Data Adapter should be familiar. Knowledge of these components may be obtained by reading the first chapters of the **General Concepts** document.

Protocol Versions

The TLCP protocol can evolve over time and it is versioned in the form:

👉 **major.minor.subminor**

Example: 2.1.0

The general rule is that if only the subminor version number changes, no upgrades to the Lightstreamer Server are required to support the new version of TLCP.

Example:

- 👉 Lightstreamer Server v. 6.1 is released, with support for TLCP v. 2.0.0.
- 👉 Later on, Lightstreamer Server v. 6.2 is released, which, among other things, introduces explicit support for TLCP v. 2.0.3.
- 👉 A client is developed against TLCP v. 2.0.3. Such client is guaranteed to work even if it connects to Lightstreamer Server v. 6.1, because that server supports TLCP v. 2.0.x.

On the other hand, any new version of Lightstreamer Server is guaranteed to work with any previous version of TLCP.

Main Concepts

Some of the main concepts of TLCP are the following:

- 👉 Communication between client and Server always operates inside the context of a **session**. Each session has a specific **session ID**.
- 👉 A session must be bound to a network connection in order to receive downstream **real-time notifications** and in particular **real-time updates**. The connection to which a session is bound is called the **stream connection** of that session.
- 👉 The **session creation request** provides both a session and its initial stream connection. The session may be later bound to a different stream connection with a **session binding request**.
- 👉 Real-time updates are organized into **items** and **fields**. The set of items and fields for which real-time updates are received can be changed by means of session **control requests**.

- 👉 Control requests, depending on the **transport**, may be sent on **separate network connections** or on the session **stream connection**.
- 👉 The control request that adds new items to the set being received is called a **subscription**. The opposite control request, that removes items and fields from the set, is called an **unsubscription**.
- 👉 A specific subset of control requests is dedicated to sending upstream **messages**.

Transports

TLCP supports two different transports:

- 👉 HTTP (and HTTPS).
- 👉 WebSocket, or WS for short (and Secure WebSocket, or WSS for short).

Edition Note: WSS/HTTPS is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

The protocol is designed to minimize differences on the two transports, while still leveraging transport-specific advantages, e.g. with HTTP the stream connection is implemented as an HTTP request whose response is of (almost) **infinite length**. On the other hand, since HTTP is a **half-duplex** transport, each control request requires a separate connection, parallel to the main stream connection.

With WS, which is a **full-duplex** transport, the stream connection is a simple WS connection and control requests may be sent directly on it, eliminating the connection overhead of HTTP.

Transports may also be **mixed**: as far as the correct session ID is specified in each control request, a stream connection on WS may be controlled by control requests on HTTP and vice-versa. Mixing of transports is actively exploited by official Lightstreamer client libraries to implement the Stream-Sense algorithm, which automatically discovers the best transport to use at any given moment.

Session Life Cycle

The session may be in one of 2 states:

- 👉 **Bound** to a stream connection;
- 👉 **Unbound** from a stream connection.

When the session is bound, real-time notifications are sent downstream to the stream connection for the client to consume them. When the session is unbound, they are **buffered** to be sent later.

A session may become unbound for a number of reasons, e.g.:

- 👉 A stream connection is dropped by any cause.
- 👉 The client may explicitly request it because it is switching to another transport (e.g. from HTTP to WS).

- ▶ The transport may reach some limit (e.g. an HTTP 1.0 response has reached its content-length).

Once the session is unbound, the client may rebind it to another stream connection with a session binding request. When this happens, the session **restarts** sending real-time notifications from where it stopped. In particular, all subscriptions are preserved, with all their items and fields.

As said, unbounding may also be caused by a **connection drop**, due to either a manual close of the connection by the client or to a network problem. In the former case, the Client is willing to abandon the session; but in the latter case, discarding the session is undesired. Since the two cases cannot be distinguished, upon connection drop, the session is not closed, hence the Client is still allowed to issue a rebind. However, some notifications may have been lost with the failed connection and must be resumed by the new one. The details of this **session recovery** mechanism are provided below. The session recovery attempt may fail. In this case, the Client can't but create a **new stream connection** as if it was the first time it connects.

A special use case of session rebinding is a communication mode called **long polling**, which official Lightstreamer client libraries adopt with HTTP transport when a network appliance in the middle is caching the response and blocking real-time notifications.

Long polling works as follows:

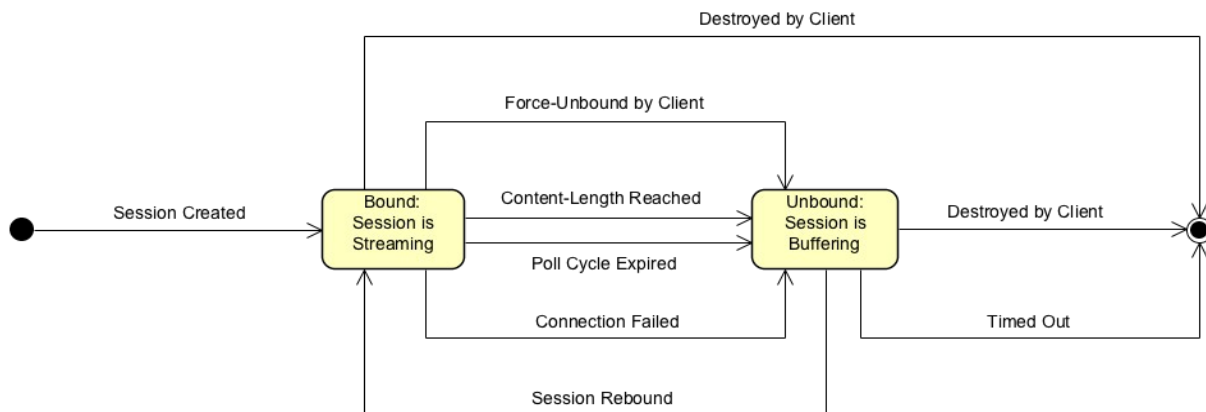
1. An initial stream connection is created with a **limited life-time** (e.g. a few seconds).
2. During this life-time, if real-time notifications are available they are sent and the session is unbound immediately after that, **closing the stream connection**. In this way, caching appliances in the middle **flush their cache** and notifications can be received with minimum latency.
3. If no real-time notifications are available, when this life-time expires the Server unbinds the sessions anyway and closes the stream connection.
4. A new stream connection is immediately re-established by the client with a **binding** request, and the loop restarts from point 2.

While it applies particularly well to HTTP cases, this communication mode is transport independent and may also be applied to WS under particular circumstances, such as to forcibly slow down a connection with a client that can't keep up on high frequency updates.

Note that a session may never be bound to more than one stream connection. Moreover, when unbound, buffering lasts only for a limited (although configurable) amount of time. After that, the **session is discarded** and buffered events (and their originating subscriptions) are lost.

When the Client wants to abandon a session, it can simply **drop the connection** (if currently bound) and refrain from binding again. In order to prevent the Server from keeping the session unbound for some time, it can also request the immediate destroy of the session.

The diagram below shows the set of session states and possible transitions:



Session Recovery

In case of an unexpected interruption of a stream connection, the Client can try to rebind to the same session and resume the notification flow from the exact point it was interrupted. This can be achieved provided that:

1. The count of all the notifications received since the start of the session has been kept. More precisely, the notifications to be counted are the ones related with the subscription and message activity (as reported below in [Real-Time Update](#), [Other Subscription-Related Notifications](#), and [Message-Related Notifications](#)). We will call them “**data notifications**”.
2. A special binding request is issued, in which the count of all the received data notifications is specified as the requested starting point.
3. The Server, upon reception of the binding request, has kept the session alive and has stored all data notifications already sent since the point requested. The amount of sent data stored and the time it is kept depend on Server configuration.
If this is available, the stream connection will start successfully and, before sending any data notification, the Server will specify, through a proper notification, the modified starting point, which, however, may also be lower (i.e. earlier) than requested.
4. If the starting point of the received flow is lower than requested, the first data notifications received are discarded, until the desired point is reached.

The described mechanism only involves the data notifications; any other notifications received must always be treated as usual: they are not related with data, but only with the session and connection lifecycle and are also not meant to be resumed.

Note that an interruption of the connection on the Server side is not needed to perform a recovery action. For instance, if a connection becomes mute, the client can issue a recovery request, even though, perhaps, the notification flow is just being delayed somewhere. However, once issued, the recovery request invalidates any further notifications (of any type) that may be later received on the original connection.

Note that, upon the successful response to a recovery request, the streaming is resumed normally. For instance, the stream connection carrying recovery data may terminate and ask for a further rebind before all notifications to be recovered are sent; in this case, a normal rebind will be enough to furtherly resume the notification flow.

Subscription Data Model

As introduced before, a **subscription** is the operation with which the client asks the Server to send real-time updates about some items and fields. An **unsubscribe** is the opposite operation: to ask the Server to stop sending real-time updates about previously subscribed items and fields.

A subscription request requires the following information:

- A client-generated **subscription ID**. It must be a progressive integer number starting with 1, and must be unique within the session.
- A **group name**, i.e. a single variable-length symbolic name that represents the **set of items** it wants to subscribe to:
 - ◆ The mapping between the group name and the corresponding item set is implemented in the **Metadata Adapter**.
 - ◆ A typical example of group is the name of a stock list, e.g. "NASDAQ-100", which the Metadata Adapter maps to the list of 100 items composing the NASDAQ 100 list.
 - ◆ Nothing forbids to use a second-level syntax to express a literal list of items, e.g. "AAPL MSFT GOOGL". It's up to the Metadata Adapter to interpret the group name.
 - ◆ Note that the same item may be included in multiple groups.
- A **schema name**, i.e. a single variable-length symbolic name that represents the **set of fields** it wants to subscribe to:
 - ◆ As with the group, the mapping between the schema name and the corresponding field set is implemented in the **Metadata Adapter**.
 - ◆ As with the group, nothing forbids to use a second-level syntax to express a literal list of fields, e.g. "NAME LAST OPEN MIN MAX". It's up to the Metadata Adapter to interpret the group name.
- A **subscription mode** that tells the Server how to consider real-time updates for this set of items:
 - ◆ With MERGE mode, the subscription represents a **fixed table** where **items are rows** and **fields are columns**. Each real-time update **changes the content of a specific row**.
 - ▶ A common example is a stock quote table.
 - ◆ With COMMAND mode, the subscription represents a **dynamic table**, where **items are rows** and **fields are columns**. Each real-time update may **add a new row**, **delete an existing row** or **change the content of a specific row**.
 - ▶ A common example is a stock portfolio.
 - ◆ With DISTINCT mode, the subscription represents a **set of growing tables**, where **each item is a separate table** and **fields are columns**. Each real-time update **appends a new row** at the end of a specific table.
 - ▶ A common example is a news feed, with a different news topic for each item.
 - ◆ For an in-depth discussion about subscription modes see the *General Concepts* document.
- Optionally, the name of the **Data Adapter** that implements the subscribed group.

Once a subscription is active, **real-time updates** include the following information:

- The **subscription ID**.

- 🚩 The **progressive number of the item** updated. Item numbers are progressive integers starting with 1.
 - ◆ The order of items of a certain group is defined by the **Metadata Adapter**.
 - ◆ The same item may appear in different groups with a different number. The client is supposed to know *a priori* this ordering.
- 🚩 A list with **values of updated fields**.
 - ◆ The order of fields of a certain schema is defined by the **Metadata Adapter**.

Finally, an **unsubscribe** request requires just the following information:

- 🚩 The **subscription ID**.

Requests, Responses, and Notifications

General rules apply on request and response handling:

- 🚩 Control requests are always identified by a client-generated **request ID**. It may be any combination of letters and numbers, but typically a progressive number will do, and must be unique within the connection.
 - ◆ It needs to be unique only within a reasonable amount of time, e.g. a few minutes. The Server will never respond to a request with a delay of several minutes. If you use a progressive integer, you can periodically reset the counter to avoid sending long integer strings.
- 🚩 Responses (including errors) are always identified by the **corresponding** request ID, unless the request syntax was so misleading that a request ID could not be parsed.
- 🚩 Responses (including errors) are always sent on the **same connection** the request was sent to:
 - ◆ If the control request was sent on a separate connection, such as with HTTP transport, the response is sent on that same connection.
 - ◆ If the control request was sent on the stream connection, such as with WS transport, the response is sent on the stream connection.
- 🚩 The requests are handled in parallel and independently of one another, also when the transport supports a sequence of requests (as in the WebSocket case). As a consequence, the responses to multiple requests can come in any order.

For instance, if an unsubscribe request immediately follows the related subscription request, it is possible that the unsubscribe request will be processed first. In this case, it will fail and only the subscription will take place. The failed unsubscribe request will be notified with a proper error response.

There are only a few exceptions to the parallel processing of subsequent requests, that will be specified within the request descriptions.
- 🚩 If the request was successful, a **notification** may be sent on the **stream connection** (if appropriate for the request).
- 🚩 Requests so wrong as to break the protocol (or involved in severe processing issues) may cause the interruption of the connection they were sent to. Such interruption may either be abrupt or provide an error code leveraging the transport syntax (e.g. HTTP Response Status Code or WebSocket Close Status Code). However, the interruption details are not in the scope of the TLCP protocol.

Request Syntax

Requests follow general rules for query string composition in an HTTP URL, with minor differences for the HTTP and WS cases.

HTTP Transport

General form is as follows:

```
POST /lightstreamer/<request-name>.txt?LS_protocol=TLCP-2.5.0 <HTTP-version>
<HTTP-request-headers>
<empty-line>
<param1>=<value1>&...&<paramN>=<valueN>
```

Where:

- 👉 **<request-name>** determines the kind of request; e.g. “create_session” for session creation request, “control” for control requests, etc.
- 👉 **LS_protocol=TLCP-2.5.0** is a required parameter to specify the request protocol.
- 👉 **<param>=<value>** pairs set the **parameters** of the request; e.g. “LS_reqId=123” to specify the request ID, or “LS_session=S1aa6c792585db57aT1726545” to specify the session ID.
- 👉 **<HTTP-version>**, **<HTTP-request-headers>** and **<empty-line>** are part of the HTTP 1.x request specifications as usual. See [Appendix C: HTTP Request Headers](#) for more information on accepted HTTP headers.
- 👉 The line separator is **CR-LF**, as per HTTP 1.x specifications.

In the specific case of **control requests**, multiple requests with the same request name may be sent in a single **batch** with the following syntax:

```
POST /lightstreamer/<req-name>.txt?LS_protocol=TLCP-2.5.0[&LS_session=<session-ID>] <HTTP-version>
<HTTP-request-headers>
<empty-line>
<param1>=<value1>&...&<paramN>=<valueN>
...
<param1>=<value1>&...&<paramN>=<valueN>
```

Where:

- 👉 **<LS_session>=<session-ID>** pair in the **query string**, if present, acts as a **default value** for subsequent requests specified in the body; e.g. “LS_session=S1aa6c792585db57aT1726545” if all requests act on session ID S1aa6c792585db57aT1726545.
- 👉 Each **<param1>=<value1>&...&<paramN>=<valueN>** line in the request body specifies a **separate control request**.
- 👉 The line separator is **CR-LF**, also within the request body. The separator is optional after the last line, unless the line is empty (which is possible, in principle).

According to the general policy, also requests of the same batch are executed concurrently, and responses may arrive out of order.

The request body can be expressed in any charset supported by the Java installation which runs the Server, but we recommend **UTF-8**, which is guaranteed to be supported.

Reserved characters in the values of the parameters specified in the body must be **percent-encoded**. Such reserved characters are limited to:

- ➔ Ascii CR and LF (used for line separation);
- ➔ & and = (used for parameter separation);
- ➔ % and + (used for percent-encoding / URL Encoding).

However, any further percent-encoding is supported, provided that it is based on the UTF-8 character set (regardless of the charset used for the request body). This allows you to take advantage of existing encoding libraries which include the handling of the above characters, although this will introduce inefficiencies.

On the other hand, the parameters on the request querystring don't make use of any reserved character. However, should a user-agent perform any extra percent-encoding on the request line, it would be supported as well.

Obviously, the response will be received as the content of the HTTP response **body**.

WS Transport

Open a WebSocket to the Lightstreamer Server, using the URI and subprotocol below:

➤ URI: **/lightstreamer**

➤ Sec-WebSocket-Protocol: **TLCP-2.5.0.lightstreamer.com**

Once the WS connection has been established, the general form of a request is as follows:

```
<request-name>
<param1>=<value1>&...&<paramN>=<valueN>
```

Where:

- **<request-name>** determines the kind of request; e.g. "create_session" for session creation request, "control" for control requests, etc.
- **<param>=<value>** pairs set the **parameters** of the request; e.g. "LS_reqId=123" to specify the request ID, or "LS_session=S1aa6c792585db57aT1726545" to specify the session ID.
- As in the HTTP case, the line separator is **CR-LF** (note that name and parameters are on **separate lines**). The separator is optional after the last line, unless the line is empty (which is possible, in principle).

Each request must be sent in a single WS message. Note that with WS the name extension (e.g. ".txt") **is missing**.

With WS, like with HTTP, multiple control requests can be sent in a single batch with the following syntax:

```
<request-name>
<param1>=<value1>&...&<paramN>=<valueN>
...
<param1>=<value1>&...&<paramN>=<valueN>
```

Anyway, consider that since WS is a full-duplex transport, the advantage of sending a batch of requests is not as evident as in the HTTP case, where the overhead of multiple HTTP connections is greatly reduced.

According to the general policy, also requests of the same batch are executed concurrently, and responses may arrive out of order. Moreover, with WS, responses may always arrive **interspersed** with notifications, when control requests are sent on the stream connection.

As long as a session is bound to a WS, no other session can be bound. After unbinding of a session, the same or a different session can be bound.

Note that, after binding the second session, it is possible that late responses to control requests related with the first session arrive interspersed with notifications for the second session.

The WS messages to be used to carry requests must be of **text** (as opposed to binary) type, hence they must be based on the **UTF-8** character set.

Reserved characters in the values of the parameters specified in the message must be **percent-encoded**. Such reserved characters are limited to:

- ➔ Ascii CR and LF (used for line separation);
- ➔ & and = (used for parameter separation);
- ➔ % and + (used for percent-encoding / URL Encoding).

However, any further percent-encoding is supported, provided that it is also based on the UTF-8 character set. This allows you to take advantage of existing encoding libraries which include the handling of the above characters, although this will introduce inefficiencies.

The response will be received as a WS text message, or a **sequence** of them in case of a session creation / binding request.

Common Response and Notification Syntax

All of requests use a simple syntax for their response, either failed or successful. The same base syntax is also adopted by notifications on the stream connection. This syntax is as follows:

```
<tag>,<argument1>,...,<argumentN>
```

Where:

- 👉 **<tag>** is a short (up to 8 characters) uppercase string identifying the **kind** of response or notification.
- 👉 **<argument1>,...,<argumentN>** are the arguments of the specific response or notification.

If an argument contains meta characters, such as the comma, or some other special characters, they are **percent-encoded**, with the notable exception of the last argument of a real-time update (see the parsing algorithm below). The character set is **UTF-8**, for both encoded and unencoded content. The line terminator is **CR-LF**.

Note that an argument may be empty, although most arguments, because of their meaning in the context of the response, will never be empty.

To simplify parsing, each different tag has a **fixed number of arguments**. Once you have extracted the tag by finding the first comma, you know how many other commas are present, and thus how to extract the following arguments.

Some examples of tags and arguments are:

```
CONOK,<session-ID>,<request-limit>,<keep-alive>,<control-link>
REQERR,<request-ID>,<error-code>,<error-message>
SUBOK,<subscription-ID>,<num-items>,<num-fields>
```

```
MSGFAIL, <sequence>, <prog>, <error-code>, <error-message>

END, <cause-code>, <cause-message>
```

A simple parsing algorithm is the following:

- 👉 Read the HTTP response or the WS message **line by line** (line terminator is **CR-LF**).
- 👉 For each line, trim the trailing line terminator and look for the first **comma** from left to right (it may not be there).
- 👉 The text until the first comma (or the full line, if it's not there) is the **tag**.
- 👉 **Switch** on the **tag**:*
 - ◆ Each tag has a fixed **number of arguments** (e.g. for a "REQERR" tag it's 3).
 - ◆ **Split** the remaining part of the string **comma by comma**, from left to right, to obtain each argument. Note that the last argument may contain **additional commas**.
 - ◆ **Percent-decode** non-numeric arguments, **except** for the last argument of a real-time update (tag "U").**
- 👉 Interpret the response or notification accordingly.

Notes:

[*] The **first 4 characters** of a tag are **unique**. If parsing with a programming language that does not support switching on string literals, the first four characters of a tag (which are all simple Ascii characters) may be easily mapped to a 32 bit integer and then switch on its corresponding numeral.

E.g. in C language:

```
#define STR_SWITCH(a,b,c,d) (((a & 0xff) << 24) | \
                           ((b & 0xff) << 16) | \
                           ((c & 0xff) << 8)  | \
                           (d & 0xff))

char tag[8] = /* Zero-padded tag read from network */;
unsigned int iTag= STR_SWITCH(tag[0], tag[1], tag[2], tag[3]);
switch (iTag) {
case STR_SWITCH('R','E','Q','O'):
    // Handle "REQOK"
    break;

case STR_SWITCH('R','E','Q','E'):
    // Handle "REQERR"
    break;

// ...
}
```

[**] The last argument of a **real-time update** is itself a variable-length set of fields, with its own **second-level syntax**. To avoid unnecessary encoding of frequently used characters, this argument follows specific encoding rules and should not be percent-decoded during the first pass.

