Clustering for Load Balancing and Fail Over

Target: Lightstreamer Server v. 7.1 or greater
Last updated: 7/8/2020

Table of contents

1 Introduction

Clustering of Lightstreamer Servers is achieved through any standard Load Balancer (LB), including physical appliances by Cisco, F5, etc., and cloud-based balancers such as Amazon AWS ELB. Software-based load balancer are usually less effective, because they tend to introduce intermediate buffers, which hide the actual status of the TCP connection between the client and the server (Lightstreamer Server monitors such status in order to throttle bandwidth and manage network congestions).

In any case, the LB should be configured in such a way that data packets are forwarded to the clients immediately, without any buffering. Buffering the response will make the client switch from streaming mode to polling mode.

The Lightstreamer Clients can use two different Layer-7 protocols to communicate with Lightstreamer Server: **HTTP** and **WebSocket**. In case the WebSocket protocol is not supported by the LB, Lightstreamer automatically falls back to HTTP. The LB should always be configured to make HTTP work properly. Being able to support WebSockets too is highly recommended, but not strictly required. If you cannot configure the LB to support WebSockets, then, upon the client attempt to use WebSockets, the connection may become unresponsive; in this case, the automatic fallback action may take a few seconds. To avoid the delay, it may be preferable to disable WebSocket support directly on the Server, by setting to `N` the `<enabled>` element within the `<websocket>` block in lightstreamer_conf.xml.

The Lightstreamer Clients create two categories of connections:

- **S**: *The session creation connection*, which is the first connection in the lifecycle of a Lightstreamer client session.
  Through this interaction, the client authenticates and gets its client-id. Depending on the client implementation, this connection may be a short lived HTTP POST request, or it may be long lived and implement the actual streaming channel, over either HTTP or WebSocket.
  Upon unexpected session termination (usually due to a disconnection), another "S" type connection is done by the client.
  These connections can always be freely balanced (with no stickiness requirements).

- **CR**: *The control and rebind connections*, which are used for sending subscriptions and other commands and for implementing the streaming channel in many cases.
  These connections can be based on either HTTP or WebSocket.
  These connections require to be sticky to the server that has served the corresponding session creation connection.

In case of WebSocket, subscriptions and other commands can be issued on the same WebSocket that is hosting the streaming channel. Doing this obviously ensures the stickiness, but it is not always possible.
Therefore, the cluster configuration has to ensure the stickiness property.

Usually, the LB should be configured to switch traffic at Layer 7 (HTTP). But there are cases (especially with WebSockets) where configuring traffic switch at Layer 4 (TCP) enhances compatibility. Configuring LS at Layer 4 usually has some consequences:

- Cookie-based stickiness is not supported.
- X-Forwarded headers are not supported. Such HTTP headers are inserted by the LB to preserve the actual IP address, protocol, and port of the originating client. But a good alternative exists at Layer 4: the Proxy Protocol[1]. It is supported by Lightstreamer Server and allows to preserve the originating client address even without the HTTP headers. Furthermore,

[1] http://www.haproxy.org/download/1.5/doc/proxy-protocol.txt

in some cases a Layer-4 acts as a real low-level switch, so that the originating IP might be kept directly.

Browser-based clients may pose additional constraints depending on the kind of deployment and on the target browsers. Ignoring these constraints may lead to less than optimal connections on some browsers. In the introduction page of the Web Client SDK's API docs there are a few remarks on how to ensure that an "optimal connection" is established. Note that in all cases (as we will see below) in which the `<control_link_address>` setting is leveraged, the Server can be addressed through different hostnames, so the constraints should be considered with regard to all hostnames involved.

The Lightstreamer Clients are divided into two different groups:

- Clients implementing the **Unified Client API**: as can be inferred by the name, the Unified API clients share the same interfaces, abstractions, and behavior, while being available for very different platforms.

- Clients not implementing the Unified Client API: these are legacy libraries having different characteristics and interfaces.

This document will specify when some configuration pertains specifically to clients based on the Unified API.

To illustrate the available options for the cluster configurations, let's suppose that the Lightstreamer Server has been deployed on two boxes (so that the cluster contains two nodes). All the different solutions explained in the following chapters can be naturally extended to the cases where more nodes are used. Fictitious public and private IP addresses and hostnames will be used in the examples.