For more details on its syntax, and a second-level parsing algorithm, see [Real-Time Update](#).

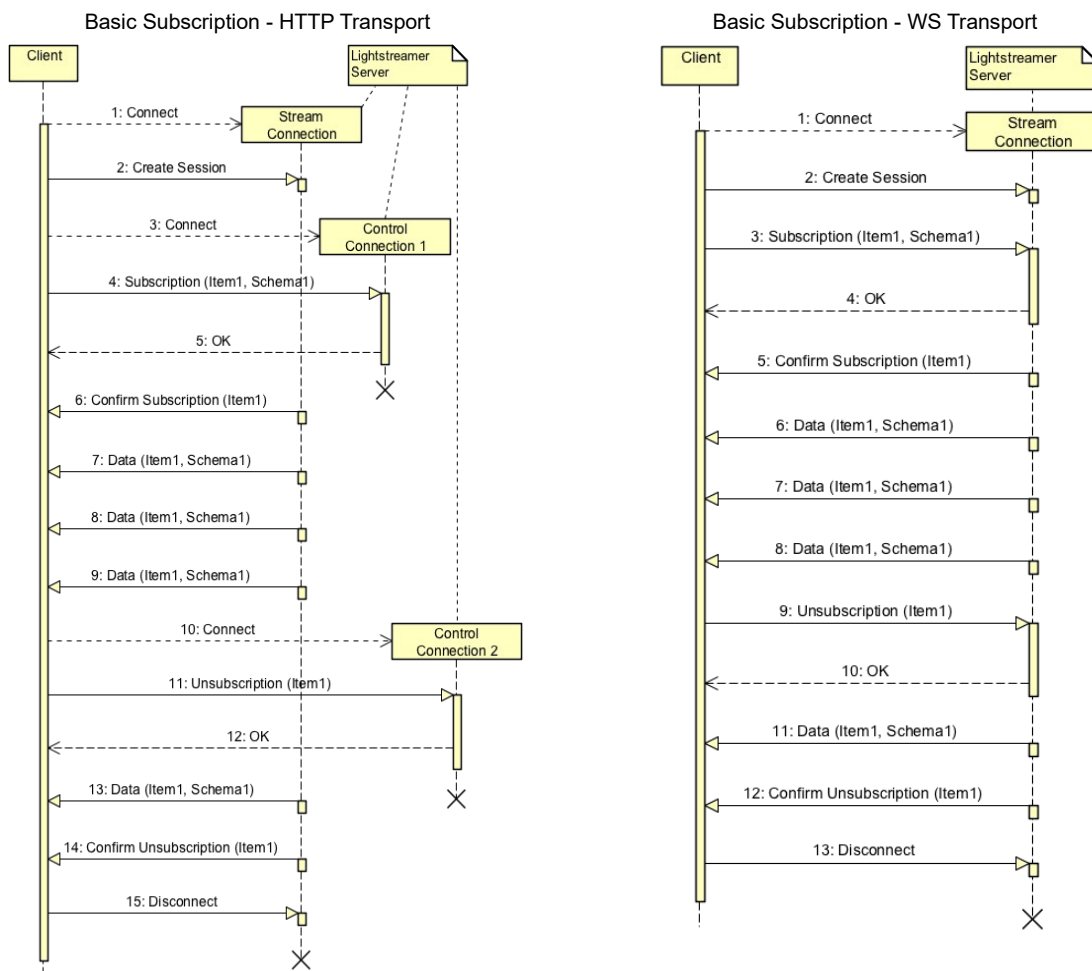
Workflow Examples

The following diagrams illustrate a few notable examples of TLCP workflow. Diagrams are presented for both HTTP and WS transports.

Basic Subscription

This workflow shows the simplest *client-server interaction*:

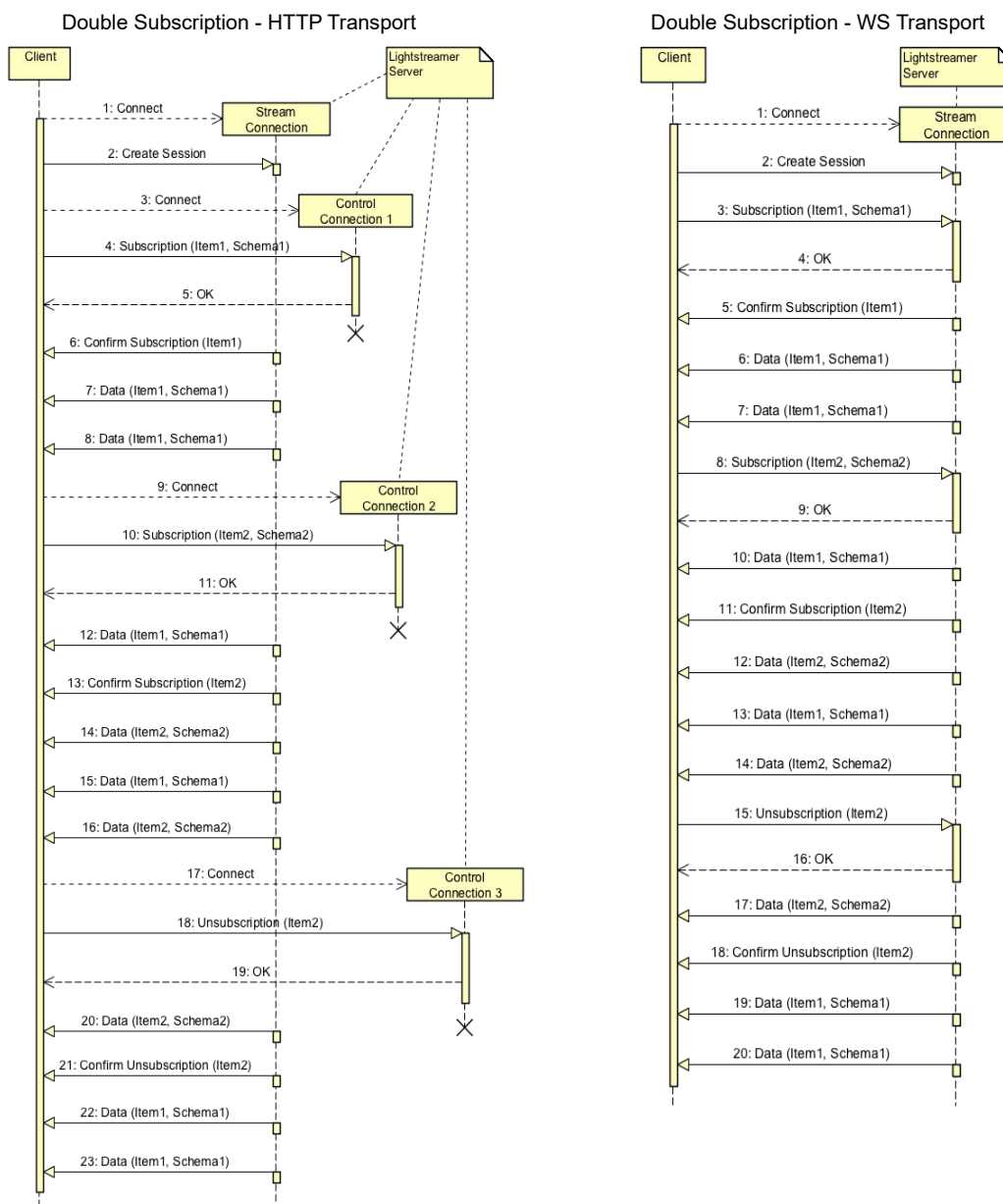
- 👉 The client **opens** the stream connection with the Server, which accepts the connection.
- 👉 Then, the client **subscribes** to the item *item1* (with the related field schema *schema1*) and the Server starts sending real-time updates to the client.
- 👉 The client **unsubscribes** from the item *item1*, the Server stops sending real-time updates. Note that real-time updates may follow the unsubscription request response, but not the unsubscription notification.
- 👉 Finally the client **closes** the stream connection.



Double Subscription

The following workflow shows a more complex example, that is:

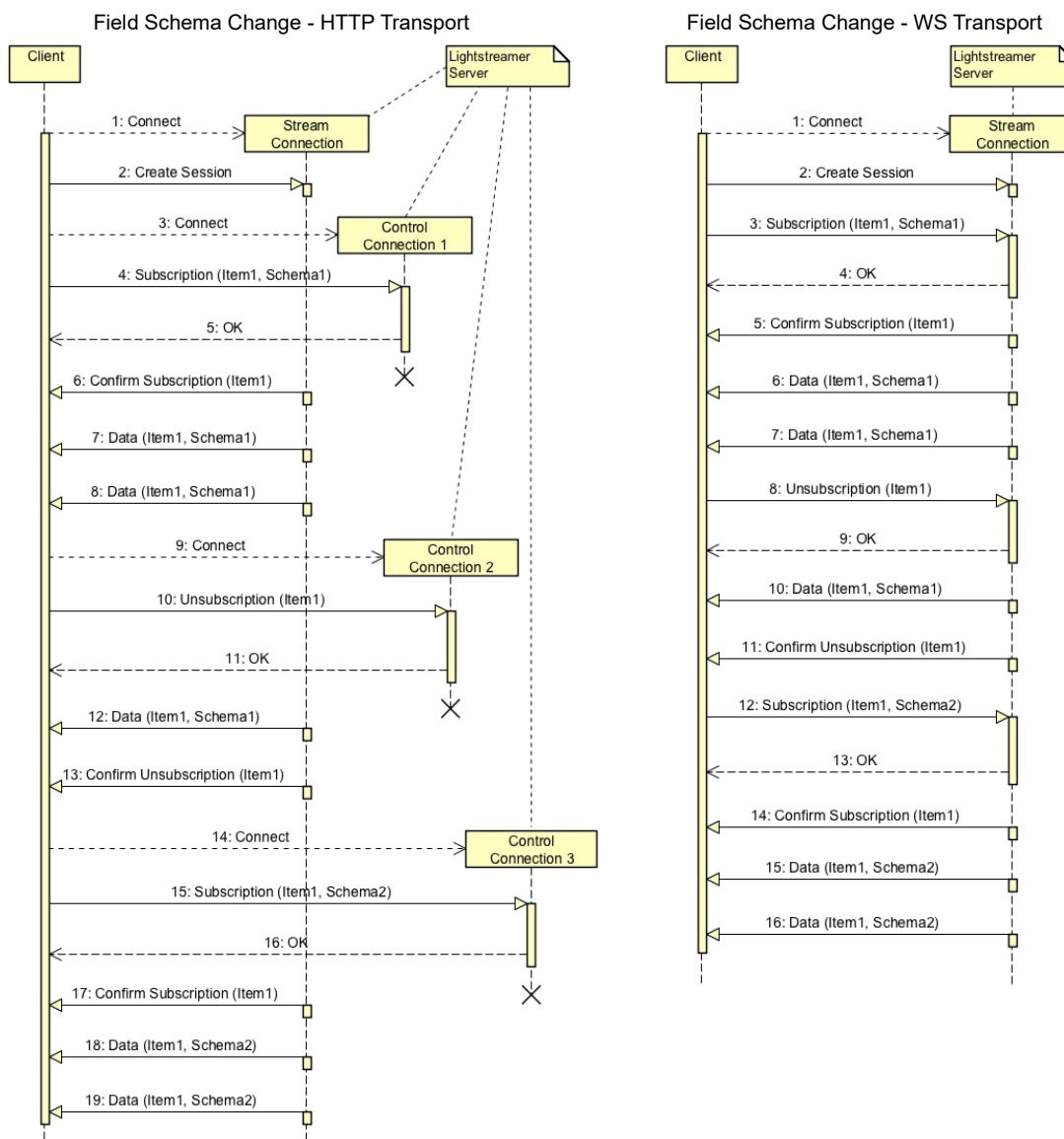
- The client opens the **stream connection** with the Server.
- The client **subscribes** to *item1* (with the related field schema *schema1*) and the Server starts sending real-time updates for *item1*.
- The client **subscribes** to *item2* (with the related field schema *schema2*) and the Server starts sending real-time updates for *item2*.
- The client **unsubscribes** from *item2*, the Server stops sending real-time updates for *item2* but continues sending them for *item1*.



Field Schema Change

In this workflow, the client decides to change the **set of fields** for which it is receiving real-time updates:

- The client opens the **stream connection** with the Server.
- Then, the client **subscribes** to *item1* (with the related field schema *schema1*) and the Server starts sending real-time updates for *item1*.
- The client decides to change the subscription field schema, so it **unsubscribes** from *item1* (and the Server stops sending real-time updates for *item1*) and then **re-subscribes** to *item1* with the new field schema *schema2* (and the Server re-starts sending real-time updates for *item1*, but with **different fields**).



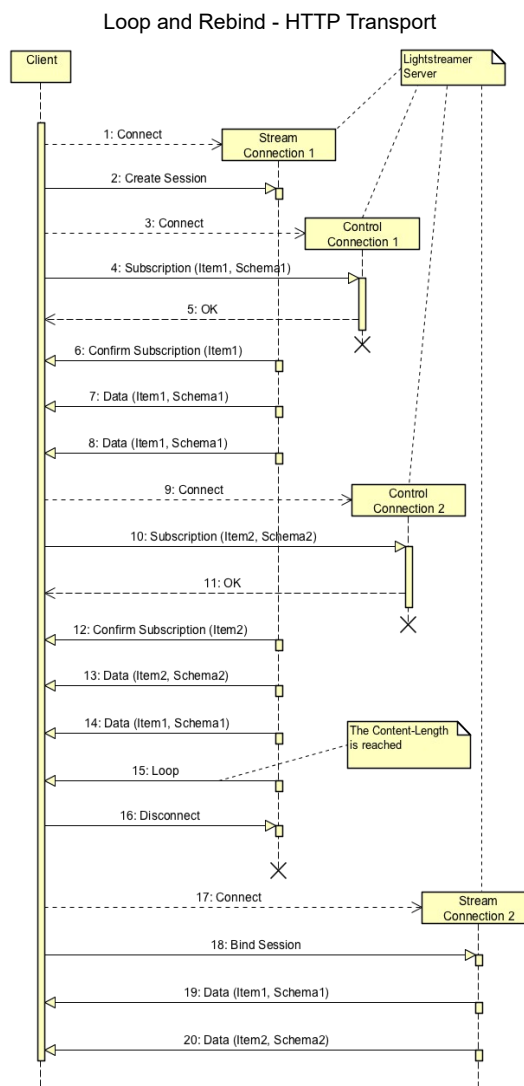
Session Rebinding

The following workflows show a few situations where a session needs to be rebound.

HTTP Loop and Rebind

This example shows a rebind as a reaction to reaching the *Content-Length* of a stream connection that was requested in HTTP 1.0:

- 👉 The client opens the **stream connection** with the Server.
- 👉 Then, the client **subscribes** to *item1* and *item2* (with their related field schemas). The Server starts sending real-time updates for *item1* and *item2*.
- 👉 When the **Content-Length is reached**, the server sends the specific command **Loop** and closes the connection. The client rebinds the session to a new stream connection and real-time updates restart.

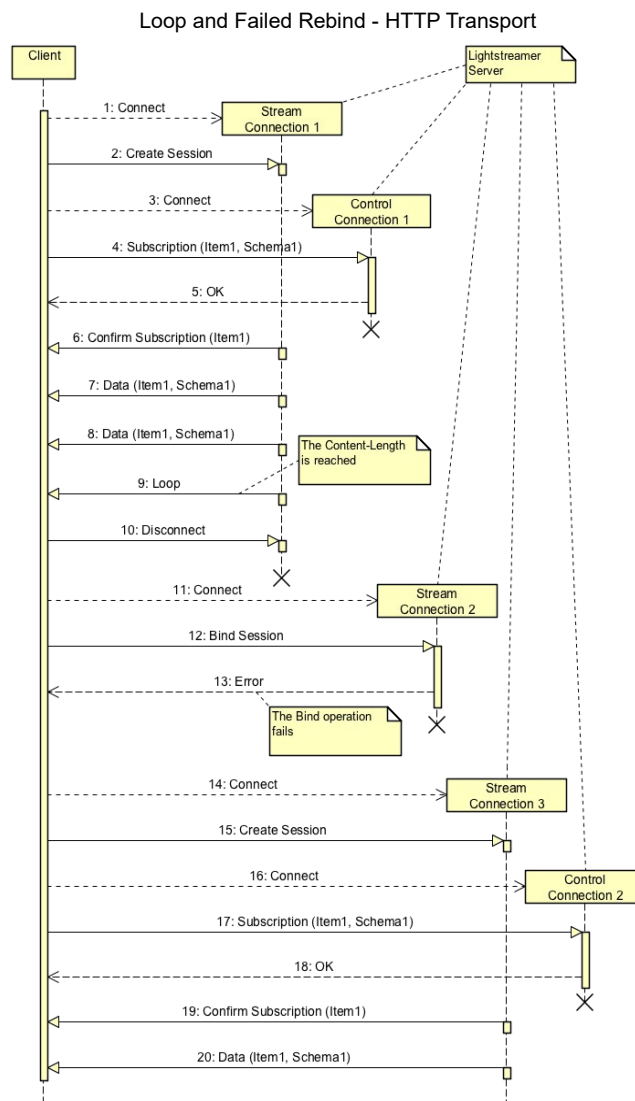


HTTP Loop and Failed Rebind

This case shows how to react to a rebind failure. In fact, The rebinding operation may fail for a number of reasons, e.g.:

- 👉 If **too much time** passes since the end of the previous stream connection, the Server **deletes** the old session.
- 👉 If the Server has been shut down and a brand new instance is now responding on the same port.
- 👉 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** Server that will not recognize the old session (see the **Clustering** document on how to avoid it).

In this situation, the client should create a **new stream connection**, as if it was the first time it connects to the Server, and re-execute all the **subscriptions** that were active at the moment the previous stream connection terminated.



By the way, in the last error case described above (i.e. a cluster of Servers with incorrect affinity), the issue is permanent and will not be overcome by a reconnection but only by fixing the configuration. The Server tries to recognize this case and use a different error code.

WS Long Polling

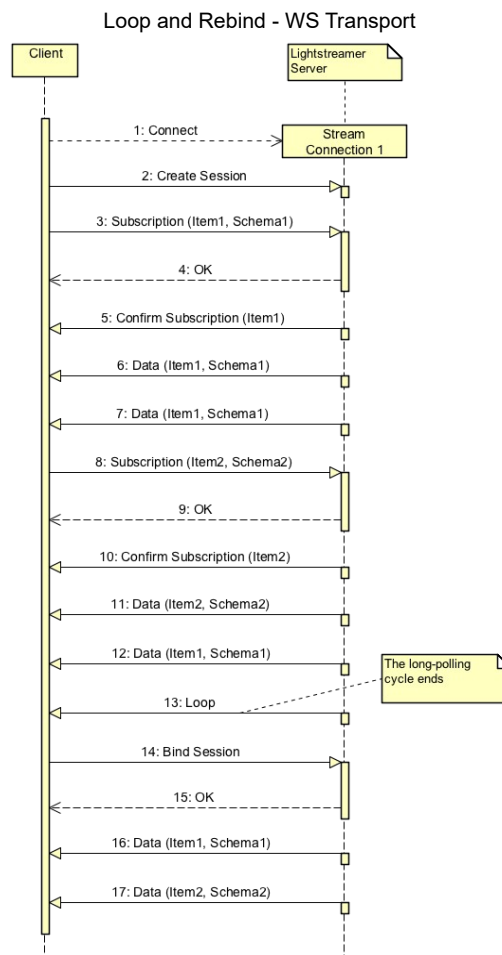
When a WS stream connection needs to be rebound, the server sends the **Loop** command exactly as in the HTTP case, but it **does not close the connection**. The client is free to choose whether to rebind the session to the same stream connection or open a new one.

This difference with HTTP is due to the different use cases where rebinding is applied:

- 👉 With HTTP rebinding is usually due to an exhausted or malfunctioning stream connection. In both case keeping the connection open wouldn't help.
- 👉 With WS rebinding is usually unnecessary. As described in Chapter 1, official Lightstreamer client libraries use it as an expedient to slow down real-time updates in a client that can't keep up with the Server.

In this workflow:

- 👉 The client opens the **stream connection** with the Server and **subscribes** to *item1* and *item2*. The Server starts sending real-time updates for *item1* and *item2*.
- 👉 When the **poll cycle ends**, the server sends the **Loop** command and unbinds the session. The client then rebinds the session to the **same** stream connection and real-time updates restart.



2 Request/Response Reference

This chapter details the complete syntax reference for all TLCP requests and responses. The general syntax is discussed in [Request Syntax](#), together with a sample algorithm for response parsing. The syntax is reported here for your convenience:

👉 Requests general syntax for HTTP transport:

```
POST /lightstreamer/<req-name>.txt?LS_protocol=TLCP-2.5.0[&LS_session=<session-ID>] <HTTP-version>
<HTTP-request-headers>
<empty-line>
<param1>=<value1>&...&<paramN>=<valueN>
...
<param1>=<value1>&...&<paramN>=<valueN>
```

👉 Requests general syntax for WS transport:

```
<req-name>
<param1>=<value1>&...&<paramN>=<valueN>
...
<param1>=<value1>&...&<paramN>=<valueN>
```

Reserved characters (as per section 2.2 of RFC 3896) in the value of a parameter must be **percent-encoded**.

👉 Responses and notifications general syntax (with HTTP transport the response is in the body of the HTTP response):

```
<tag>,<argument1>,...,<argumentN>
```

Line separator is **CR-LF**. Encoding is **UTF-8**.

Please note that in the above syntax examples and in all the examples that follow we don't make the **CR-LF** parts explicit and just use separate text lines. As a consequence, forging sample requests by starting from a copy-and-paste from our examples may not yield correct requests, depending on your editor end-of-line settings. Please always ensure the presence of the full **CR-LF** instead of only **CR** or only **LF**.

Session Creation Request

A **session creation request** provides the means to obtain an initial **session ID** and an initial **stream connection** to receive real-time notifications.

It is defined as follows:

👉 Request name: `create_session`

👉 Parameters:

- ◆ `LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg`

Client identifier. Must be set with the special string `mgQkwtwdysogQz2BJ4Ji%20kOj2Bg` for all custom developed clients.

- ◆ `LS_user`

Optional. User name (used for authentication). This string should be interpreted and verified by the Metadata Adapter, so the developer is free to decide its 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 its meaning.
- ◆ `LS_adapter_set`
Optional. 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. If omitted, an adapter set named “DEFAULT” is assumed.
- ◆ `LS_requested_max_bandwidth`
Optional. Maximum bandwidth requested by the client, expressed in kbps (it can be a decimal number, with a dot as decimal separator). See the *General Concepts* document for more information.
Edition Note: Bandwidth Control is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).
- ◆ `LS_content_length`
Optional. Content-Length to be used for the connection content, expressed in bytes. If too low or not present, the Content-Length is assigned by Lightstreamer Server, based on its own configuration.
- ◆ `LS_supported_diffs`
Optional. Text “diff” formats accepted for the indication of update values. In fact, the protocol allows the Server to choose among a few “diff” algorithms to express new values as differences from previous values (see [Real-Time Update](#)), but the set can be restricted through this parameter. The value should be a (possibly empty) comma-separated list of format tags, as specified in [Real-Time Update](#).
- ◆ `LS_polling`
Optional. Requests a **polling connection**. If set to `true`, the Server will send only the notifications 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**, expressed in milliseconds. 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 in the response.
- ◆ `LS_idle_millis`
Only if `LS_polling` is `true`. **Optional.** Time the Server is allowed to **wait for a notification**, if none is present at request time, expressed in milliseconds. 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.
- ◆ `LS_inactivity_millis`
Only if `LS_polling` is not `true`. **Optional.** Maximum time the Client is committed to wait before issuing a request to the Server while this connection is open. If no request is needed, a `heartbeat` pseudo-request can be sent. If no request is received for more than this time, the Server can assume that the client is stuck and act accordingly.
- ◆ `LS_keeplive_millis`
Only if `LS_polling` is not `true`. **Optional. Longest inactivity time** allowed for the connection, expressed in milliseconds. If such a long inactivity occurs, the Server sends a keeplive message (i.e. a `PROBE` notification). 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 the Server, based on its own configuration. Anyway, the keepalive time used is notified to the client in the response.

◆ **LS_send_sync**

Only if LS_polling is not true. Optional. If set to `false`, instructs the Server not to send the `SYNC` notifications on this connection. If omitted, the default is `true`.

◆ **LS_reduce_head**

Optional. If set to `true`, instructs the Server not to send the `SERVNAME` and `CLIENTIP` notifications on this connection. Moreover, this instructs the Server not to send the `CONS` notifications throughout all the session. If omitted, the default is `false`.

◆ **LS_ttl_millis**

Optional. Maximum time the client is committed to wait for an answer to this request before giving up, expressed in milliseconds. This information allows the Server to interrupt the processing of the request and discard it if it detects that the specified time limit cannot be obeyed.

Notes:

- ▶ The special value `unknown` is also supported. In this case, the request is kept until completion. This is also the default behavior.
- ▶ A request can also be discarded before completion when the connection is closed.
- ▶ A request can also be discarded before completion (with error code 5) when the Server is overloaded and it determines that a significant delay is still going to be accumulated.
- ▶ The special value `unlimited` is also supported. It is similar to `unknown`, but, in this case, the request is kept also if the Server is overloaded.

Request example with **HTTP** transport:

```
POST /lightstreamer/create_session.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 65
Content-Type: text/plain

LS_user=user&LS_password=password&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg
```

Request example with **WS** transport:

```
create_session
LS_user=user&LS_password=password&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg
```

See [Session Creation and Binding Responses](#) section in this chapter for sample responses.

Session Binding Request

A **session binding request** provides the means to rebind an existing **session ID** to a **new stream connection** and restart the flow of real-time updates and notifications.

It is defined as follows:

👉 **Request name:** `bind_session`

Parameters:

◆ LS_session

Optional with WS transport. The Server internal string representing the **session** the client wants to **bind** to.

Notes:

- ▶ When omitted with WS transport, the rebind will apply to the last session that was bound to the WS connection. In other words, the request can be made to rebind on the same stream connection the session was previously bound to. Hence it is required that a preceding session creation or binding request took place and was fulfilled successfully.
- ▶ It is **always required** with HTTP transport.

◆ LS_recovery_from

Optional. Progressive number of the last **data notification** received in the session (or, equivalently, total number of data notifications received from the beginning of the session), to be used to determine the starting point for the response flow. This is related with the Session Recovery mechanism detailed above.

If not present, the Server will assume that all updates sent in the previous connection (up to the `LOOP` notification) have been received by the client and will restart from the next update.

On the other hand, by specifying a number, the client can recover from an interruption of the previous stream connection, provided that the Server is still keeping the missed notifications. This will also cause a `PROG` notification to be issued on the response before any **data notification**.

◆ LS_content_length

Optional. Content-Length to be used for the connection content. If too low or not present, the Content-Length is assigned by Lightstreamer Server, based on its own configuration.

◆ LS_polling

Optional. Requests a **polling connection**. If set to `true`, the Server will send only the notifications 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 in the response.

◆ LS_idle_millis

Only if LS_polling is true. Optional. Time the Server is allowed to **wait for a notification**, 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.

◆ LS_inactivity_millis

Only if LS_polling is not true. Optional. Maximum time the Client is committed to wait before issuing a request to the Server while this connection is open. If no request is needed, a `heartbeat` pseudo-request can be sent. If no request is received for more than this time, the Server can assume that the client is stuck and act accordingly.

◆ LS_heartbeat_millis

Only if `LS_polling` is not `true`. Optional. Longest inactivity time allowed for the connection. If such a long inactivity occurs, the Server sends a keepalive message (i.e. a `PROBE` notification). 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 the Server, based on its own configuration. Anyway, the keepalive time used is notified to the client in the response.

◆ **`LS_send_sync`**

Only if `LS_polling` is not `true`. Optional. If set to `false`, instructs the Server not to send the `SYNC` notifications on this connection. If omitted, the default is `true`.

◆ **`LS_reduce_head`**

Optional. If set to `true`, instructs the Server not to send the `CONOK`, `SERVNAME`, and `CLIENTIP` notifications on this connection. If omitted, the default is `false`.

Request example with **HTTP transport**:

```
POST /lightstreamer/bind_session.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 36
Content-Type: text/plain

LS_session=S73d162c183916f0dT2729905
```

Request example with **WS transport**:

```
bind_session
LS_session=S73d162c183916f0dT2729905
```

See [Session Creation and Binding Responses](#) section in this chapter (just below) for sample responses.

Session Creation and Binding Responses

Session creation and binding requests share the same responses.

Successful Session Creation or Binding Response

👉 **Tag:** `CONOK` (short of `CON`nection `OK`)

👉 **Format:** `CONOK,<session-ID>,<request-limit>,<keep-alive>,<control-link>`

👉 **Arguments:** 4

◆ **`<session-ID>`**

The Server internal string representing the **session ID**.

◆ **`<request-limit>`**

The **maximum length** allowed by the Server for a client request, expressed in bytes. It is configured through the `<request_limit>` element in the Server configuration file. It is necessary that the system has been configured in advance to prevent any single request from exceeding this limit; however, this information can be useful when batching multiple control requests.

◆ **`<keep-alive>`**

The **longest inactivity time** guaranteed throughout connection life time, expressed in milliseconds.

For a stream connection, when no notifications have been sent for this time, a `PROBE` notification 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. In practice, this value corresponds to the requested `ls_idle_millis` (but for a possible Server upper limit).

Not receiving any message for longer than this time may be the signal of a problem.

◆ **<control-link>**

If a **control link** is specified in the Server configuration file, this is the address (IP address, or hostname, and port) to which every following session rebind and control request must be sent to, in case this requires the establishment of a new socket (possibly a websocket). Opening the new socket towards the "control link" address endpoint may be needed to ensure session affinity in case of clustering.

If no control link is specified in the Server configuration file, this parameter has the special value asterisk "*" (UTF-8 code 0x2a). Its meaning is that the client should send all the session rebind and control requests to the same address to which it opened this stream connection.

Note that, with WS transport, issuing requests directly on the stream connection is always allowed. Yet, the establishment of a new socket may still be needed, for instance in case of an attempt to recovery the session when the current stream connection is unresponsive.

Also note that, upon the first time the "control link" address is used to open a new stream connection to rebind to a session, the streaming activity, that was flowing to the initial address, will switch to the "control link". Depending on the server-side architecture, this may mean moving the communication load to a different part of the infrastructure (see the **Clustering** document for examples).

Edition Note: Server Clustering is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

Response example with HTTP transport:

```
HTTP/1.1 200 OK
Server: Lightstreamer-Server/7.0.0 build 1972
Content-Type: text/enriched; charset=iso-8859-1
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 1 Jan 1970 00:00:00 GMT
Date: Fri, 1 Jul 2016 14:01:19 GMT
Transfer-Encoding: chunked

2E
CONOK,S73d162c183916f0dT2729905,50000,5000,*
```

Response example with WS transport:

```
CONOK,S73d162c183916f0dT2729905,50000,5000,*
```

👉 **Notifications:** After this response, whatever content follows is a **real-time update or any other kind of notification**. With WS transport, they can be **interspersed** with responses to control requests sent on the stream connection. After a `CONOK` is sent the following notifications are also immediately sent (in any order):

- `SERVNAME` (can be omitted on a Binding Response, if unchanged)
- `CLIENTIP` (if available - can also be omitted on a Binding Response, if unchanged)
- `CONS`

- PROG (if requested via `LS_recovery_from ON bind_session`)

Other Session Creation or Binding Responses

👉 **Tag: CONERR** (short of CONnection ERRor)

👉 **Format:** `CONERR,<error-code>,<error-message>`

👉 **Used when:** with both session creation and binding, when the session could not be created or bound.

👉 **Arguments: 2**

◆ `<error-code>`

Error code sent by the Server kernel or by the Metadata Adapter. For a list of supported error codes see [Appendix A](#).

◆ `<error-message>`

Error message sent by Server kernel or by the Metadata Adapter. Note that a message sent by the Metadata Adapter may be empty.

Error response example:

```
CONERR,2,Requested Adapter Set not available
```

👉 **Tag: END** (short of session END)

👉 **Format:** `END,<cause-code>,<cause-message>`

👉 **Used when:** only with session rebinding, when the requested session has just been forcibly closed on the Server side.

👉 **Arguments: 1**

◆ `<cause-code>`

Cause code sent by the Server kernel. For a list of supported cause codes see [Appendix A](#).

◆ `<cause-message>`

Short **cause description** sent by the Server kernel.

End response example:

```
END,8,Session count limit reached
```

Control Requests

Control requests provide means to change the **set of items and fields** sent with real-time updates, **send messages** and **change parameters** of an active subscription or session.

Common Parameters

All control requests (but for message send requests, that will be covered separately) share a common name and some mandatory parameters. In particular one of them, `LS_op`, indicates the specific control operation being requested.

👉 **Request name:** `control`

👉 **Common Parameters:**

◆ `LS_session`

Optional with WS transport. The **session ID** received from the Server at the beginning of the stream connection.

Notes:

- ▶ When omitted with WS transport, its default value is the ID of the last session that was bound to the WS connection (it may be currently still bound or temporarily unbound due to a polling cycle). Hence this requires that a preceding session creation or binding request was issued.
- ▶ When omitted with WS transport to leverage the default provided by a preceding session creation or binding request, the control request is accepted immediately, regardless that the preceding request has started. In any case, the response will always follow the initial response of the preceding request. Obviously, a failure of such preceding request would cause this control request to fail as well.
- ▶ With HTTP transport, it may be specified on the query string, instead of the request body, to address the same session with a multiple request.

◆ `LS_reqId`

ID of the request. It may be any combination of letters and numbers, but typically a progressive number will do, and must be unique within the connection.

◆ `LS_op`

Op-code of the specific control operation requested. See each request for its specific op-code.

Subscription Request

👉 **Parameters:**

◆ `LS_session, LS_reqId`

See the [Common Parameters](#) section of this chapter.

◆ `LS_op=add`

Specifies that a **new subscription** must be added to the session.

◆ `LS_subId`

ID of the subscription. It must be a progressive integer number starting with 1, and must be unique within the session.

◆ `LS_data_adapter`

Optional. Logical name of the **Data Adapter** available in the **Adapter Set** that will provide data for this subscription. This Data Adapter must supply all the requested items. If omitted, a Data Adapter named "DEFAULT" within the Adapter Set is assumed..

◆ `LS_group`

Identification name of the **item group** being subscribed. This name is interpreted by the Metadata Adapter.

◆ **LS_schema**

Identification name of the **field schema** for which the subscription should provide real-time updates. This name is interpreted by the Metadata Adapter.

◆ **LS_selector**

Optional. Identification name of a **selector** related to the items in the subscription. This name is interpreted by the Metadata Adapter.

◆ **LS_mode**

Subscription mode of all the items in subscription.

Notes:

- ▶ Possible values are: `RAW`, `MERGE`, `DISTINCT` and `COMMAND`
- ▶ See the *General Concepts* document for details about subscription modes.

◆ **LS_requested_buffer_size**

Optional. Dimension of the buffers related to the items in the subscription, expressed in number of update events. The Server may limit the buffer size, however. See the *General Concepts* document for more details.

Notes:

- ▶ If set to `unlimited`, no limit is requested to the Server.
- ▶ Default value: `1` if `LS_mode` is `MERGE` and `unlimited` if `LS_mode` is `DISTINCT`.
- ▶ Considered only if `LS_mode` is `MERGE` OR `DISTINCT` and `LS_requested_max_frequency` is not set to `unfiltered`.

◆ **LS_requested_max_frequency**

Optional. Maximum update frequency for the items in the subscription, expressed in updates/sec. The Server may limit the frequency, however. See the *General Concepts* document for more details.

Notes:

- ▶ If set to `unlimited`, the Server forwards each update as soon as possible, with no frequency limit
- ▶ If set to `unfiltered`, the Server forwards each update as soon as possible, with no frequency limit, but also without losses. More precisely, any loss will be signaled through an overflow (`ov`) notification. Considered only if `LS_mode` is `MERGE`, `DISTINCT` OR `COMMAND`.
- ▶ If set to a decimal number (with a dot as decimal separator), the maximum frequency is set to the corresponding number of updates/sec. Considered only if `LS_mode` is `MERGE`, `DISTINCT` OR `COMMAND` (with `COMMAND` the maximum frequency applies to the `UPDATE` commands sent for each key).
- ▶ Default value: `unlimited`.

Edition Note: A further global frequency limit could also be imposed by the Server, depending on Edition and License Type; this specific limit also applies to RAW mode and to unfiltered dispatching. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

◆ **LS_snapshot**

Optional. Requested snapshot length for the items in the subscription, expressed in number of events or as a boolean.

Notes:

- ▶ If set to `true`, the Server sends the snapshot (possibly empty) for the items contained in the subscription. Considered only if `LS_mode` is `MERGE`, `DISTINCT` OR `COMMAND`. In the `DISTINCT` case, the length of the snapshot is decided by the Server.

- ▶ If set to false, the Server does not send the snapshot for the items contained in the subscription.
- ▶ If set to an integer number, the length of the snapshot is set to the corresponding number of updates. Admitted only if `LS_mode` is `DISTINCT`. The Server sends the snapshot for the items contained in the subscription, limiting the length to the requested value. The number of updates received may be lower if the Server imposes a stricter limit or fewer updates are available.
- ▶ Default value: `false`.

◆ **LS_ack**

Optional. Only if the stream connection is on WS transport. If `false` the `REQOK` response is not sent.

Notes:

- ▶ Skipping the `REQOK` response may be useful when issuing many subscriptions at once. Note that a successful subscription is also notified with `SUBOK` on the stream connection.
- ▶ If the request fails, the consequent `REQERR` or `ERROR` response is always sent.
- ▶ Ignored if used with HTTP transport, as an HTTP response is always sent.
- ▶ Default value is `true`.

👉 **Notifications:** If successful, a `SUBOK` or `SUBCMD` notification is sent on the stream connection, followed by a `CONF` notification. After that, real-time updates of the subscribed items start to be delivered, beginning with the snapshot part, if requested. See [Successful Subscription](#) and [Successful Subscription with Command Mode](#) for more information.

Request example with HTTP transport:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 147
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbcbfT2255126&LS_reqId=1&LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS_mode=MERGE
```

Request example with WS transport:

```
control
LS_reqId=1&LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS_mode=MERGE
```

Multiple request example with HTTP transport:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0&LS_session=Sd9fce58fb5dbbcbfT2255126 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 223
Content-Type: text/plain

LS_reqId=1&LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS_mode=MERGE
LS_reqId=2&LS_op=add&LS_subId=2&LS_group=item2&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS_mode=MERGE
```

Multiple request example with WS transport:

```
control
LS_reqId=1&LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS_mode=MERGE
```

```
LS_reqId=2&LS_op=add&LS_subId=2&LS_group=item2&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS_mode=MERGE
```

See [Control Responses](#) section in this chapter for sample responses.

Unsubscription Request

Parameters:

- ◆ **LS_session, LS_reqId**
See the [Common Parameters](#) section of this chapter.
- ◆ **LS_op=delete**
Specifies that an **existing subscription** must be deleted from the session.
- ◆ **LS_subId**
ID of the existing subscription. It must be the same progressive integer number used during subscription.
- ◆ **LS_ack**
Optional. Only if the stream connection is on WS transport. If `false` the `REQOK` response is not sent.

Notes:

- ▶ Skipping the `REQOK` response may be useful when issuing many unsubscriptions at once. Note that a successful unsubscription is also notified with `UNSUB` on the stream connection.
- ▶ If the request fails, the consequent `REQERR` or `ERROR` response is always sent.
- ▶ Ignored if used with HTTP transport, as an HTTP response is always sent.
- ▶ Default value is `true`.

- ▶ **Notifications:** If successful, an `UNSUB` notification is sent on the stream connection, after the last real-time update of subscribed items has been delivered. See [Successful Unsubscription](#) for more information.

Request example with HTTP transport:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 71
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbfbT2255126&LS_reqId=2&LS_op=delete&LS_subId=1
```

Request example with WS transport:

```
control
LS_reqId=2&LS_op=delete&LS_subId=1
```

See [Control Responses](#) section in this chapter for sample responses.

Subscription Reconfiguration Request

Parameters:

- ◆ **LS_session, LS_reqId**

See the [Common Parameters](#) section of this chapter.

- ◆ **LS_op=reconf**
Specifies that an **existing subscription** must be reconfigured.
- ◆ **LS_subId**
ID of the existing subscription. It must be the same progressive integer number used during subscription.
- ◆ **LS_requested_max_frequency**
Optional. New maximum update frequency for the items in the subscription, expressed in updates/sec. See the *General Concepts* document for more details. If not supplied, no modification is requested.

Notes:

- ▶ May be a decimal number (with dot as a decimal separator).
- ▶ If the value is the string `unlimited`, the request is to relieve any existing frequency limit.
- ▶ Admitted only if the subscription is not subscribed to with `unfiltered max frequency`.
- ▶ Considered only if the subscription mode of the subscription is `MERGE`, `DISTINCT` or `COMMAND` (in `COMMAND` mode the maximum frequency applies to the `UPDATE` commands sent for each key).

Edition Note: A further global frequency limit could also be imposed by the Server, depending on Edition and License Type; this specific limit also applies to RAW mode and to unfiltered dispatching. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

- ◆ Note: this control request is designed to change different subscription configuration parameters, in the future. As of TLCP-2.5.0, the only changeable parameter is the subscription maximum frequency.

- 👉 **Notifications:** If successful, a `CONF` notification is sent on the stream connection, before real-time updates of the subscribed items starts to be delivered with the new frequency. See [Subscription Reconfiguration](#) for more information.

Request example with **HTTP transport**:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 102
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbfbfT2255126&LS_reqId=3&LS_op=reconf&LS_subId=1&LS_requested_max_frequency=2.0
```

Request example with **WS transport**:

```
control
LS_reqId=3&LS_op=reconf&LS_subId=1&LS_requested_max_frequency=2.0
```

See [Control Responses](#) section in this chapter for sample responses.

Session Constrain Request

👉 **Parameters:**

- ◆ **LS_session, LS_reqId**

See the [Common Parameters](#) section of this chapter.