2 HTTP-Based Scenarios

2.1 Leverage LB Stickiness – Options 1.A

The “**sticky**” property of the Load Balancer is leveraged. Usually Load Balancers offer at least two types of stickiness mechanisms, which leads to two distinct options:

2.1.1 Option 1.A.1 – IP-Address Based

The Balancer routes the requests based on the source IP address. After a request from a new IP address is received, all the subsequent requests from the same IP address are routed to the same node chosen for the first request.
Note: Please bear in mind that some Internet providers exist that dynamically change the source IP address throughout different requests coming from the same Client. So the IP-address based mechanism is not always reliable, making the “cookie-based” one preferable.

The network configuration is like the following one:

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S, CR	push.mycompany.com	Load Balancer	200.0.0.10	80	LS Server 1 or LS Server 2 (balancing + sticky)	10.0.0.1 or 10.0.0.2	80

The lightstreamer_conf.xml file will be the same for both Servers and will contain the following element:

```
<http_server name="HTTP Server">
   <port>80</port>
</http_server>
```

No clustering configuration through the `<control_link_address>` element is necessary.

WebSocket: If the Balancer does not support WebSocket, it is suggested to configure it at Layer 4 (TCP), rather than at Layer 7 (HTTP).

2.1.2 Option 1.A.2 – Cookie Based

The Balancer must support a “Sticky Cookie-Insert” mode, that means that a suitable session cookie is inserted by the Balancer in the responses. Subsequent requests from the Client will send the cookie back, so that the Balancer will be able to correctly route them.

The network configuration is like the following one:

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S, CR	push.mycompany.com	Load Balancer	200.0.0.10	80	LS Server 1 or LS Server 2 (balancing + sticky)	10.0.0.1 or 10.0.0.2	80

The lightstreamer_conf.xml file will be the same for both Servers and will contain the following element:

```
<http_server name="HTTP Server">
   <port>80</port>
</http_server>
```

No clustering configuration through the `<control_link_address>` element is necessary.

In order for the technique of inserting the “Sticky Cookie” to be successful, it is required that the client application manages cookies according with RFC 6265. In this way, session cookies set by the Load Balancer will be sent back by the client upon any further connection to the same domain.
This is normal behaviour for browsers and all applications running in a browser context can take advantage of that. For standalone applications, this property may have to be enforced by the SDK library or even at application level.

For some clients (currently Web and Node.js) and depending on the environment, the following setting may need to be added to the client initialization code, before connecting:

```
lsClient.connectionOptions.setCookieHandlingRequired (true);
```

This has the effect of disabling transports for which cookie handling is not available.
See the remarks on how to ensure that an “optimal connection” is established in the introduction page of the Web Client SDK’s API docs.

For clients implementing the Unified Client API in which the optional `setEarlyWSOpenEnabled` setting is available, the following should also be added to the client initialization code, before connecting:

```
lsClient.connectionOptions.setEarlyWSOpenEnabled(false);
```

However, this is also the default in the most recent versions of these APIs.

WebSocket: As this option works at Layer 7 (Layer 4 is impossible because it does not allow cookie insertion), it is required that the Balancer supports WebSocket.
Note that WebSocket is not currently used by some non-browser client SDKs, which rely exclusively on HTTP(S).

Resume: The table below resumes the requirements for various cases, based on the different Lightstreamer Client SDK and execution context.

SDK	cookie handling	notes
Web (Unified API)	OK	Managed by the browser; Remember to issue setCookieHandlingRequired(true)
Node.js (Unified API)	OK	Managed by LS library; Remember to issue setCookieHandlingRequired(true)

Java SE (Unified API)	OK	Managed by LS library
Android (Unified API)	OK	Managed by LS library
.NET Standard (Unified API)	OK	Managed by LS library
Silverlight	OK	Managed by the browser
Unity	OK	Managed by LS library
Flash	OK	Managed by the browser
Flex	OK	Managed by the browser
iOS (Unified API)	OK	Managed by the system
macOS (Unified API)	OK	Managed by the system
tvOS (Unified API)	OK	Managed by the system
watchOS (Unified API)	OK	Managed by the system
Java ME	TO BE ENFORCED	Just issue: java.net.CookieHandler.setDefault(new java.net.CookieManager(null, java.net.CookiePolicy.ACCEPT_ALL));
Blackberry	TO BE ENFORCED	Just issue: java.net.CookieHandler.setDefault(new java.net.CookieManager(null, java.net.CookiePolicy.ACCEPT_ALL));

2.2 Leverage Application Stickiness – Options 1.B

With these options, stickiness is provided by application-level mechanisms, rather than leveraging the stickiness mechanisms offered by the Load Balancer.

As we will show, this involves the setting of the `<control_link_address>` Server configuration element. This setting specifies a hostname that addresses the Server instance directly. As such, it should be different for each instance and from the Balancer hostname. This allows the instance reached through the "S" connection to notify itself to the client, which, in turn, will issue the "CR" connections directly to that instance.