◆ **LS_op=constrain**

Specifies that an **existing session** must change its constraints.

◆ **LS_requested_max_bandwidth**

Optional. New maximum bandwidth requested by the client, expressed in kbps. See the *General Concepts* document for more details. If not supplied, no modification is requested.

Notes:

- ▶ May be a decimal number, with a dot as a decimal separator.
- ▶ If the value is the string `unlimited`, the request is to relieve any existing bandwidth limit.

Edition Note: Bandwidth Control is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

- ◆ Note: this control request is designed to change different session constraints, in the future. As of TLCP-2.5.0, the only changeable constraint is the session maximum bandwidth.

- 👉 **Notifications:** If successful, a `CONS` notification is sent on the stream connection, before real-time updates starts to be delivered with the new bandwidth. See [Session Constraints Changed](#) for more information.

Request example with **HTTP transport**:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 95
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbcbfT2255126&LS_reqId=4&LS_op=constrain&LS_requested_max_bandwidth=50.0
```

Request example with **WS transport**:

```
control
LS_reqId=4&LS_op=constrain&LS_requested_max_bandwidth=50.0
```

See [Control Responses](#) section in this chapter for sample responses.

Force Session Rebind Request

👉 Parameters:

◆ **LS_session, LS_reqId**

See the [Common Parameters](#) section of this chapter.

◆ **LS_op=force_rebind**

Specifies that an **existing session** must be forced to rebind.

◆ **LS_polling_millis**

Optional. Expected time between the closing of the connection and the **next binding request**, expressed in milliseconds.

◆ **LS_close_socket**

Optional. If `true` the corresponding stream connection is forcibly closed by the Server. If omitted, the stream connection may be left open and the client may reuse it with both HTTP (by application-wide or system-wide connection pooling) and WS.

- 👉 **Notifications:** If successful, a `LOOP` notification is sent on the stream connection. After that, real-time updates stop being delivered and the session is unbound from the connection. See [Stream Connection Loop](#) for more information.

Request example with **HTTP transport**:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 66
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbfbT2255126&LS_reqId=5&LS_op=force_rebind
```

Request example with **WS transport**:

```
control
LS_reqId=5&LS_op=force_rebind
```

See [Control Responses](#) section in this chapter for sample responses.

Session Destroy Request

👉 Parameters:

- ◆ `LS_session, LS_reqId`

See the [Common Parameters](#) section of this chapter.

- ◆ `LS_op=destroy`

Specifies that an **existing session** must be terminated.

- ◆ `LS_cause_code`

Optional. A **numeric code** to be reported in the consequent `END` message that will be received on the stream connection in place of the default code `31`. Useful if the request is issued by some server-side process.

Notes:

- ▶ Only `0` or a negative code are supported; a positive code will be collapsed to `0`.

- ◆ `LS_cause_message`

Optional. Considered only if `LS_cause_code` is also supplied. A **text** to be reported in the consequent `END` message that will be received on the stream connection. Useful if the request is issued by some server-side process.

Notes:

- ▶ If `LS_cause_code` is not present, the cause in the `END` message will be provided by the Server, regardless of this setting.
- ▶ On the other hand, if `LS_cause_code` is present and `LS_cause_message` is not specified, then the reported message will be "null".
- ▶ The supplied message should be in simple ASCII, otherwise it might be altered in order to be sent to the client; multiline text is also not allowed.
- ▶ The supplied message should be short. If longer than 35 characters and if a content-length limit on the connection content is in force, it may be replaced with "[Custom message skipped]".

◆ **LS_close_socket**

Optional. If `true` the corresponding stream connection is forcibly closed by the Server. If omitted, the stream connection is left open and the client may reuse it with both HTTP (by application-wide or system-wide connection pooling) and WS.

👉 **Notifications:** If successful, an `END` notification is sent on the stream connection. After that, real-time updates stop being delivered and the session is terminated. See [Stream Connection End](#) for more information.

Request example with HTTP transport:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 61
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbcbfT2255126&LS_reqId=6&LS_op=destroy
```

Request example with WS transport:

```
control
LS_reqId=6&LS_op=destroy
```

See [Control Responses](#) section in this chapter for sample responses.

MPN Control Requests

Mobile Push Notifications (MPN) requests are a subset of control requests dedicated to the interaction with the MPN Module, a component of the Server that can route real-time updates to mobile devices via their native push notification service (e.g. APNs for Apple™ platforms and FCM for Google™ platforms). See Chapter 5 of the *General Concepts* document for an in-depth introduction to the MPN Module.

Edition Note: MPN is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

MPN Device Registration Request

👉 **Parameters:**

◆ **LS_session, LS_reqId**

See the [Common Parameters](#) section of this chapter.

◆ **LS_op=register**

Specifies that a **new or existing MPN device** must be registered.

Note:

- ▶ Registering an MPN device is a mandatory operation to enable any subsequent MPN request on the session. Registering the same device multiple times, on different sessions or even on the same one, poses no issues.

◆ **PN_type**

Platform type of the device. Currently supported platform types are:

- ▶ **Apple** for Apple™ platforms, such as iOS™, macOS™ and tvOS™.
- ▶ **Google** for Google™ platforms, such as Android™.

- ▶ No other platforms are supported at time of writing.

◆ **PN_appId**

ID of the application. Corresponds to the app's bundle ID on Apple™ platforms and to the app's package name on Google™ platforms.

Note:

- ▶ For an application ID to be accepted, it must first be added to the list of known applications in the Server configuration file corresponding to the platform's MPN provider. See §5.5 of the *General Concepts* document for more information.

◆ **PN_deviceToken**

Platform's known token for the device. Corresponds to the device token on Apple™ platforms and to the registration ID on Google™ platforms.

Notes:

- ▶ The device token must be acquired through the platform's native registration process for the device.
- ▶ If the device token changed with the latest call to the platform's native registration process, this parameter should report the **previous value**, while the current one should be specified with the `PN_newDeviceToken` parameter. See below.

◆ **PN_newDeviceToken**

Optional. Platform's current token for the device, if changed. Tracking token changes lets the Server keep existing MPN subscriptions active. If omitted, the device is considered new and existing MPN subscriptions are lost. See §5.1.1 of the *General Concepts* document for more information.

- 👉 **Notifications:** If successful, an `MPNREG` notification is sent on the stream connection. See [MPN Device Registered](#) for more information.

Request example with **HTTP transport**:

```
POST /lightstreamer/control.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 102
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbcbfT2255126&LS_reqId=7&LS_op=register&PN_type=Apple&PN_appId=com.mydomain.myapp&PN_deviceToken=740f4707bebcf74f9b7c25d48e3358945f6aa01da5ddb387462c7eaf61bb78ad
```

Request example with **WS transport**:

```
control
LS_reqId=7&LS_op=register&PN_type=Apple&PN_appId=com.mydomain.myapp&PN_deviceToken=740f4707bebcf74f9b7c25d48e3358945f6aa01da5ddb387462c7eaf61bb78ad
```

See [Control Responses](#) section in this chapter for sample responses.

MPN Device Badge Reset

👉 **Parameters:**

◆ **LS_session, LS_reqId**

See the [Common Parameters](#) section of this chapter.

◆ **LS_op=reset_badge**

Specifies that an **existing MPN device** must have its badge reset.

- ◆ **PN_deviceId**
ID of the MPN device. Must be the same ID notified by `MPNREG` after a successful registration.

Note:

- ▶ For the MPN device ID to be accepted, the device must have been registered *on the current session*. Registration is a mandatory operation to enable any subsequent MPN request.

- 👉 **Notifications:** If successful, an `MPNZERO` notification is sent on the stream connection. See [MPN Device Badge Reset](#) for more information.

MPN Subscription Activation

👉 Parameters:

- ◆ **LS_session, LS_reqId**
See the [Common Parameters](#) section of this chapter.
- ◆ **LS_op=activate**
Specifies that a **new MPN subscription** must be activated or an **existing MPN subscription** must be changed.
- ◆ **PN_deviceId**
ID of the MPN device. Must be the same ID notified on `MPNREG` after a successful registration.
Note:
 - ▶ For the MPN device ID to be accepted, the device must have been registered *on the current session*. Registration is a mandatory operation to enable any subsequent MPN request.
- ◆ **PN_subscriptionId**
Optional. ID of an existing MPN subscription. Must be the same ID notified on `MPNOK` after a successful activation.
Notes:
 - ▶ If omitted, the MPN subscription is created as new.
 - ▶ If specified, an MPN subscription corresponding to the ID must already exist, and in this case its parameters are changed according to the request. Moreover, if the existing MPN subscription is in `TRIGGERED` state, it will be reactivated.
- ◆ **LS_subId**
ID of the subscription. It must be a progressive integer number starting with 1, and must be unique within the session.
Notes:
 - ▶ The `LS_subId` and `PN_subscriptionId` identifiers should not be confused.
 - ▶ `LS_subId` is a transient, client-assigned progressive integer. Like the same parameter for real-time subscriptions, its scope is limited to the current session and its use is only to later associate the MPN subscription ID through an `MPNOK` notification (see also below).
 - ▶ `PN_subscriptionId` is a persistent, Server-assigned 36-character UUID. Its scope is beyond the session life-cycle and its use is precisely to identify an MPN subscription within the current or any subsequent session. During an activation

request it must be specified only if it is a modification of an existing MPN subscription.

◆ **LS_data_adapter**

Optional. Logical name of the **Data Adapter** available in the **Adapter Set** that will provide data for this subscription. This Data Adapter must supply all the requested items. If omitted, a Data Adapter named “DEFAULT” within the Adapter Set is assumed.

◆ **LS_group**

Identification name of the **item group** being subscribed. This name is interpreted by the Metadata Adapter.

◆ **LS_schema**

Identification name of the **field schema** for which the subscription should provide real-time updates. This name is interpreted by the Metadata Adapter.

◆ **LS_mode**

Subscription mode of all the items in subscription.

Notes:

- ▶ Possible values are: **MERGE** and **DISTINCT**
- ▶ See the *General Concepts* document for details about subscription modes.

◆ **LS_requested_buffer_size**

Optional. Dimension of the buffers related to the items in the subscription, expressed in number of update events. The Server may limit the buffer size, however. See the *General Concepts* document for more details.

Notes:

- ▶ If set to **unlimited**, no limit is requested to the Server.
- ▶ Default value: **1** if **LS_mode** is **MERGE** and **unlimited** if **LS_mode** is **DISTINCT**.

◆ **LS_requested_max_frequency**

Optional. Maximum update frequency for the items in the subscription, expressed in updates/sec. The Server may limit the frequency, however. See the *General Concepts* document for more details.

Notes:

- ▶ If set to **unlimited**, the Server forwards each update as soon as possible, with no frequency limit.
- ▶ If set to a decimal number (with a dot as decimal separator), the maximum frequency is set to the corresponding number of updates/sec. Considered only if **LS_mode** is **MERGE**, **DISTINCT** or **COMMAND** (with **COMMAND** the maximum frequency applies to the **UPDATE** commands sent for each key).
- ▶ Default value: **unlimited**.

Edition Note: A further global frequency limit could also be imposed by the Server, depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

◆ **PN_notificationFormat**

Notification format. JSON structure to be used for formatting a subscription update into a native push notification. Must be in accordance with the specifications of the MPN device's platform.

Notes:

- ▶ The use of a JSON structure ensures that new features introduced by the platform's MPN provider can be introduced with minimum effort. Anyway, for safety

reasons, the JSON structure is checked for compliance against known platform's specifications. Unsupported fields may be stripped out.

- ▶ Fields in the JSON structure may contain arguments that are expanded by the Server with values from the subscription update. See §5.2.1 of the *General Concepts* document for more information.

◆ `PN_trigger`

Optional. Trigger expression. A Java boolean expression that triggers the sending of a native push notification.

Notes:

- ▶ If omitted, the MPN subscription sends a native push notification for each subscription update.
- ▶ If specified, the MPN subscription tests the trigger expression on each subscription update, and sends a native push notification only the first time the expression evaluates to true. After that, the MPN subscription enters the `TRIGGERED` state.
- ▶ The expression may contain arguments that are expanded by the Server with fields of the subscription update. See §5.2.2 of the *General Concepts* document for more information.

◆ `PN_coalescing`

Optional. Coalescence flag. A flag that specifies whether the MPN subscription must be coalesced with an existing subscription that shares the same **base parameters**: Data Adapter, Group, Schema, Mode and Trigger expression.

Notes:

- ▶ Can't be `true` if `PN_subscriptionId` is specified: coalescence makes sense only when activating a new MPN subscription.
- ▶ If `false` or omitted, the MPN subscription is always activated as new, whatever the parameters. If an MPN subscription with the same base parameters exists, they will send duplicated notifications to the MPN device.
- ▶ If `true` and an MPN subscription with the same base parameters exists, its parameters are changed according to the request. In other words, the request is interpreted as a change request, and not as new activation. Moreover, if the existing MPN subscription is in `TRIGGERED` state, it will be reactivated.
- ▶ If `true` and no MPN subscriptions with the same base parameters exist, the MPN subscription is activated as new.
- ▶ See §5.2.3 of the *General Concepts* document for more information.

- 👉 **Notifications:** If successful, an `MPNOK` notification is sent on the stream connection. See [MPN Subscription Activated](#) for more information.

MPN Subscription Reconfiguration

👉 Parameters:

◆ `LS_session, LS_reqId`

See the [Common Parameters](#) section of this chapter.

◆ `LS_op=pn_reconf`

Specifies that an **existing MPN subscription** must be changed, but only in the notification part, without modifying the subscription part.

Note:

- ▶ Any notification parameter (Max Frequency, Notification Format, and Trigger expression) can be omitted if not meant to be modified.
- ▶ If the current MPN subscription is in `TRIGGERED` state, it will also be reactivated. This is done always, even if no notification parameter is specified.
- ▶ The subscription parameters (Data Adapter, Group, Schema, and Mode) have to be specified, because they have to be consistent with the notification parameters. In case they don't match the current MPN subscription, the request will be refused. The Adapter Set of the current session should also be the same as the Adapter Set that provides data for the current MPN subscription.

◆ `PN_deviceId`

ID of the MPN device. Must be the same ID notified on `MPNREG` after a successful registration.

Note:

- ▶ For the MPN device ID to be accepted, the device must have been registered on *the current session*. Registration is a mandatory operation to enable any subsequent MPN request.

◆ `PN_subscriptionId`

ID of an existing MPN subscription. Must be the same ID notified on `MPNOK` after a successful activation.

◆ `LS_data_adapter`

Optional. Logical name of the **Data Adapter** that provides data for the current MPN subscription. If omitted, a Data Adapter named "`DEFAULT`" is assumed.

◆ `LS_group`

Identification name of the **item group** referred by the current MPN subscription.

◆ `LS_schema`

Identification name of the **field schema** referred by the current MPN subscription.

◆ `LS_mode`

Subscription mode of the items referred by the current MPN subscription.

◆ `LS_requested_max_frequency`

Optional. Maximum update frequency for the items in the subscription, expressed in updates/sec.

- ▶ If specified, it replaces any limit in use for the current MPN subscription. See the notes in [MPN Subscription Activation](#) for details.
- ▶ If not specified, then the value used for the current MPN subscription is kept.

◆ `PN_notificationFormat`

Optional. Notification format. JSON structure to be used for formatting a subscription update into a native push notification.

- ▶ If specified, it replaces the format in use for the current MPN subscription. See the notes in [MPN Subscription Activation](#) for details.
- ▶ If not specified, then the value used for the current MPN subscription is kept.

◆ `PN_trigger`

Optional. Trigger expression. A Java boolean expression that triggers the sending of a native push notification.

- ▶ If specified, it replaces any trigger in use for the current MPN subscription. See the notes in [MPN Subscription Activation](#) for details.
- ▶ If not specified, then the value used for the current MPN subscription is kept.

- ▶ If an empty string is supplied, this removes any trigger in use for the current MPN subscription, hence the MPN subscription will send a native push notification for each subscription update.

👉 **Notifications:** If successful, an `MPNCONF` notification is sent on the stream connection. See [MPN Subscription Activated](#) for more information.

MPN Subscription Deactivation

👉 Parameters:

- ◆ `LS_session, LS_reqId`
See the [Common Parameters](#) section of this chapter.
- ◆ `LS_op=deactivate`
Specifies that a **one or more existing MPN subscription** must be deactivated.
- ◆ `PN_deviceId`
ID of the MPN device. Must be the same ID notified by `MPNREG` after a successful registration.
Note:
 - ▶ For the MPN device ID to be accepted, the device must have been registered *on the current session*. Registration is a mandatory operation to enable any subsequent MPN request.
- ◆ `PN_subscriptionId`
Optional. ID of an existing MPN subscription. Must be the same ID notified on `MPNOK` after a successful activation.
Note:
 - ▶ If specified, only the MPN subscription corresponding to the ID is deactivated.
 - ▶ If omitted and `PN_subscriptionStatus` is also omitted, all MPN subscriptions of the MPN device are deactivated.
- ◆ `PN_subscriptionStatus`
Optional. Status of existing MPN subscriptions. Must be one of the following: `ACTIVE` OR `TRIGGERED`.
 - ▶ If specified, all MPN subscriptions corresponding to the status are deactivated.
 - ▶ If omitted and `PN_subscriptionId` is also omitted, all MPN subscriptions of the MPN device are deactivated.

👉 **Notifications:** If successful, an `MPNDEL` notification is sent on the stream connection for each deactivated MPN subscription. See [MPN Subscription Deactivated](#) for more information.

Message Send Request

Sending a message is still a control request, but makes use of a different request name and hence does not require the `LS_op` parameter.

👉 **Request name:** `msg`

👉 Parameters:

- ◆ `LS_session, LS_reqId`

See the [Common Parameters](#) section of this chapter.

◆ **LS_message**

Any text string. This string should be interpreted and verified by the **Metadata Adapter**, so the developer is free to decide its meaning.

◆ **LS_sequence**

Optional. 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.

Notes:

- ▶ 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 omitted, the message is processed immediately, possibly concurrently with others, regardless of any progressive number assigned. Yet, if a progressive number is specified, previous numbers for which no unsequenced message has been received after a server-side timeout may be notified by the Server as skipped.
- ▶ The special identifier "UNORDERED_MESSAGES", used in previous text protocols, is now reserved and can't be used. If specified leads to an error.

◆ **LS_msg_prog**

Optional. The **progressive number** of the message within the specified sequence, starting from 1. It is also used to identify the message in the `MSGDONE` and `MSGFAIL` notifications, even if no sequence is specified.

Notes:

- ▶ It is mandatory whenever `LS_sequence` is specified or `LS_outcome` is set to `true`.
- ▶ Even if `LS_sequence` is omitted *and* `LS_outcome` is set to `false`, it can be specified to enforce checks for duplicates and missing messages (see `LS_sequence`).

◆ **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, expressed in milliseconds.

Notes:

- ▶ If too high or not specified, the timeout is assigned by the Server, based on its own configuration.
- ▶ If `LS_sequence` is omitted, the setting is ignored.

◆ **LS_ack**

Optional. Only if the stream connection is on WS transport. If `false` the `REQOK` response is not sent.

Notes:

- ▶ Skipping the `REQOK` response may be useful when sending frequent loads of messages with no requirements for processing. E.g. when sending the status of a frequently updated object. Note that a successful message submission is also notified with `MSGDONE` on the stream connection.
- ▶ If the request fails, the consequent `REQERR` or `ERROR` response is always sent.
- ▶ Ignored if used with HTTP transport, as an HTTP response is always sent.

- ▶ Default value is `true`.

◆ **LS_outcome**

Optional. If `false` the `MSGDONE` or `MSGFAIL` notification is not sent.

Notes:

- ▶ As with the `LS_ack` argument, skipping the `MSGDONE` or `MSGFAIL` notification may be useful when sending frequent loads of messages with no requirements for processing. E.g. when sending the status of a frequently updated object.
- ▶ If both `LS_ack` and `LS_outcome` are `false`, there will be no notification at all for a successful message submission (a “fire and forget” policy).
- ▶ Default value is `true`.

👉 **Notifications:** If `LS_outcome` is omitted or set to `true`, a `MSGDONE` or a `MSGFAIL` notification is sent on the stream connection, depending if the message is successfully or unsuccessfully delivered, respectively. See [Message Successfully Sent](#) and [Message Send Failed](#) for more information.

Request example with **HTTP transport**:

```
POST /lightstreamer/msg.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 95
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbfbT2255126&LS_reqId=11&LS_sequence=CHAT&LS_msg_prog=1&LS_message=Hello
```

Request example with **WS transport**:

```
msg
LS_reqId=11&LS_sequence=CHAT&LS_msg_prog=1&LS_message=Hello
```

See *Control Responses* section in this chapter (just below) for sample responses.

Control Responses

All control requests, including message send requests, share the same responses.

Successful Control Response

👉 **Tag:** **REQOK** (short of REQuest OK)

👉 **Format:** `REQOK,<request-ID>`

👉 **Arguments:** 1

◆ `<request-ID>`

The **ID** of the request.

Response example:

```
REQOK, 1
```

Other Control Responses

👉 **Tag:** REQERR (short of REQuest ERRor)

👉 **Format:** REQERR,<request-ID>,<error-code>,<error-message>

👉 **Used when:** an error occurred and the request could not be completed.

👉 **Arguments:** 3

◆ <request-ID>

The **ID** of the request.

◆ <error-code>

Error code sent by the Server kernel or by the Metadata Adapter. For a list of supported error codes see [Appendix B](#).

◆ <error-message>

Error message sent by Server kernel or by the Metadata Adapter. Note that a message sent by the Metadata Adapter may be empty.

Error response example:

```
REQERR,19,Specified subscription not found
```

👉 **Tag:** ERROR

👉 **Format:** ERROR,<error-code>,<error-message>

👉 **Used when:** the request could not be interpreted or processed.

👉 **Arguments:** 2

◆ <error-code>

Error code sent by the Server kernel. For a list of supported error codes see [Appendix B](#).

◆ <error-message>

Error message sent by Server kernel.

Error response example:

```
ERROR,68,Internal error during request processing
```

Client Heartbeat

The Heartbeat is a pseudo-request that has the sole purpose of keeping the client-server communication channel engaged. This can serve various purposes:

1. Preventing the communication infrastructure from closing an inactive socket that is ready for reuse for more HTTP control requests, to avoid connection reestablishment overhead. However it is not guaranteed that the connection will be kept open, as the underlying TCP implementation may open a new socket each time a HTTP request needs to be sent.

2. Detecting when a WebSocket connection has been interrupted but not explicitly closed.
3. Allowing the Server to detect cases in which the client is stuck but the socket of the stream connection is kept open by some intermediate node.
This must be done in combination with supplying the `LS_inactivity_millis` parameter to the `create_session` and `bind_session` requests.

Note that any Control Request has the same effect of a Heartbeat, hence no Heartbeat is needed as long as Control Requests are sent.

This request, obviously, is not meant to receive a response.

👉 **Request name:** `heartbeat`

👉 **Parameters:**

◆ `LS_session`

Optional with WS transport. The **session ID** received from the Server at the beginning of the stream connection. This is only needed for the purpose 3 above, whereby the client declares that it is still listening to the stream connection for the specified session.

Notes:

- ▶ When omitted with WS transport, it obviously refers to the last session that was bound to the WS connection (it may be currently still bound or temporarily unbound due to a polling cycle), if any.
- ▶ With HTTP transport, it may be specified on the query string, instead of the request body.

👉 **Notifications:** No notifications are involved.

Request example with **HTTP transport:**

```
POST /lightstreamer/heartbeat.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 36
Content-Type: text/plain

LS_session=Sd9fce58fb5dbbfbT2255126
```

Request example with **WS transport:**

```
heartbeat
LS_session=Sd9fce58fb5dbbfbT2255126
```

or:

```
heartbeat
<empty line>
```

(note that, since the request here has no parameters, the final line terminator must be included).

The response is returned only with HTTP transport. With WS transport there is no response, but for the case an of error in the formulation of the request.

Successful Response

👉 **Tag:** **REQOK** (short of REQuest OK)

👉 **Format:** REQOK

Response example (only with **HTTP transport**):

```
REQOK
```

Other Responses

Error response example:

👉 **Tag:** ERROR

👉 **Format:** ERROR,<error-code>,<error-message>

👉 **Used when:** the request could not be interpreted or processed. Note that there is no error if the specified session is not found.

👉 **Arguments:** 2

◆ <error-code>

Error code sent by the Server kernel. For a list of supported error codes see [Appendix B](#).

◆ <error-message>

Error message sent by Server kernel.

Error response example:

```
ERROR,68,Internal error during request processing
```

WebSocket establishment check

This is a pseudo-request that has the sole purpose of testing a newly established WebSocket, to ensure that the communication channel is working with no interference from intermediate nodes. It consists in the Server simply echoing the request.

For this reason, the request is only allowed on a WebSocket and it is typically expected to be the first one issued on the WebSocket.

The Server will send the response as soon as possible and, in particular, the response is guaranteed to precede the responses of any other subsequent requests.

👉 **Request name:** wsok

👉 **Parameters:** No parameters are involved.

👉 **Notifications:** No notifications are involved.

Request example with **WS transport**:

```
wsok
```

Note that, not only does the request require no parameters, but also the parameter line, needed with other types of requests (and possibly empty), is missing in this case.

Successful Response

👉 **Tag:** **WSOK** (short of WebSocket OK)

👉 **Format:** `wsok`

Response example:

```
wsok
```

Other Responses

There are no error responses.

3 Notification Reference

This chapter reports the complete syntax reference for all TLCP notifications. The general syntax is discussed in [Common Response and Notification Syntax](#), together with a sample algorithm for response parsing. It is reported here for your convenience:

- 👉 **Notifications** general syntax (with HTTP Transport the response is in the body of the HTTP response):

```
<tag>,<argument1>,...,<argumentN>
```

Line separator is **CR-LF**. Encoding is **UTF-8**.

Real-Time Update

- 👉 **Tag: U** (short of Update)

- 👉 **Format:** U,<subscription-ID>,<item>,<field-1-value>|<field-2-value>|...|<field-N-value>

- 👉 **Format:** U,<subscription-ID>,<item>,<field-1-value>|^<number-of-unchanged-fields>|...|<field-N-value>

- 👉 **Format:** U,<subscription-ID>,<item>,<field-1-value>|^P<JSON Patch value>|...|<field-N-value>

- 👉 **Format:** U,<subscription-ID>,<item>,<field-1-value>|^T<TLCP-diff value>|...|<field-N-value>

- 👉 **Used when:** to send a real-time update on the content of the fields of an item. The notification is also used to send the item's snapshot.

- 👉 **Arguments: 3**

- ◆ <subscription-ID>

The **ID** of the subscription.

- ◆ <item>

Index of the item being updated. The index is 1-based, and the order of items in the subscription is determined by the **Metadata Adapter**.

- ◆ <field-1-value>|...|<field-N-value>

Pipe-separated list of field values. See below. The order of fields in the subscription is determined by the **Metadata Adapter**.

Decoding the Pipe-Separated List of Values

The third argument of a real-time update contains the field values for the specified item. The second-level syntax of this argument is designed to be as compact as possible.

The following encoding rules apply:

- ▶ Each value is an **UTF-8 string**.
- ▶ An **empty value** means the field is **unchanged** compared to the previous update of the same field.
- ▶ A value corresponding to a **hash sign** “#” means the field is **null**.
- ▶ A value corresponding to a **dollar sign** “\$” means the field is **empty**.

- ▶ A value corresponding to a **caret followed by a number**, such as “`^3`”, means the following number of fields are **unchanged**.
- ▶ A value starting with a **caret followed by a letter**, such as “`^P`”, means the remaining part of the value represents the difference between the previous and new update, expressed in a “diff” format, where the letter is the tag of the format used.
As a consequence, the real update value should be computed based on the previous update of the same field and the provided difference, according to the format.
The available tags and related “diff” formats are the following:
 - **P – JSON Patch**
The value format is specified by [RFC 6902](#).
The JSON Patch algorithm can be applied only between values that are valid JSON representations. Hence, upon reception of a JSON Patch, it is guaranteed that the previous update value is a valid JSON representation (and, obviously, the new update value will be a valid JSON representation too).
 - **T – TLCP-diff**
The value format is a custom encoding. The format is specified in [Appendix D](#).
This format can be applied to all strings, only provided that their encoding in UTF-16 (the format used internally in Java) doesn't contain surrogate pairs. Obviously, if a TLCP-diff is received, this guarantees that both the previous and new value are compliant.
- ▶ Meta characters, such as the pipe “`|`”, CR-LF, etc., are **percent-encoded**.

A simple decoding algorithm is the following:

- 👉 Set a pointer to the first field of the schema.
- 👉 Look for the next pipe “`|`” from left to right and take the substring to it, or to the end of the line if no pipe is there.
- 👉 Evaluate the substring:
 - ◆ If its value is empty, the pointed field should be left **unchanged** and the pointer moved to the next field.
 - ▶ Note: this case never happens on the first update of a subscription.
 - ◆ Otherwise, if its value corresponds to a single hash sign “`#`” (UTF-8 code 0x23), the pointed field should be set to a **null value** and the pointer moved to the next field.
 - ◆ Otherwise, if its value corresponds to a single dollar sign “`$`” (UTF-8 code 0x24), the pointed field should be set to an **empty value** (“”) and the pointer moved to the next field.
 - ◆ Otherwise, if its value begins with a caret “`^`” (UTF-8 code 0x5E):
 - ▶ Note: this case never happens on the first update of a subscription.
 - ▶ check if the following character is a letter or a digit
 - ▶ case digit:
 - take the substring following the caret and convert it to an integer number;
 - for the corresponding count, leave the fields **unchanged** and move the pointer forward;
 - e.g. if the value is “`^3`”, leave unchanged the pointed field and the following two fields, and move the pointer 3 fields forward.
 - ▶ case letter:
 - Note: this case never happens if the value of the pointed field is currently null.

- take the substring following the letter and decode any **percent-encoding**
 - use the obtained value as a difference to be applied to the value of the pointed field, based on the format specified by the letter
 - set the pointed field to the value finally obtained
- ◆ Otherwise, the value is an actual content: decode any **percent-encoding** and set the pointed field to the decoded value, then move the pointer to the next field.
- ▶ Note: “#”, “\$” and “^” characters are percent-encoded if occurring at the beginning of an actual content (and only in this case).

👉 Return to the second step, unless there are no more fields in the schema.

Snapshot vs Real-Time Updates

As said, this notification can be used both to carry snapshot information and to provide real-time updates, whereas the snapshot may be made of 0 or multiple such notifications depending on the subscription mode. The following table resumes when a notification is part of a snapshot or it is a real-time update, based on the subscription parameters.

LS_snapshot	LS_mode	EOS notification already received?	First notification?	The notification is...
false or missing	-	(cannot be received)	-	real-time update
true or a number (a number is only meaningful for DISTINCT mode)	RAW	(cannot be received)	-	real-time update
	MERGE	(cannot be received)	YES	snapshot
			NO	real-time update
	DISTINCT	YES	-	real-time update
		NO	-	snapshot
	COMMAND	YES	-	real-time update
NO		-	snapshot	

Notification Examples

As an example, suppose a subscription to a typical stock quote adapter, with subscription ID 3, just one item, and the following fields in the schema: `timestamp`, `price`, `change`, `minimum`, `maximum`, `bid`, `ask`, `open`, `close` and `status`.

This is a hypothetical stream of real-time updates and the effect it has on field values on the client:

```
U,3,1,20:00:33|3.04|0.0|2.41|3.67|3.03|3.04|##|#$
```

timestamp	price	change	minimum	maximum	bid	ask	open	close	status
20:00:33	3.04	0.0	2.41	3.67	3.03	3.04	<null>	<null>	<empty>
Changed	Changed	Changed	Changed	Changed	Changed	Changed	Changed	Changed	Changed

The initial update (the snapshot) carries information for all the fields. It contains two null fields (`open` and `close`) and an empty field (`status`).

```
U,3,1,20:00:54|3.07|0.98|||3.06|3.07|||Suspended
```

timestamp	price	change	minimum	maximum	bid	ask	open	close	status
20:00:54	3.07	0.98	2.41	3.67	3.06	3.07	<null>	<null>	Suspended
Changed	Changed	Changed	Unchanged	Unchanged	Changed	Changed	Unchanged	Unchanged	Changed

This update contains 4 unchanged fields: `minimum`, `maximum`, `open` and `close`.

```
U,3,1,20:04:16|3.02|-0.65|||3.01|3.02|||S
```

timestamp	price	change	minimum	maximum	bid	ask	open	close	status
20:04:16	3.02	-0.65	2.41	3.67	3.01	3.02	<null>	<null>	<empty>
Changed	Changed	Changed	Unchanged	Unchanged	Changed	Changed	Unchanged	Unchanged	Changed

This update contains again 4 unchanged fields: `minimum`, `maximum`, `open` and `close`. Moreover, it sets the `status` field to its initial empty value.

```
U,3,1,20:04:40|^4|3.02|3.03|||
```

timestamp	price	change	minimum	maximum	bid	ask	open	close	status
20:04:40	3.02	-0.65	2.41	3.67	3.02	3.03	<null>	<null>	<empty>
Changed	Unchanged	Unchanged	Unchanged	Unchanged	Changed	Changed	Unchanged	Unchanged	Unchanged

This update includes the special syntax for multiple contiguous unchanged fields: `price`, `change`, `minimum` and `maximum`. The `open`, `close` and `status` fields are also unchanged.

```
U,3,1,20:06:10|3.05|0.32|^7
```

timestamp	price	change	minimum	maximum	bid	ask	open	close	status
20:06:10	3.03	-0.32	2.41	3.67	3.02	3.03	<null>	<null>	<empty>
Changed	Changed	Changed	Unchanged	Unchanged	Unchanged	Unchanged	Unchanged	Unchanged	Unchanged

This update includes again the special syntax for multiple contiguous unchanged fields: `minimum`, `maximum`, `bid`, `ask`, `open`, `close` and `status`.

```
U,3,1,20:06:49|3.08|1.31|||3.08|3.09|||
```

timestamp	price	change	minimum	maximum	bid	ask	open	close	status
20:06:49	3.08	1.31	2.41	3.67	3.08	3.09	<null>	<null>	<empty>
Changed	Changed	Changed	Unchanged	Unchanged	Changed	Changed	Unchanged	Unchanged	Unchanged

The final update contains once more 4 unchanged fields: `minimum`, `maximum`, `open` and `close`.

Notification Examples with the use of “diff” values

For this example, suppose a subscription to a dynamic text, with subscription ID 5, just one item, and the following fields in the schema: `timestamp`, `text`, such as the `text` field has a complex structure that is carried with Json structures.

This is a hypothetical stream of real-time updates and the effect it has on field values on the client. Percent decoding is also needed for some values:

```
U,5,1,20:00:33|{"text": "aa%7Cbb", "attributes": [ { "font": "courier" }, "... " ] }
```

	timestamp	text
diff received		
actual value	20:00:33	{ "text": "aa bb", "attributes": [{ "font": "courier" }, "..."] }

The initial update (the snapshot) carries information for all the fields. With "... " we represent a possibly long snippet in the Json value.

```
U,5,1,20:00:54|^P[{"op":"replace","path":"/text","value":"aa%7Cbb%7Ccc"},
{"op":"add","path":"/attributes/1","value":"bold"}]
```

	timestamp	text
diff received		[{"op":"replace","path":"/text","value":"aa bb cc"}, {"op":"add","path":"/attributes/1","value":"bold"}]
actual value	20:00:54	{ "text": "aa bb cc", "attributes": [{"font": "courier"}, "bold", "..."] }

This update contains the new values. Since both the previous and the new value of the `text` field happen to be valid JSON representations, the Server chose to provide the field update as a JSON Patch.

```
U,5,1,20:04:16|
```

	timestamp	text
diff received		
actual value	20:04:16	{ "text": "aa bb cc", "attributes": [{"font": "courier"}, "bold", "..."] }

In the final update, the `text` field is unchanged; this is carried, as usual, by an empty field.

Other Subscription-Related Notifications

The following notifications provide insights on the status of a specific subscription.

Successful Subscription

- 👉 **Tag:** SUBOK (short of SUBscription OK)
- 👉 **Format:** SUBOK,<subscription-ID>,<num-items>,<num-fields>
- 👉 **Used when:** to notify of a successful subscription (but for those that make use of `COMMAND` mode).
- 👉 **Arguments: 3**
 - ◆ <subscription-ID>
The ID of the subscription.
 - ◆ <num-items>
Number of items in the subscription. The number and the order of items in the subscription is determined by the **Metadata Adapter**.
 - ◆ <num-fields>

Number of fields in the subscription. The number and the of order of fields in the subscription is determined by the **Metadata Adapter**.

This notification is sent after a subscription request has been received and the corresponding subscription has been activated on the Server. After this notification, real-time updates related to the subscription items and fields start to be sent.

Notification Example:

```
SUBOK, 3, 1, 10
```

Successful Subscription with Command Mode

- 👉 **Tag: SUBCMD** (short of SUBscription in CoMmanD mode)
- 👉 **Format:** SUBCMD,<subscription-ID>,<num-items>,<num-fields>,<key-field>,<command-field>
- 👉 **Used when:** to notify of a successful subscription that makes use of `COMMAND` mode.
- 👉 **Arguments: 5**
 - ◆ <subscription-ID>
The **ID** of the subscription.
 - ◆ <num-items>
Number of items in the subscription. The number and the order of items in the subscription is determined by the **Metadata Adapter**.
 - ◆ <num-fields>
Number of fields in the subscription. The number and the of order of fields in the subscription is determined by the **Metadata Adapter**.
 - ◆ <key-field>
Index of the field that contains the key. The index is 1-based; the order of fields in the subscription is determined by the **Metadata Adapter**.
 - ◆ <command-field>
Index of the field that contains the command. The index is 1-based; the order of fields in the subscription is determined by the **Metadata Adapter**.

This notification is sent after a subscription request has been received and the corresponding subscription has been activated on the Server. After this notification, real-time updates related to the subscription items and fields start to be sent.

Notification Example:

```
SUBCMD, 3, 1, 10, 1, 2
```

Successful Unsubscription

- 👉 **Tag: UNSUB** (short of UNSUBscription)
- 👉 **Format:** UNSUB,<subscription-ID>
- 👉 **Used when:** to notify of a successful unsubscription.

👉 Arguments: 1

- ◆ `<subscription-ID>`
The **ID** of the subscription.

This notification is sent after an unsubscription request has been received and the corresponding subscription has been deactivated on the Server. After this notification, no more real-time updates related to the subscription items and fields will be sent.

Notification Example:

```
UNSUB, 3
```

End of Snapshot

👉 **Tag: EOS** (short of End Of Snapshot)

👉 **Format:** `EOS,<subscription-ID>,<item>`

👉 **Used when:** to notify the end of the snapshot of an item.

👉 Arguments: 2

- ◆ `<subscription-ID>`
The **ID** of the subscription.
- ◆ `<item>`
Index of the item whose snapshot has ended. The index is 1-based, and the order of items in the subscription is determined by the **Metadata Adapter**.

This notification is sent immediately after the last **U** notification bringing the snapshot has been sent. The snapshot represents the “initial state” of an item, and may be composed of multiple **U** notifications in case the subscription is in `DISTINCT` or `COMMAND` mode. With `MERGE` mode the snapshot is composed of just one **U** notification and hence this notification is not sent. With `RAW` mode the snapshot is not supported.

For an in-depth discussion about the snapshot and subscription modes see the *General Concepts* document.

Notification Example:

```
EOS, 3, 1
```

Snapshot Clearing

👉 **Tag: CS** (short of Clear Snapshot)

👉 **Format:** `CS,<subscription-ID>,<item>`

👉 **Used when:** to notify that the snapshot of an item has been cleared.

👉 Arguments: 2

- ◆ `<subscription-ID>`
The **ID** of the subscription.

◆ `<item>`

Index of the item whose snapshot has been cleared. The index is 1-based, and the order of items in the subscription is determined by the **Metadata Adapter**.

This notification is sent when the Data Adapter explicitly asks the Server to clear the snapshot of a subscription. If the subscription is in `MERGE` mode, no notification is sent, as the clearing is handled by the Server by sending an update with null fields. On the other hand, if the subscription is in `DISTINCT` or `COMMAND` mode, the notification is sent to the client so that it can clear its representation of the item (typically a list or a history, with these modes).

Notification Example:

```
CS,3,1
```

Overflow

👉 **Tag: OV** (short of OVerflow)

👉 **Format:** `OV,<subscription-ID>,<item>,<overflow-size>`

👉 **Used when:** to notify that one or more update events for an item have been dropped due to buffer limitations on the Server.

👉 **Arguments: 3**

◆ `<subscription-ID>`

The **ID** of the subscription.

◆ `<item>`

Index of the item whose update events have been cleared. The index is 1-based, and the order of items in the subscription is determined by the **Metadata Adapter**.

◆ `<overflow-size>`

The number of real-time update events that have been dropped.

This notification can only be sent if the item was subscribed in `RAW` or `COMMAND` mode (for `ADD` and `DELETE` events only), or if it was subscribed in `MERGE`, `DISTINCT`, or `COMMAND` mode (the modes for which events dropping is allowed) and **unfiltered dispatching** was requested. So, in all these cases, whenever the Server has to drop an event because of resource limits, it notifies the client in this way.

Notification Example:

```
OV,3,1,5
```

Subscription Reconfiguration

👉 **Tag: CONF** (short of subscription reCONFiguration)

👉 **Format:** `CONF,<subscription-ID>,<max-frequency>,<filtered|unfiltered>`

👉 **Used when:** to notify that a subscription has been successfully configured, or reconfigured with a different (or even the same) max frequency.

👉 **Arguments: 3**

◆ `<subscription-ID>`

The **ID** of the subscription.

◆ `<max-frequency>`

New maximum frequency of the subscription, expressed in updates/sec as a decimal number. The value `unlimited` is also possible, meaning that no limit is applied on the frequency.

Notes:

- ▶ This value reports the maximum frequency for *all* the items in the subscription, considering that different items may have different maximum frequencies (as the Metadata Adapter can limit the frequency on an item by item basis).
- ▶ The computed maximum frequency may be an approximation of the requested maximum frequency, due to the Server internal handling of update scheduling.

◆ `<filtered|unfiltered>`

Filtered/unfiltered flag for the subscription.

This notification is sent at the start of a subscription and after a subscription has been reconfigured following a Subscription Reconfiguration Request (see [Subscription Reconfiguration Request](#)). After this notification, real-time updates are sent with the specified maximum frequency.

Notification Example:

```
CONF,3,3.0,filtered
```

MPN-Related Notifications

The following notifications provide identifiers and status of MPN Module entities, such as MPN devices and MPN subscriptions. The MPN Module is a component of the Server that can route real-time updates to mobile devices via their native push notification service (e.g. APNs for Apple™ platforms and FCM for Google™ platforms). See Chapter 5 of the *General Concepts* document for an in-depth introduction to the MPN Module.

Edition Note: MPN is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

MPN Device Registered

👉 **Tag:** `MPNREG` (short of MPN REGistration)

👉 **Format:** `MPNREG,<MPN-device-ID>,<MPN-data-adapter-name>`

👉 **Used when:** to notify that an MPN device has been successfully registered.

👉 **Arguments:** 2

◆ `<MPN-device-ID>`

The **ID** of the MPN device. This ID is assigned by the Server and is formatted as a 36-character UUID.

◆ `<MPN-data-adapter-name>`

Name of the Data Adapter providing the status of the MPN device and its MPN subscriptions. The name of the adapter is assigned by the Server and is formatted as an alphanumeric string, compliant with adapter name restrictions.

This notification is sent after an MPN device registration has been requested and accepted by the Server (see [MPN Device Registration](#)). After this notification the MPN device may be addressed for subsequent requests on the current session, such as MPN subscription activation and badge reset.

The second argument is the name of an internal Data Adapter always made available in the session's Adapter Set when the MPN Module is enabled. This Data Adapter supplies special items that provide information for monitoring the status of the MPN device and its MPN subscriptions. See paragraph "The Internal MPN Data Adapter" of the *General Concepts* document for all the information needed.

Notification Example:

```
MPNREG,8c1eaa79-acd1-4cb4-b2a7-3b62bb449798,MPN_INTERNAL_DATA_ADAPTER
```

MPN Device Badge Reset

- 👉 **Tag:** **MPNZERO** (short of MPN device badge set to ZERO)
- 👉 **Format:** **MPNZERO,<MPN-device-ID>**
- 👉 **Used when:** to notify that an MPN device had its badge successfully reset.
- 👉 **Arguments: 1**
 - ◆ **<MPN-device-ID>**
The **ID** of the MPN device.

This notification is sent after the reset of an MPN device badge has been requested and accepted by the Server (see [MPN Device Badge Reset](#)).

Notification Example:

```
MPNZERO,8c1eaa79-acd1-4cb4-b2a7-3b62bb449798
```

MPN Subscription Activated

- 👉 **Tag:** **MPNOK** (short of MPN subscription OK)
- 👉 **Format:** **MPNOK,<subscription-ID>,<MPN-subscription-ID>**
- 👉 **Used when:** to notify that an MPN subscription has been successfully activated.
- 👉 **Arguments: 2**
 - ◆ **<subscription-ID>**
The **ID** of the subscription. This ID is assigned by the client and is a progressive integer starting with 1.
 - ◆ **<MPN-subscription-ID>**
The **ID** of the MPN subscription. This ID is assigned by the Server and is formatted as a 36-characters UUID.

This notification is sent after the an MPN subscription activation has been requested and accepted by the Server (see [MPN Subscription Activation](#)). After this notification, native push notifications start to be sent to the mobile device.

Notification Example:

```
MPNOK, 3, af9fbcf1-7d03-4bb5-9712-59a28d12d22a
```

MPN Subscription Reconfigured

- 👉 **Tag:** **MPNCONF** (short of MPN reCONFigured)
- 👉 **Format:** **MPNCONF**, <MPN-subscription-ID>
- 👉 **Used when:** to notify that a previous request to reconfigure a MPN subscription has been accomplished.
- 👉 **Arguments: 1**
 - ◆ <MPN-subscription-ID>
The ID of the MPN subscription. This ID was specified in the client request.

This notification is sent after the an MPN subscription reconfiguration has been requested and accepted by the Server (see [MPN Subscription ReconfigurationActivation](#)). After this notification, native push notifications will continue with different parameters and, if the MPN subscription was in `TRIGGERED` state, they will be resumed.

Differently from the `CONF` notification for data subscriptions, this notification does not specify the new parameters. For this kind of information you should rely on the dedicated internal Data Adapter, whose name was received upon registration (see [MPN Device Registered](#)).

Notification Example:

```
MPNCONF, , af9fbcf1-7d03-4bb5-9712-59a28d12d22a
```

MPN Subscription Deactivated

- 👉 **Tag:** **MPNDEL** (short of MPN subscription DELETED)
- 👉 **Format:** **MPNDEL**, <MPN-subscription-ID>
- 👉 **Used when:** to notify that an MPN subscription has been successfully activated.
- 👉 **Arguments: 1**
 - ◆ <MPN-subscription-ID>
The ID of the MPN subscription.

This notification is sent after the an MPN subscription deactivation has been requested and accepted by the Server (see [MPN Subscription Deactivation](#)). After this notification, no more native push notifications will be sent to the mobile device.

Notification Example:

```
MPNDEL, af9fbcf1-7d03-4bb5-9712-59a28d12d22a
```

Message-Related Notifications

The following notifications provide insights on the status of a specific upstream message.

Message Successfully Sent

👉 **Tag:** MSGDONE (short of MeSsaGe send DONE)

👉 **Format:** MSGDONE, <sequence>, <prog>, <response>

👉 **Used when:** to notify that a message has been successfully sent and processed.

👉 **Arguments:** 3

◆ <sequence>

Sequence identifier specified in the send request, or “*”, if a sequence had not been specified..

◆ <prog>

Message **progressive number** specified in the send request.

◆ <response>

Response message from the Metadata Adapter. If not supplied (i.e. supplied as null), an empty message is received here.

This notification is sent after the corresponding message has been successfully processed. A message is considered processed when the Metadata Adapter has received it in its `notifyUserMessage` event and returned with no exceptions. See the *Metadata Adapter SDK API Reference* for more information.

Notification Example:

```
MSGDONE,Orders_Sequence,3,Processed with ID 32652506
```

Message Send Failed

👉 **Tag:** MSGFAIL (short of MeSsaGe send FAILED)

👉 **Format:** MSGFAIL, <sequence>, <prog>, <error-code>, <error-message>

👉 **Used when:** to notify that a message has not been sent or its processing failed.

👉 **Arguments:** 4

◆ <sequence>

Sequence identifier specified in the send request, or “*”, if a sequence had not been specified.

◆ <prog>

Message **progressive number** specified in the send request.

◆ <error-code>

Error code sent by the Server kernel or by the Metadata Adapter. For a list of supported error codes see [Appendix B](#).

◆ <error-message>

Error message sent by Server kernel or by the Metadata Adapter. Note that a message sent by the Metadata Adapter may be empty.

This notification is sent when the corresponding message has not been received by the Server or its processing failed for some reason.

Notification Example:

```
MSGFAIL,Orders_Sequence,4,38,The specified progressive number has been skipped by timeout
```

Session-Related Notifications

The following notifications provide insights on the status of the current session.

Session Constraints Changed

👉 **Tag:** **CONS** (short of CONStrain)

👉 **Format:** **CONS**, <bandwidth>

👉 **Used when:** to notify that a session has been successfully configured, or reconfigured with a different (or even the same) maximum bandwidth.

👉 **Arguments:** 1

◆ <bandwidth>

New maximum bandwidth of the session, expressed in kbps as a decimal number. The value `unlimited` is also possible, meaning that no limit is applied on the bandwidth. The value `unmanaged` is possible as well; it is equivalent to `unlimited`, but it also notifies that the client is not allowed to limit the bandwidth for this session.

This notification is sent at the beginning of the session and after the session constraints, in particular its maximum bandwidth, has been changed following a Session Constrain Request (see [Session Constrain Request](#)). The bandwidth may be changed also by server-side actions. After this notification, real-time updates are sent within the specified maximum bandwidth limit.

Notification Example:

```
CONS, 50
```

Time Synchronization

👉 **Tag:** **SYNC** (short of SYNChronism)

👉 **Format:** **SYNC**, <seconds-since-initial-header>

👉 **Used when:** to notify the time elapsed on the Server since the session was bound.

👉 **Arguments:** 1

◆ <seconds-since-initial-header>

Time elapsed on the Server since the session was bound, expressed in seconds.

This notification is sent periodically to notify the client of the time elapsed on the Server. Official Lightstreamer client libraries exploit this notification to detect when they are not keeping up with the current flow of real-time updates. The notifications are not sent on polling connections.

Notification Example:

```
SYNC,120
```

Client IP Address

- 👉 **Tag:** **CLIENTIP** (short of CLIENT IP address)
- 👉 **Format:** **CLIENTIP,<client-IP>**
- 👉 **Used when:** to notify the IP address of the client, as seen by the Server.
- 👉 **Arguments: 1**
 - ◆ **<client-IP>**
IP address of the client, as seen by the Server.

This notification is sent to notify the client of its own IP address, as it appears on the Server. Note that this value, in principle, may be different for each request, even though related to the same Session; hence it is reissued at each bind, unless unchanged. Official Lightstreamer client libraries exploit this notification to detect when the client has changed its network (e.g. it passed from a 3G WAN to a local Wi-Fi), so that it can try again to reconnect with a transport that previously failed. Note that this notification may be disabled with the Server configuration element `<client_identification>`, which should be set consistently on all ports.

Notification Example with IPv4 address:

```
CLIENTIP,127.0.0.1
```

Notification Example with IPv6 address:

```
CLIENTIP,::1
```

Server Name

- 👉 **Tag:** **SERVNAME** (short of SERVer NAME)
- 👉 **Format:** **SERVNAME,<server-name>**
- 👉 **Used when:** to notify the Server configured name.
- 👉 **Arguments: 1**
 - ◆ **<server-name>**
The configured name of the Server.

This notification is sent to let the client know the configured name of the Server (i.e. the “name” attribute specified for the listening port in Lightstreamer Server configuration). Note that this value, in principle, may be different for each request, even though related to the same Session; hence it is reissued at each bind, unless unchanged. Official Lightstreamer client libraries provide this information to the application, for example to improve diagnostics.

Notification Example:

```
SERVNAME,Lightstreamer HTTP Server
```

Progressive of Last Data Notification

- 👉 **Tag:** **PROG** (short of PROGressive)
- 👉 **Format:** `PROG,<progressive>`
- 👉 **Used when:** to notify the starting point of the response flow.
- 👉 **Arguments: 1**
 - ◆ `<progressive>`

The progressive number of the last **data notification** already sent (or, equivalently, the total count of data notifications already sent); the response flow will start from this point.

This notification is sent only when requested via `LS_recovery_from` ON `bind_session` requests. The specified progressive should be the same as the one specified through `LS_recovery_from`, but actually it can be lower. In this case, the initial **data notifications** received must be skipped, until the required progressive number is reached. However, any notification of different kind received must always be obeyed.

Notification Example:

```
PROG,24576
```

No Operation

- 👉 **Tag:** **NOOP** (short of NO OPeration)
 - 👉 **Format:** `NOOP,<preamble>`
 - 👉 **Used when:** to send dummy content to the client.
 - 👉 **Arguments: 1**
 - ◆ `<preamble>`
- Dummy content** to be ignored.

The purpose of this notification is to fill the receive buffer of the client browser or the operating system during the initial setup phase of the session. While this may seem weird, consider that certain operating systems buffer the content of an HTTP response until some specific length (e.g. 2 Kbytes). Preemptively filling this buffer lets the client receive subsequent content as soon as it is sent by the Server.

Notification Example:

```
NOOP,sending placeholder data
```

Probe (Keep-Alive)

- 👉 **Tag:** **PROBE**
- 👉 **Format:** `PROBE`

👉 **Used when:** to keep the stream connection alive.

👉 **Arguments: 0**

This notification is sent periodically by the Server when no other activity has been sent on the stream connection. The interval is specified in the Server configuration, but it may be changed during session creation; the actual interval is reported in the session creation response. The notifications are not sent on polling connections.

Official Lightstreamer client libraries monitor this notification to detect when the connection is stalled: if after the expected interval plus a configurable timeout no `PROBE` has been received, the connection is closed and reopened.

Notification Example:

```
PROBE
```

Stream Connection Loop

👉 **Tag: LOOP**

👉 **Format:** `LOOP,<expected-delay>`

👉 **Used when:** to signal that the session needs to be rebound.

👉 **Arguments: 1**

◆ `<expected-delay>`

Expected delay before rebinding, expressed in milliseconds.

Notes:

- ▶ A value of 0 means that the client should rebind the session as soon as possible.
- ▶ A value greater than 0 is actually used only on polling sessions requested with `LS_polling_millis > 0`, i.e. synchronous polling. In this situation, the `expected-delay` may be lower than originally requested with `LS_polling_millis`.

Reasons for a session to be rebound are mainly the following:

- ▶ The polling cycle is complete.
- ▶ The Content-Length of an HTTP stream connection has been reached.
- ▶ A rebind has been explicitly asked by the client.

In all these situations the session is unbound and a `LOOP` notification is sent. The client is expected to react by rebinding the session (see [Session Binding Request](#)).

Note: since this notification marks the end of the stream connection, with HTTP transport, any spurious character following the CR-LF of this notification may be safely ignored.

Notification Example:

```
LOOP,5000
```

Stream Connection End

👉 **Tag: END**

👉 **Format:** `END,<cause-code>,<cause-message>`

👉 **Used when:** to signal that the session has been closed by the Server.

👉 **Arguments: 1**

◆ `<cause-code>`

Cause code sent by the Server kernel. For a list of supported cause codes see [Appendix A](#).

◆ `<cause-message>`

Short **cause description** sent by the Server kernel.

This notification is sent when the Server closes the session for some reason, e.g. on request by an administrator or as a consequence of a Session Destroy request.

Note: since this notification marks the end of the stream connection, with HTTP transport, any spurious character following the CR-LF of this notification may be safely ignored.

Notification Example:

```
END,8,Session count limit reached
```

4 Special Use Cases

This chapter details some special use cases of TLCP that fall outside of the rules described so far.

Session Creation and Control Combo Request

Under particular circumstances, you may want to create and control a session with a single request. To accomplish this, simply create a request that includes the name for session creation and the parameters of both the session creation and control requests. Note that message send requests are not supported by this syntax.

With HTTP transport, an example of session creation and subscription is the following:

```
POST /lightstreamer/create_session.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 166
Content-Type: text/plain

LS_user=&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20kOj2Bg&LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOT
E_ADAPTER&LS_mode=MERGE
```

With WS transport, the same example is the following:

```
create_session
LS_user=&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20kOj2Bg&LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOT
E_ADAPTER&LS_mode=MERGE
```

Note that:

- ▶ The `LS_session` parameter is omitted from the control request, as the request refers to the session to be created.
- ▶ The `LS_reqId` parameter is also omitted. The control request, in fact, has **no specific response** (see below), so specifying a request ID is useless.
- ▶ The control request does not count towards monitoring statistics.

The response of this combo request is only that of the session creation, but **the whole operation is considered atomic**: if the control request fails, the session creation also fails.

Hence, if the **combo request is successful**, a sample response with **HTTP transport** would be:

```
HTTP/1.1 200 OK
Server: Lightstreamer-Server/7.0.0 build 1972
Content-Type: text/enriched; charset=iso-8859-1
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 1 Jan 1970 00:00:00 GMT
Date: Fri, 1 Jul 2016 14:01:19 GMT
Transfer-Encoding: chunked

2E
CONOK,S73d162c183916f0dT2729905,50000,5000,*
```

While with **WS transport** it would simply be:

```
CONOK,S73d162c183916f0dT2729905,50000,5000,*
```

On the other hand, if the **combo request fails**, either in the session creation or control part, a sample response with **HTTP transport** could be:

```
HTTP/1.1 200 OK
Server: Lightstreamer-Server/7.0.0 build 1972
Content-Type: text/plain; charset=iso-8859-1
Cache-Control: no-store
Cache-Control: no-transform
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 1 Jan 1970 00:00:00 GMT
Date: Fri, 1 Jul 2016 14:01:19 GMT
Content-Length: 130

CONERR,64,Unexpected error while initializing the session: Metadata Provider refusal:
Mode MERGE is not supported for item item1
```

While with **WS transport** it would simply be:

```
CONERR,64,Unexpected error while initializing the session: Metadata Provider refusal:
Mode MERGE is not supported for item item1
```

Session Creation and Multiple Control Combo Request

A variant of the previous special case is a session creation request combined with multiple control requests.

With HTTP transport, an example of session creation and multiple subscription is the following:

```
POST /lightstreamer/create_session.txt?LS_protocol=TLCP-2.5.0 HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
Content-Length: 376
Content-Type: text/plain

LS_user=&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg
LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&
LS_mode=MERGE
LS_op=add&LS_subId=1&LS_group=item2&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&
LS_mode=MERGE
LS_op=add&LS_subId=1&LS_group=item3&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&
LS_mode=MERGE
```

With WS transport, the same example is the following:

```
create_session
LS_user=&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg
LS_op=add&LS_subId=1&LS_group=item1&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&
LS_mode=MERGE
LS_op=add&LS_subId=1&LS_group=item2&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&
LS_mode=MERGE
LS_op=add&LS_subId=1&LS_group=item3&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&
LS_mode=MERGE
```

Note that:

- ▶ As in the previous special case, the `LS_session` and `LS_reqId` parameters are **omitted**.
- ▶ Control requests are separated on **multiple lines**, as in the case of **batching**. See [Request Syntax](#) for more information.
- ▶ These control requests do not count towards monitoring statistics.

In this special case also the session creation and multiple control requests are considered **atomic**: if one the control requests fails, the session creation fails too. Hence, sample successful and failure responses are equivalent to those of the previous special case.

Use of HTTP GET in Place of HTTP POST

While all documentation of TLCP so far specified to use the HTTP POST method, the Server also accepts the HTTP GET method.

When using the GET method the following considerations apply:

- Parameters must be specified in the **query string**, since some HTTP client libraries may not allow to send any request body with a GET method.
 As a consequence, the parameter values must be **encoded as URI components**, according to the requirements of the HTTP specifications. This is consistent with the percent-encoding rules needed by the TLCP protocol, provided that for non-ascii characters the percent-encoding is based on the **UTF-8** character set.
- The query string is **limited in length** (its limit can be set in the Server configuration file).
- With a session creation request, **username and password are visible** on the query string and more prone to be logged by intermediate proxies.
- Since on the Server the HTTP GET method may actually change the Server state, it **can not be considered nullipotent**, and hence it is not compliant with HTTP specification.
 - ◆ Moreover, consider that an intermediate network appliance may wrongly assume it is nullipotent and resend the same request in a subsequent moment.

While in some cases the HTTP GET method may solve practical problems (e.g. when diagnosing a connection problem from the address bar of a browser), its use is strongly discouraged for the reasons above.

An example of session creation with HTTP GET method is the following:

```
GET /lightstreamer/create_session.txt?LS_protocol=TLCP-
2.5.0&LS_user=&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
```

Combo requests, as introduced above, are also supported when single control requests are involved, as in the following example:

```
GET /lightstreamer/create_session.txt?LS_protocol=TLCP-
2.5.0&LS_user=&LS_adapter_set=DEMO&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20kOj2Bg&LS_op=add&LS_subId=1&LS_group=item2&LS_schema=last_price&LS_data_adapter=QUOTE_ADAPTER&LS
_mode=MERGE HTTP/1.1
Host: push.lightstreamer.com
Accept: */*
```

Response is the same as with HTTP POST method. See [Session Creation and Binding Responses](#) for more information.

5 Hands On

This chapter shows the TLCP protocol hands on by using the cURL utility as a mean to connect, send requests and receive responses from a Lightstreamer Server. First two workflows are taken from [Workflow Examples](#), where they are shown with their sequence diagram also. Sample sessions are presented for HTTP transport only, as there is no widely accepted cURL equivalent for WS transport.

About cURL

cURL is a widespread command-line utility used to test HTTP requests and responses. It is available for most operating systems and can be downloaded at:

► <https://curl.haxx.se>

Its basic syntax is:

```
curl [options] <URL>
```

The options used in the following examples are:

👉 -X <method>

Specifies the HTTP method to be used in the request.

👉 -d '<data>'

Specifies the content of the request body.

👉 -N

Specifies to avoid buffering the response.

👉 -v

Specifies to be verbose.

A sample cURL command-line request is the following:

```
curl -X POST -d "LS_adapter_set=DEMO" http://push.lightstreamer.com/lightstreamer/create_session.txt
```

Server Setup

Any just-downloaded installation of Lightstreamer Server can run the following sample sessions. Once started (see the *GETTING_STARTED.TXT* document in the installation directory), the Server:

- 👉 Responds to TCP port 8080 to both HTTP and WS connections.
- 👉 Provides an Adapter Set called “WELCOME” and a Data Adapter called “STOCKS” that simulates a simple stock exchange.
 - ◆ Item names are named “item1” to “item30” and provide stock quotes with different simulated prices, names and update frequencies.
 - ◆ Field names are the following: “stock_name”, “time”, “last_price”, “pct_change”, “min”, “max”, “ask”, “bid”, “bid_quantity”, “ask_quantity”, “ref_price”, “open_price”.

- 👉 Requires no authentication.

To better support the following manual interaction, the temporary relieve of some Server timeouts may be advisable. This involves enlarging the `<session_timeout_millis>` and `<session_recovery_millis>` elements in the Server configuration file, for instance to 100000. Then, a Server restart will be needed.

Basic Subscription

The Basic Subscription example has the following workflow (see [Basic Subscription](#)):

1. Connection and session creation.
2. Subscription to an item on a control connection.
3. Receive some data.
4. Unsubscription from an item on a control connection.
5. Disconnection.

Connection and Session Creation

First things first, let's connect and create the session. Our request needs to specify:

- 👉 The protocol: `LS_protocol=TLCP-2.5.0`
- 👉 The client identifier: `LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg`
- 👉 The Adapter Set "WELCOME": `LS_adapter_set=WELCOME`
- 👉 Finally, the host, port and name: `http://localhost:8080/lightstreamer/create_session.txt`

Here it is the complete call with cURL:

```
curl -v -N -X POST -d "LS_adapter_set=WELCOME&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg"
http://localhost:8080/lightstreamer/create_session.txt?LS_protocol=TLCP-2.5.0
```

Note that it is a **single line**. See [Session Creation Request](#) for more information.

The response from the Server looks like the following:

```
CONOK,S1d7c802482843a26T5626355,50000,5000,*
SERVNAME,Lightstreamer HTTP Server
CLIENTIP,0:0:0:0:0:0:1
NOOP,sending placeholder data
[...]
NOOP,sending placeholder data
CONS,unlimited
PROBE
PROBE
PROBE
[...]
```

The response contains:

- 👉 The `CONOK` response, with the session ID (underlined), request limit, keep alive time and control link.
- 👉 The `SERVNAME` notification, with the name of the Server.
- 👉 The `CLIENTIP` notification, with the client IP.

- 👉 Some `NOOP` notifications (the majority are omitted), needed to fill up the receive buffer of the client.
- 👉 The `CONS` notification, with the maximum bandwidth allowed (`unlimited` means no limit).
- 👉 Finally, repeating each 5 seconds, a `PROBE` notification to keep the stream connection alive.

Subscription to an Item

The previous call with `cURL` opened the stream connection and must be left intact to receive some data. So, on a separate command line, let's subscribe to an item using a request with the following characteristics:

- 👉 The operation is a subscription: `LS_op=add`
- 👉 The subscription ID to identify the subscription later.
- 👉 The Data Adapter "STOCKS": `LS_data_adapter=STOCKS`
- 👉 The group is simply the name of an item: `LS_group=item1`
- 👉 The schema is composed by juxtaposing some field names: `LS_schema=stock_name time last_price`
 - ◆ Note that juxtaposing field names to form a schema name requires some logic on the Metadata Adapter side. The official Adapter SDKs provide this logic in the *LiteralBasedProvider* Metadata Adapter example.
- 👉 The session ID is the one reported on the session creation response.
- 👉 The request ID to be able to match the corresponding response.
- 👉 Finally, the name is that of a control request: `control`

Here it is the complete call with `cURL`:

```
curl -v -N -X POST -d
"LS_op=add&LS_subId=1&LS_data_adapter=STOCKS&LS_group=item1&LS_schema=stock_name time
last_price&LS_mode=MERGE&LS_session=<session-ID>&LS_reqId=1"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined). See [Subscription Request](#) for more information.

The response from the Server is:

```
REQOK, 1
```

With `1` being the request ID. On the other command-line, where the stream connection is running, some new notifications appear:

```
[...]
PROBE
SYNC, 14
SUBOK, 1, 1, 3
CONF, 1, unlimited, filtered
PROBE
[...]
```

The meaning being:

- 👉 The `SYNC` notification reports the elapsed time on the server (in seconds).
- 👉 The `SUBOK` notification reports that the subscription with ID 1 has been successfully activated.
- 👉 That `CONF` notification reports that the subscription with ID 1 has no frequency limit and is filtered.

See [Time Synchronization](#), [Successful Subscription](#) and [Subscription Reconfiguration](#) for more information.

Receive Some Data

On the command-line where the stream connection is running, from time to time some real-time update notifications appear:

```
[...]
PROBE
U,1,1,Anduct|11:02:33|3.11
PROBE
[...]

[...]
PROBE
SYNC,102
U,1,1,|11:03:39|
PROBE
[...]

[...]
PROBE
U,1,1,|11:04:29|3.09
PROBE
[...]
```

The subscribed item changes its values with an average period of 30 seconds. When it changes, the Server sends a real-time update notification with the new data (only those that were specified in the schema).

So the first real-time update:

```
👉 U,1,1,Anduct|11:02:33|3.11
```

Specifies that the first item of subscription with ID 1 has now the following values:

- 👉 "Anduct" as `stock_name`
- 👉 "11:02:33" as `time`
- 👉 3.11 as `last_price`

Following real-time updates provide changes to some of these fields, leaving unspecified all the fields that are unchanged. See [Decoding the Pipe-Separated List of Values](#) for more information.

Unsubscription from an Item

On a separate command line, let's unsubscribe using a request with the following characteristics:

- 👉 The operation is an unsubscription: `LS_op=delete`
- 👉 The subscription ID is the one used during subscription: `LS_subId=1`

- 👉 The session ID is the one reported on the session creation response.
- 👉 The request ID to be able to match the corresponding response.

Here it is the complete call with cURL:

```
curl -v -N -X POST -d "LS_op=delete&LS_subId=1&LS_session=<session-ID>&LS_reqId=2"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined). See [Unsubscription Request](#) for more information.

The response from the Server is:

```
REQOK, 2
```

With 2 being the request ID. On the other command-line, where the stream connection is running, a new notifications appears:

```
UNSUB, 1
```

The meaning of UNSUB notification is that the subscription with ID 1 has been successfully deactivated. See [Successful Unsubscription](#) for more information.

Disconnection

Disconnecting from a stream connection is done by simply closing the underlying socket. The Server automatically detects the connection termination and destroys the session, freeing up its resources. In case of the stream connection with cURL, pressing Ctrl-C is enough.

In particular cases where a network appliance in the middle (such as a reverse proxy) may keep the connection alive, a control request may be sent to forcefully destroy the session on the Server. The request has the following characteristics:

- 👉 The operation is a session destroy: `LS_op=destroy`
- 👉 The session ID is the one reported on the session creation response.
- 👉 The request ID to be able to match the corresponding response.

Here it is the complete call with cURL:

```
curl -v -N -X POST -d "LS_op=destroy&LS_session=<session-ID>&LS_reqId=3"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. See [Session Destroy Request](#) for more information.

The response from the Server is:

```
REQOK, 3
```

with 3 being the request ID. On the other command-line, where the stream connection is running, a new notifications appears:

```
END,31,Destroy invoked by client
```

The meaning of END being that the session has been successfully destroyed. See [Stream Connection End](#) for more information.

Session Rebinding

The Session Rebinding example has the following workflow (see [Session Rebinding](#)):

1. Connection and session creation.
2. Subscription to a first item on a control connection.
3. Subscription to a second item on a control connection.
4. Receive data until the Content-Length is reached.
5. Rebind the session on a new stream connection.

Session Creation and Subscription to Items

Repeating the operations of the previous example, let's open a stream connection using a slightly modified create session request:

```
curl -v -N -X POST -d "LS_adapter_set=WELCOME&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20kOj2Bg&LS_content_length=10000"
http://localhost:8080/lightstreamer/create_session.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. In this request we added a parameter:

👉 The Content-Length is forcefully limited to 10 Kbytes: `LS_content_length=10000`

See [Session Creation Request](#) for more information.

The response from the Server looks like the following:

```
CONOK,Se939a67a9be2d336T3823582,50000,5000,*
SERVNAME,Lightstreamer HTTP Server
CLIENTIP,0:0:0:0:0:0:1
NOOP,sending placeholder data
[...]
NOOP,sending placeholder data
CONS,unlimited
PROBE
PROBE
PROBE
PROBE
PROBE
[...]
```

Recall that the underlined part is your **session ID**.

Let's now subscribe to `item1` and `item2` with two control requests executed on a separate command line:

```
curl -v -N -X POST -d
"LS_op=add&LS_subId=1&LS_data_adapter=STOCKS&LS_group=item1&LS_schema=stock_name time
last_price&LS_mode=MERGE&LS_session=<session-ID>&LS_reqId=1"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0

curl -v -N -X POST -d
"LS_op=add&LS_subId=2&LS_data_adapter=STOCKS&LS_group=item2&LS_schema=stock_name time
last_price&LS_mode=MERGE&LS_session=<session-ID>&LS_reqId=2"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that they are **single lines**. Put the session ID of your session where appropriate (underlined). See [Subscription Request](#) for more information.

The response from the Server is:

```
REQOK, 1
```

for the first subscription, and:

```
REQOK,2
```

for the second. On the other command-line, where the stream connection is running, some new notifications appear:

```
[...]
PROBE
SUBOK,1,1,3
CONF,1,unlimited,filtered
PROBE
[...]
PROBE
SYNC,28
SUBOK,2,1,3
CONF,2,unlimited,filtered
[...]
```

These notifications tell us that the items have been subscribed successfully. See [Time Synchronization](#), [Successful Subscription](#) and [Subscription Reconfiguration](#) for more information.

Receive Data Until the Content-Length Is Reached

Once subscribed, on the command-line where the stream connection is running, a number of real-time updates appear:

```
[...]
U,2,1,Ations Europe|15:55:08|14.81
U,2,1,||14.66
U,2,1,|15:55:09|14.62
U,2,1,||14.71
U,2,1,|15:55:10|14.63
U,2,1,||14.77
U,2,1,|15:55:11|
U,2,1,||14.61
U,1,1,Anduct|15:55:12|3.16
U,2,1,|15:55:12|14.5
U,2,1,|15:55:13|14.64
U,2,1,|15:55:14|
U,2,1,||14.74
U,2,1,||14.66
[...]
```

The subscription to item2, in fact, sends real-time updates with an average period of 500 milliseconds. This is intentional: by using this frequently changing item the stream connection reaches its content-length in a just a few minutes.

Once the stream connection reaches the content-length, the connection terminates with a notification:

```
U,2,1,|15:57:51|16.27
U,2,1,||16.24
U,2,1,||16.11
U,2,1,|15:57:52|15.99
U,2,1,||15.93
LOOP,0
```

The `LOOP` notification tells that the session must be rebound. Its only argument is the expected delay: a `0` here means that the session should be rebound as soon as possible, with no delay.

Rebind the Session on a New Stream Connection

To rebind the session we need a request that specifies:

- 👉 The session ID reported on the session creation response.
- 👉 The name is that of a session rebind request: `bind_session`

Here it is the complete call with cURL:

```
curl -v -N -X POST -d "LS_session=<session-ID>"
http://localhost:8080/lightstreamer/bind_session.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined). See [Session Binding Request](#) for more information.

The Server responds with a new stream connection and immediately starts sending updates for previous subscriptions:

```
CONOK,Se939a67a9be2d336T3823582,50000,5000,*
NOOP,sending placeholder data
[...]
NOOP,sending placeholder data
CONS,unlimited
U,2,1,|15:57:53|17.7
U,2,1,||17.81
U,2,1,|15:57:54|17.72
U,2,1,||17.62
U,2,1,||17.74
[...]
```

Session Recovery

The Session Recovery example has the following workflow (see also [Session Recovery](#)):

1. Connection and session creation.
2. Subscription to an item on a control connection.
3. Receive data.
4. Manually interrupt the connection.
5. Recovery the session on a new stream connection.

Session Creation and Subscription to an Item

Let's open a stream connection first:

```
curl -v -N -X POST -d "LS_adapter_set=WELCOME&LS_cid=mgQkwtwdysogQz2BJ4Ji%20kOj2Bg"
http://localhost:8080/lightstreamer/create_session.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**.

The response from the Server looks like the following:

```
CONOK,S22dee113e3f71b1fT4327493,50000,5000,*
SERVNAME,Lightstreamer HTTP Server
CLIENTIP,0:0:0:0:0:0:1
NOOP,sending placeholder data
[...]
NOOP,sending placeholder data
CONS,unlimited
PROBE
PROBE
PROBE
[...]
```

Recall that the underlined part is your **session ID**.

Let's now subscribe to `item2` with a control request executed on a separate command line:

```
curl -v -N -X POST -d
"LS_op=add&LS_subId=1&LS_data_adapter=STOCKS&LS_group=item2&LS_schema=stock_name time
last_price&LS_mode=MERGE&LS_session=<session-ID>&LS_reqId=1"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined).

The response from the Server is:

```
REQOK, 1
```

Receive Data and Interrupt the Connection

On the other command-line, where the stream connection is running, some new notifications, followed by a number of real-time updates, appear:

```
[...]
PROBE
SUBOK, 1, 1, 3
CONF, 1, unlimited, filtered
PROBE
U, 1, 1, Ations Europe|15:55:08|14.81
U, 1, 1, ||14.66
U, 1, 1, |15:55:09|14.62
U, 1, 1, ||14.71
U, 1, 1, |15:55:10|14.63
U, 1, 1, ||14.77
U, 1, 1, |15:55:11|
U, 1, 1, ||14.61
U, 1, 1, |15:55:12|14.5
U, 1, 1, |15:55:13|14.64
U, 1, 1, |15:55:14|
SYNC, 25
U, 1, 1, ||14.74
U, 1, 1, ||14.66
```

Now, interrupt the connection by just killing the cURL process, but keep its console output, so that you can count the "data notifications" received.

The count should start with the `SUBOK` notification and include the `CONF` and all the `U` notifications. In our example, the count yields 15.

Recover the Session on a New Stream Connection

To rebind the session in order to recover from the interruption point, we need a request that specifies:

- 👉 The session ID reported on the session creation response.
- 👉 The recovery point, that is, the count of data notifications received, as determined above.
- 👉 The name is that of a session rebind request: `bind_session`

In order to better show the recovery mechanism, we suggest actually choosing a recovery point lower than the number of data notifications received (let's say 11, in our example), in order to simulate notifications already sent by the Server but not received. As a consequence, a few notifications will be duplicated.

Here it is the complete call with cURL:

```
curl -v -N -X POST -d "LS_session=<session-ID>&LS_recovery_from=11"
http://localhost:8080/lightstreamer/bind_session.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined).

The Server responds with a new stream connection and immediately starts sending updates for previous subscriptions, specifying the new starting point and starting from there:

```
CONOK, S22dee113e3f71b1fT4327493, 50000, 5000, *
NOOP, sending placeholder data
[...]
NOOP, sending placeholder data
CONS, unlimited
PROG, 11
U, 1, 1, |15:55:13|14.64
U, 1, 1, |15:55:14|
U, 1, 1, ||14.74
U, 1, 1, ||14.66
U, 1, 1, |15:55:15|17.7
U, 1, 1, ||17.81
U, 1, 1, |15:55:16|17.72
U, 1, 1, ||17.62
SYNC, 5
U, 1, 1, ||17.74
[...]
```

Remember that the starting point specified by the Server may also be lower than the requested one, although this is not expected to happen in this simple scenario. This must be taken into account.

Sending And Receiving Messages

The following example has no corresponding diagram in Chapter 2, but its workflow is relatively simple:

1. Connection and session creation.
2. Subscription to an item on a control connection.
3. Send a message.
4. Receive a message echo on the item.

Session Creation and Subscription to an Item

Repeating the operations of the previous example, let's open a stream connection using a create session request:

```
curl -v -N -X POST -d "LS_adapter_set=WELCOME&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20kOj2Bg&LS_send_sync=false" http://localhost:8080/lightstreamer/create_session.txt?
LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. See [Session Creation Request](#) for more information.

By adding "LS_send_sync=false" we will now get rid of the SYNC notifications on the stream connection.

The response from the Server looks like the following:

```
CONOK, Sa5268cc2d401967bT4311553, 50000, 5000, *
SERVNAME, Lightstreamer HTTP Server
CLIENTIP, 0:0:0:0:0:0:1
NOOP, sending placeholder data
[...]
NOOP, sending placeholder data
CONS, unlimited
```

```
PROBE
PROBE
PROBE
[...]
```

Recall that the underlined part is your **session ID**.

The item we are now going to subscribe is not part of the “STOCKS” Data Adapter, but rather the “CHAT” Data Adapter. This adapter provides the following:

- ◆ A single item named “chat_room” that simulates a simple **chat feed**.
- ◆ Field names are the following: “timestamp”, “message”, “IP”, “nick”.
- ◆ For this data model the appropriate subscription mode is “DISTINCT”. For more information on subscription modes see [Subscription Data Model](#).

Let’s now subscribe to `chat_room` with a control request executed on a separate command line:

```
curl -v -N -X POST -d
"LS_op=add&LS_subId=1&LS_data_adapter=CHAT&LS_group=chat_room&LS_schema=timestamp
message&LS_mode=DISTINCT&LS_session=<session-ID>&LS_reqId=1"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined). See [Subscription Request](#) for more information.

The response from the Server is:

```
REQOK,1
```

As usual, on the other command-line, where the stream connection is running, some new notifications appear:

```
[...]
PROBE
SUBOK,1,1,2
CONF,1,unlimited,filtered
PROBE
[...]
```

These notifications tell us that the item has been subscribed successfully. See [Time Synchronization](#), [Successful Subscription](#) and [Subscription Reconfiguration](#) for more information.

Send a Message and Receive its Echo

To send a message we need a request with the following characteristics:

- 👉 The content of the message is just a well-known international salute: `LS_message=CHAT|Ciao`
 - ◆ The “CHAT|” part of the message content is **not part of the protocol**, it’s just a second-level syntax in use by this specific Metadata Adapter. The complete message content is in fact “CHAT|Ciao”, but the adapter will strip out up to the pipe and forward just “Ciao” to the chat feed.
- 👉 The progressive number of the message: `LS_msg_prog=1`
- 👉 The session ID is the one reported on the session creation response.
- 👉 The request ID to be able to match the corresponding response.

- 👉 A sequence to which the message belongs may be added, if the message should be kept in order with others.
- 👉 Finally, the name is that of a message send request: `msg`

Here it is the complete call with cURL:

```
curl -v -N -X POST -d "LS_session=<session-ID>&LS_message=CHAT|
Ciao&LS_msg_prog=1&LS_reqId=2" http://localhost:8080/lightstreamer/msg.txt?
LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined). See [Message Send Request](#) for more information.

The response from the Server is:

```
REQOK, 2
```

In the case of messages, this standard response also acts as an **acknowledge** from the Server, meaning that the message has been correctly received and enqueued for processing.

On the other command-line, where the stream connection is running, some new notifications appear:

```
[...]
PROBE
MSGDONE, *, 1,
U, 1, 1 | 13:07:00 | Ciao
PROBE
[...]
```

The meaning being:

- 👉 The `MSGDONE` notification tells that the message has been processed by the Server and delivered to the Metadata Adapter. Its arguments are the **sequence of the message** (we did not specify it, and hence it is an asterisk “*”), the **progressive number** of the message, and a **response message**, which is empty in this case
- 👉 The **real-time update** reports the message as part of the chat feed, with the two fields being the timestamp and the message content.

The real-time update is, in fact, the echo of our previous message. See [Message Successfully Sent](#) and [Real-Time Update](#) for more information on these notifications.

Mobile Push Notifications

This example shows an interaction with the Mobile Push Notifications (MPN) Module, that enables the Server to route real-time updates via native push notifications to a mobile device.

Edition Note: MPN is an optional feature, available depending on Edition and License Type. To know what features are enabled by your license, please see the License tab of the Monitoring Dashboard (by default, available at /dashboard).

A Note about the terminology used in the example:

- 👉 Where the term “*notification*” is used alone, the meaning is the same as used anywhere else in the document, i.e. messages delivered by the Server on the stream connection and documented extensively in [Chapter 4](#).

- Where the term “*notification*” is used together with “*push*”, such as in “*push notifications*”, “*Mobile Push Notifications*” or even simply the “*N*” in “*MPN*”, we refer to messages delivered by mobile platforms, such as iOS™ and Android™, and displayed automatically on mobile devices.

The example shows the following workflow:

1. Connection and session creation.
2. Registration of an MPN device on a control connection.
3. Activation of an MPN subscription to an item on a control connection.
4. Disconnection and creation of a second session.
5. Deactivation of the MPN subscription on the second session.

The diagram at §5.6 of the *General Concepts* document shows this workflow.

Receiving actual push notifications requires a mobile development infrastructure fully set up, including a mobile device, an identifier for the application, the actual application running on the device and a valid device token.

To simplify things, for this example we use *simulated* push notifications, i.e. push notifications that are just logged, not sent to the platform’s MPN provider. This lets us walk through the example even with no infrastructure in place, and choose any sample string as application identifier and device token.

If you have a mobile development infrastructure already set up, it’s easy to replace sample values with real ones and see actual push notifications delivered to the device.

Additional Configuration

Before proceeding, there are a few steps required to enable the MPN Module on the Server and provide it the necessary information to run.

1st) The MPN Module must be enabled by setting to `Y` the `<enabled>` flag in the `<mpn>` section of the Server configuration file `conf/lightstreamer_conf.xml`:

```
<mpn>
[... ]
<enabled>Y</enabled>
[... ]
</mpn>
```

2nd) An SQL database must be setup and configured appropriately in the Server’s Hibernate configuration file `conf/mpn/hibernate.cfg.xml`.

If you have no SQL database already available, you can quickly setup and run an [HSQLDB](#) instance: a free, open source, single-file SQL server implemented in Java, more than sufficient for testing purposes. Add its jar file (e.g. `hsqldb-2.4.0.jar`) in the `lib/mpn/hibernate` directory.

If using HSQLDB, its sample configuration is already present in the Hibernate configuration file and looks like the following:

```
<session-factory>
[... ]
<property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
<property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
<property name="connection.username">sa</property>
<property name="connection.password"></property>
<property name="dialect">org.hibernate.dialect.HSQLDialect</property>
[... ]
</session-factory>
```

3rd) Add the application that is going to receive push notifications in the Server's MPN provider configuration file of the platform. This step is necessary because, for security purposes, the Server only accepts MPN requests for applications it knows.

In this example we use the following identifiers:

- 👉 As a platform we use iOS™ and its platform identifier is `Apple`, to be used also for all other Apple™ platforms. To use one of Google™ platforms, such as Android™, use `Google`. No other platforms are supported at time of writing.
- 👉 As an application identifier we use `com.mydomain.myapp`. If you have a mobile device with an actual iOS™ application in development, you can use its bundle ID. The corresponding identifier for Android™ is the package name.
- 👉 As a device token we use the sample string `123456`. If you have a mobile device with an actual iOS™ application in development, you can use the device token obtained through the User Notification APIs. The corresponding identifier for Android™ is the registration ID obtained through the Firebase Cloud Messaging APIs.

The Apple™ platforms' MPN provider configuration file is `conf/mpn/apple/apple_notifier_conf.xml` and the application can be added with the following lines:

```
[...]
<app id="com.mydomain.myapp">
<service_level>test</service_level>
</app>
[...]
```

If you have a mobile development infrastructure and want to send actual push notifications, you need to set the service level to something different than `test`, but this requires also a client certificate or a valid API key. Configuration files contain extensive comments on the options available.

4th and last) Set the log level of the `LightstreamerLogger.mpn.apple` logger to `INFO` in the log configuration file `conf/lightstreamer_log_conf.xml`:

```
<logger name="LightstreamerLogger.mpn.apple" level="INFO"/>
```

This makes the payload of push notifications visible on the log before they are sent.

Session Creation

Repeating operations of previous examples, let's open a stream connection using a create session request:

```
curl -v -N -X POST -d "LS_adapter_set=WELCOME&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20k0j2Bg&LS_send_sync=false" http://localhost:8080/lightstreamer/create_session.txt?
LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. See [Session Creation Request](#) for more information.

The response from the Server looks like the following:

```
CONOK,Sec3d8a1deal6eb59T2343461,50000,5000,*
SERVNAME,Lightstreamer HTTP Server
CLIENTIP,0:0:0:0:0:0:1
NOOP,sending placeholder data
[...]
NOOP,sending placeholder data
CONS,unlimited
PROBE
```

```
PROBE
PROBE
[...]
```

Recall that the underlined part is your **session ID**.

MPN Device Registration

Before we can activate an MPN subscription we need to register the mobile device. For this purpose we need a request with the following information:

- 👉 The operation is an MPN device registration: `LS_op=register`
- 👉 The session ID is the one reported on the session creation response.
- 👉 The request ID to be able to match the corresponding response.
- 👉 The device token: `PN_deviceToken=123456`
- 👉 The application ID: `PN_appId=com.mydomain.myapp`
- 👉 The platform name: `PN_type=Apple`
- 👉 Finally, the name is that of a control request: `control`

On a separate command line, register the MPN device with the following call:

```
curl -v -N -X POST -d "LS_session=<session-ID>&LS_reqId=1&LS_op=register&PN_type=Apple&PN_appId=com.mydomain.myapp&PN_deviceToken=123456" http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined). See [MPN Device Registration Request](#) for more information.

The response from the Server is:

```
REQOK, 1
```

On the other command-line, where the stream connection is running, a new notification appears:

```
[...]
PROBE
MPNREG, 28c6ec93-283a-43cd-8d03-2c82ea72510b, MPN_INTERNAL_DATA_ADAPTER
PROBE
[...]
```

The `MPNREG` notification tells us the registration completed successfully. The underlined part is your **MPN device ID**. See [MPN Device Registered](#) for more information on this notification.

The second argument of the `MPNREG` notification is the name of the internal Data Adapter for monitoring the status of the MPN device. We won't enter into these details in the example, as it would complicate it too much, but paragraph "The Internal MPN Data Adapter" of the *General Concepts* document contains all the information needed.

Note: registering the device not only provides the necessary ID for any subsequent MPN request, but it also enables them. In other words, knowing the MPN device ID is not enough to activate an MPN subscription, the device must have also been registered *on the current session*. The Server will refuse requests for an unregistered, although valid, MPN device ID.

MPN Subscription Activation

Now that we have registered the MPN device, we can proceed to activate an MPN subscription. We use the same “STOCKS” adapter used in previous examples, with its items and fields.

For this request we need the following information:

- 👉 The operation is an MPN subscription activation: `LS_op=activate`
- 👉 The subscription ID to identify the subscription later.
- 👉 The Data Adapter “STOCKS”: `LS_data_adapter=STOCKS`
- 👉 The group is simply the name of an item: `LS_group=item1`
- 👉 The schema is composed by juxtaposing two field names: `LS_schema=stock_name last_price`
- 👉 The session ID is the one reported on the session creation response.
- 👉 The request ID to be able to match the corresponding response.
- 👉 The MPN device ID is the one reported during device registration.
- 👉 The push notification format, i.e. the platform-specific JSON structure that must be sent to the MPN provider, is a simple APNs alert with the stock name and price, a sound and badge:


```
PN_notificationFormat={"aps":{"badge":1,"sound":"Default","alert":"Price of ${stock_name} now is ${last_price}"}}
```

 - ◆ Note that the push notification format contains references to subscription fields, such as `${stock_name}` and `${last_price}`. These references are replaced with values from the real-time update before the push notification is sent to the MPN provider.
- 👉 Finally, the name is that of a control request: `control`

Here is the complete request, to be run on a separate command line:

```
curl -v -N -X POST -d "LS_session=<session-ID>&LS_reqId=2&LS_subId=1&LS_op=activate&LS_data_adapter=STOCKS&LS_group=item1&LS_schema=stock_name last_price&LS_mode=MERGE&PN_deviceId=<MPN-device-ID>&PN_notificationFormat={\"aps\":{\"badge\":1,\"sound\": \"Default\", \"alert\": \"Price of ${stock_name} now is ${last_price}\"}}&PN_coalescing=true" http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session and the MPN device ID obtained before where appropriate (underlined). See [MPN Subscription Activation Request](#) for more information.

The response from the Server is:

```
REQOK, 2
```

On the other command-line, where the stream connection is running, a new notification appears:

```
[...]
PROBE
MPNOK, 1, 518443d9-b8a5-459d-ae23-caaa62b4da7a
PROBE
[...]
```

The `MPNOK` notification reports the **subscription ID** specified during the request and its corresponding **MPN subscription ID**. These two identifiers must not be confused:

- The *subscription ID* is the transient, client-assigned progressive integer specified with the argument `LS_subId`. Like the same parameter for real-time subscriptions, its scope is limited to the current session.
- The *MPN subscription ID* is a persistent, Server-assigned 36-characters UUID. Its scope is beyond the session life-cycle and its use is precisely to identify an MPN subscription within the current or any subsequent session.

See [MPN Subscription Activated](#) for more information on this notification.

Within about 30 seconds from the activation, the item1 of the “STOCKS” adapter fires a real-time update and the log file in `logs/Lightstreamer.log` reports a line that looks like this:

```
17-Nov-17 12:55:27,763|INFO |L.m.a.c.m.myapp |main.myapp POOLED THREAD 1|Payload for
subscription 518443d9-b8a5-459d-ae23-caaa62b4da7a with notification ID 1: {"aps":
{"badge":1,"sound":"Default","alert":"Price of Anduct now is 2.97"}}
```

The log line contains a reference to the MPN subscription ID, as well as a progressive number used to identify each push notification. The line also contains the full payload, with field references replaced with actual values.

If you used a real application ID and a real device token, the push notification is now displayed on your mobile device.

Disconnection and Reconnection

The goal of this step of the example is to show that MPN subscriptions survive the session and can be accessed at any later time.

Thus, on a separate command line run the following call to destroy the current session:

```
curl -v -N -X POST -d "LS_op=destroy&LS_session=<session-ID>&LS_reqId=3"
http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. The response from the Server is:

```
REQOK,3
```

As seen in previous examples, the stream connection terminates with a final notification:

```
END,31,Destroy invoked by client
```

Remain on the stream connection command line and run the following call to open a new session:

```
curl -v -N -X POST -d "LS_adapter_set=WELCOME&LS_cid=mgQkwtwdysogQz2BJ4Ji
%20kOj2Bg&LS_send_sync=false" http://localhost:8080/lightstreamer/create_session.txt?
LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. The response from the Server, as usual, looks like the following:

```
CONOK,S239e009d15f976d4T2418362,50000,5000,*
SERVNAME,Lightstreamer HTTP Server
CLIENTIP,0:0:0:0:0:0:1
NOOP,sending placeholder data
[...]
NOOP,sending placeholder data
CONS,unlimited
PROBE
PROBE
PROBE
[...]
```

Take note of the new **session ID**.

Finally, on a separate command line, we have to register again the MPN device to enable MPN requests on the session. Run the following call:

```
curl -v -N -X POST -d "LS_session=<session-ID>&LS_reqId=1&LS_op=register&PN_type=Apple&PN_appId=com.mydomain.myapp&PN_deviceToken=123456" http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session where appropriate (underlined).

The response from the Server is:

```
REQOK, 1
```

On the other command-line, where the stream connection is running, a new notification appears:

```
[...]
PROBE
MPNREG, 28c6ec93-283a-43cd-8d03-2c82ea72510b, MPN_INTERNAL_DATA_ADAPTER
PROBE
[...]
```

Note that the **MPN device ID** reported by the `MPNREG` notification is identical to the initial registration.

MPN Subscription Deactivation

Now that we have registered the MPN device on the new session, we can proceed to deactivate the initial MPN subscription.

The request includes the following information:

- 👉 The operation is an MPN subscription deactivation: `LS_op=deactivate`
- 👉 The session ID is the one reported on the latest session creation response.
- 👉 The request ID to be able to match the corresponding response.
- 👉 The MPN device ID is the one reported during MPN device registration.
- 👉 The MPN subscription ID is the one obtained during MPN subscription activation.
- 👉 Finally, the name is that of a control request: `control`

Here is the complete request, to be run on a separate command line:

```
curl -v -N -X POST -d "LS_session=<session-ID>&LS_reqId=2&LS_op=deactivate&PN_deviceId=<MPN-device-ID>&PN_subscriptionId=<MPN-subscription-ID>" http://localhost:8080/lightstreamer/control.txt?LS_protocol=TLCP-2.5.0
```

Recall that it is a **single line**. Put the session ID of your session, the MPN device ID and the MPN subscription ID where appropriate (underlined). See [MPN Subscription Deactivation Request](#) for more information.

The response from the Server is:

```
REQOK, 2
```

On the other command-line, where the stream connection is running, a new notification appears:

```
[...]  
PROBE  
MPNDEL, 518443d9-b8a5-459d-ae23-caaa62b4da7a  
PROBE  
[...]
```

The `MPNDEL` notification reports that the MPN subscription has been deleted. See [MPN Subscription Deactivated](#) for more information on this notification.

Appendix A: Session Error Codes

The following error codes may be used with session error responses such as `CONERR`, or by session notifications such as `END`:

👉 1

User/password check failed.

👉 2

Requested Adapter Set not available.

👉 3

The session specified on a `bind_session` request was initiated with a different and incompatible communication protocol or protocol version.

👉 4

The progressive specified on a `bind_session` request to recovery from has not been kept by the Server, hence the recovery attempt failed.

👉 5

The Server is temporarily overloaded and cannot accept session creation requests: retry later.

👉 6

The Metadata Adapter is temporarily unable to validate the user credentials upon a session creation request: retry later.

👉 7

Licensed maximum number of actively started sessions reached (this can only happen with some licenses).

👉 8

Configured maximum number of actively started sessions reached.

👉 9

Configured maximum server load reached on a `create_session` request.

👉 10

New sessions temporarily blocked.

👉 11

Streaming is not available because of Server license restrictions (this can only happen with special licenses).

👉 20

Specified session not found on a `bind_session` request.

👉 21

The session ID specified in a `bind_session` request is not compatible with this Server instance and it is not likely to pertain to an instance just closed; it might pertain to a different instance.

31

The session was closed (possibly by the administrator) through a client `destroy` request.

32

The session was closed on the Server side, via software or by the administrator.

33, 34

An unexpected error occurred on the Server while the session was in activity.

35

The Metadata Adapter only allows a limited number of sessions 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 this session has been performed by some client.

48

The maximum session duration has been reached, according to Server configuration or as enforced by the administrator. This is only meant as a way to refresh the session (for instance, to force a different association in a clustering scenario), hence the client should recover by opening a new session immediately.

60

Client version not supported with the current Server.

64

Generic error in the control request part of a session/control combo request.

65

Syntax error or invalid values specified for one or more arguments.

66

Unchecked exception thrown in the Metadata Adapter while managing a `create_session` request.

67

Malformed request.

68

An unexpected error occurred on the Server while managing the request.

69

Streaming request on a WebSocket not allowed, as a streaming is already in place.

👉 70

Session creation request not allowed on this Server port.

👉 71

Client type not supported with the current Server.

👉 ≤ 0

If the code is 0 or negative, it has been supplied by the **Metadata Adapter** and the interpretation is application-specific. In particular if used with an `END` notification, means the session was closed by an Administrator with a *destroy* command and the code was supplied as a custom cause code.

Appendix B: Control Error Codes

The following cause codes may be used with control error responses such as `REQERR` and `ERROR`:

3

The session specified was initiated with a different communication protocol or protocol version, which is incompatible with this request.

11

The specified session ID is not compatible with this Server instance and it is not likely to pertain to an instance just closed; it might pertain to a different instance.

13

Subscription frequency reconfiguration not allowed because the subscription is configured for unfiltered dispatching.

15

Subscription to an item in `COMMAND` mode with no “key” field in the schema.

16

Subscription to an item in `COMMAND` mode with no “command” field in the schema.

17

Bad Data Adapter name or default Data Adapter not defined.

19

Specified subscription not found.

20

Specified (or implied) session not found.

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.

29

RAW mode is not allowed by the current license terms.

30

Subscriptions are not allowed by the current license terms (for special licenses only).

32

The specified message 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 message progressive number is too low: a message with this number has already been enqueued (and possibly processed).

34

An unexpected error occurred on the Server while managing the message (for instance, a NotificationException was thrown by the Metadata Adapter).

35

Unchecked exception thrown in the Metadata Adapter while managing the message.

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.

40

The MPN Module is disabled, either by configuration or by license restrictions.

41

The MPN request failed because of some internal resource error (e.g. database connection, timeout etc.).

42

The specified MPN platform is invalid or not supported.

43

The specified MPN application ID is invalid or unknown.

44

The syntax used in the specified MPN trigger expression is invalid.

45

The specified MPN device ID is invalid or unknown.

46

The specified MPN subscription ID is invalid or unknown.

47

The specified MPN trigger expression or notification format contains an invalid argument name.

48

The MPN device corresponding to the specified ID has been suspended.

49

One or more arguments of the MPN request exceed its maximum size.

50

The specified MPN subscription contains no items or no fields.

51

The payload of the MPN subscription exceeds the maximum size for the platform.

52

The specified MPN notification format is not a valid JSON structure.

53

The specified MPN notification format is empty.

56

The subscription parameters specified don't match with the specified MPN subscription.

65

Syntax error or invalid values specified for one or more arguments.

66

Unchecked exception thrown in the Metadata Adapter while managing a subscription or reconfiguration request.

👉 67

Malformed request.

👉 68

An unexpected error occurred on the Server while managing the request.

👉 ≤ 0

If the code is 0 or negative, it has been supplied by the **Metadata Adapter** and the interpretation is application-specific.

This can affect subscription requests and message processing.

Appendix C: Server HTTP Headers

The following tables report Lightstreamer Server use of common HTTP headers for both requests and responses.

HTTP Request Headers

General Header Fields

Header	Use
Cache-Control	Ignored
Keep-Alive	Ignored (obsolete)
Date	Ignored
Pragma	Ignored
Trailer	Ignored
Transfer-Encoding	Handled (only the <code>chunked</code> case)
Via	Ignored
Warning	Ignored

Request Header Fields

Header	Use
Accept	Ignored
Accept-Charset	Ignored
Accept-Encoding	Handled (only the <code>gzip</code> case)
Accept-Language	Ignored
From	Ignored
If-Match	Ignored
If-None-Match	Ignored
If-Range	Ignored
If-Unmodified-Since	Ignored
Range	Ignored
TE	Ignored

Entity Header Fields

Header	Use
Content-Base	Ignored (obsolete)
Content-Encoding	Handled (not all cases)
Content-Language	Ignored
Content-Location	Ignored
Content-MD5	Ignored
Content-Range	Partially handled: if specified the request is refused
Content-Type	Partially handled: if not compatible the request is processed anyway
Content-Type charset	Handled: relies to charset availability in the JVM
Expires	Ignored
Last-Modified	Ignored
Extension headers	Ignored

HTTP Response Headers

General Header Fields

Header	Use
Cache-Control	Used
Pragma	Used
Trailer	Not used
Transfer-Encoding	Used
Transfer-Encoding chunk-extension	Not used
Transfer-Encoding trailer	Not used
Warning	Not used

Response Header Fields

Header	Use
Accept-Ranges	Not used
Age	Not used
ETag	Not used
Location	Not used
Proxy-Authenticate	Not used
Public	Not used (obsolete)
Retry-After	Not used
Vary	Not used

Entity Header Fields

Header	Use
Allow	Not used
Content-Base	Not used (obsolete)
Content-Encoding	Used
Content-Language	Not used
Content-Location	Not used
Content-MD5	Not used
Content-Range	Not used
Content-Type	Used
Expires	Used
Extension headers	Not used

Appendix D: The custom “TLCP-diff” format

The “diff” format used in association with the τ label, called TLCP-diff, consists in a UTF-8 character sequence that is meant to be applied to a predetermined, base UTF-8 character sequence in order to obtain a result character sequence.

Note that “TLCP-diff” values, when included in a TLCP update message, may be **percent-encoded**; here we refer to the decoded form.

Moreover, “TLCP-diff” values included in a TLCP update message obey the restriction that they don’t include any character which representation in UTF-16 requires a surrogate pair. Likewise, when a “TLCP-diff” value is included in a TLCP update message, it is ensured that the base value on which it is meant to be applied also obeys the same restriction. This simplifies the application of the “diff” in Java and possibly other languages that internally represent characters in UTF-16.

A character sequence in “TLCP-diff” format is the concatenation of one or more sections, each of which represents an instruction and has one of the following types: COPY, ADD, and DEL. The sequence is such that:

- ▶ The first section is of COPY type.
- ▶ Each non-terminal COPY section is followed by an ADD section.
- ▶ Each non-terminal ADD section is followed by a DEL section.
- ▶ Each non-terminal DEL section is followed by a COPY section.

The different sections have the following form:

- ▶ A COPY section consists in an encoded integer number (possibly 0).
- ▶ An ADD section consists in an encoded integer number (possibly 0) followed by that number of characters.
- ▶ A DEL section consists in an encoded integer number (possibly 0).

An encoded integer number is a sequence of one or more letters, such that the last one is a small letter whereas any other preceding ones are capital letters.

This letter sequence forms a radix-26 representation of the number, in which the letters weight progressively. In practice, after converting the letters to lowercase for clarity, each letter weights as its ASCII value minus the ASCII value of ‘a’.

This encoding allows parsing the “diff” character sequence into a list of instructions of the form COPY(N), ADD(N, STR), and DEL(N) in a sequential way.

The following are examples of TLCP-diff values, where we show how they can be separated in sections and their interpretation in terms of lists of instructions.

Value	Sections	List of instructions
d	d	COPY(3)
bdzap	b – dzap	COPY(1) ADD(3,zap)
bdzapcd	b – dzap – c – d	COPY(1) ADD(3,zap) DEL(2) COPY(3)
adzapad	a – dzap – a – d	COPY(0) ADD(3,zap) DEL(0) COPY(3)
AdacAa	Ad – a – c – Aa	COPY(29) ADD(0,) DEL(2) COPY(26)

The list of instructions can be applied to the previous value (*base*) to yield the new value (*result*) with the following process:

- 👉 set *result* as an empty character sequence

- set *basePos* as 0
- For each instruction in the list:
 - ◆ if COPY(N):
 - ▶ append to *result* N characters taken from base starting at position *basePos*
 - ▶ increment *basePos* by N
 - ◆ if ADD(N, STR):
 - ▶ append STR to *result*
 - ◆ If DEL(N):
 - ▶ increment *basePos* by N
- return result

Note that for some instructions N may be 0, hence they are no-ops.

The following are examples of applications of lists of instructions obtained from TLCP-diff values.

Base value	List of instructions	Final value
foo	COPY(3)	foo
foobar	COPY(3)	foo
foobar	COPY(1) ADD(3,zap)	fzap
foobar	COPY(1) ADD(3,zap) DEL(2) COPY(3)	fzapbar
foobar	COPY(0) ADD(3,zap) DEL(0) COPY(3)	zapfoo
foobar	COPY(2) ADD(0,) DEL(2) COPY(2)	foar