Because of this mechanism, during the life of a Session, different connections pertaining to the Session may be issued to different hostnames and take different routes. In particular, the "S" connection will be issued to the Load Balancer, whereas any subsequent "CR" connection will be issued to the "control link" and may or may not pass through the Balancer, depending on the server-side architecture (see also the examples below). As a consequence, the streaming flow may also take different routes.
The latter depends on client-side policies. On this matter, we can distinguish three kinds of clients:

- **NewSS**: Web / Node.js clients since SDK version 8.1.
  These clients perform a "Stream-Sense" procedure by trying, as a first attempt, to open the streaming directly on the "S" connection on a WebSocket. If this succeeds (which should be the most common case), then they may never need to open a "CR" connection to the "control link". In this scenario, the main reason for opening a "CR" connection is to recover from an unresponsive connection. If this happens, the streaming flow will migrate to the "control link".

- **BaseSS**: Other clients featuring the "Stream-Sense" algorithm.
  These clients use the "S" connection to perform a pre-flight request in HTTP to open the Session. Then, they use a new "CR" connection for the streaming. Hence, all the streaming

flow comes from the “control link”.

- **NoSS**: Clients from legacy SDKs or very old versions of other SDKs.
  These clients always try to open the streaming in HTTP on the initial “S” connection. However in case of long polling or in case of rebind, they open “CR” connections. If this happens, the subsequent streaming flow will come from the “control link”.

The possibility to address “S” and “CR” connections to different hostnames also allows for the possibility to receive them on different listening ports on the Server instances. This is not necessary in the scenarios analyzed below, where both possibilities (i.e. single port or two ports) can be configured. However, the two-ports configuration is strongly recommended, particularly in presence of **BaseSS** clients, because it enables the use of the `<port_type>` configuration setting on the ports, which, in turn, leverages the backpressure features provided by the Server.
In the examples below, we will always show and suggest the two-ports configuration.

For clients implementing the Unified Client API in which the optional `setEarlyWSOpenEnabled` setting is available (and only for startup performance reasons), the following should also be added to the client initialization code, before connecting:

```
lsClient.connectionOptions.setEarlyWSOpenEnabled(false);
```

However, this is also the default in the most recent versions of these APIs.

2.2.1 Option 1.B.1 – Servers Behind the Load Balancer

The final nodes can be directly addressed by the Client through dedicated hostnames, which the Load Balancer uses to determine the Lightstreamer Server box to forward the request to.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	80	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	81
CR	push1.mycompany.com	Load Balancer	200.0.0.10	80	LS Server 1	10.0.0.1	80
CR	push2.mycompany.com	Load Balancer	200.0.0.10	80	LS Server 2	10.0.0.2	80

The first column of the table above shows the types of connection that will reach the target box.
The two instances of the Lightstreamer Server do not need to have any public IP addresses, because name-based virtual hosting is used in the Load Balancer. All of the requests will reach the Load Balancer, which must adopt a routing algorithm based on the hostname with which the request is received (L7 or HTTP switching). If the host name is “push”, it will use the balancing algorithm (e.g. round robin). If the host name is “push1”, it will send the request to Server 1. If the host name is “push2”, it will send the request to Server 2.

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
</http_server>
```

```
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

As anticipated, the separation between ports 80 and 81 is optional; a single S+CR port can be used, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.
Notice that a solution based on the use of three different VIPs on the Load Balancer (rather than doing L7 switching) is also possible.

WebSocket: Name-based virtual hosting works at Layer 7 (Layer 4 is impossible because it does not allow inspection of the Host HTTP header), hence it requires that the Balancer supports WebSocket. If the Balancer does not support WebSocket, it is still possible to configure it at Layer 4 (TCP), rather than at Layer 7 (HTTP) and use IP-based virtual hosting.

Note on Multihosting: Only in case of a browser-based client, a multihosting environment may lead to some browsers using less than optimal connections to the Lightstreamer servers. See the remarks on how to ensure that an "optimal connection" is established in the introduction page of the Web Client SDK's API docs.

2.2.2 Option 1.B.2 – Servers Partially Behind the Load Balancer

The final nodes can be directly accessed through public IP addresses assigned to the Lightstreamer Server boxes.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	80	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	81
CR	push1.mycompany.com	LS Server1	200.0.0.1	80	-	-	-
CR	push2.mycompany.com	LS Server2	200.0.0.2	80	-	-	-

The two instances of the Lightstreamer Server should be directly reachable from the Internet (i.e. each of them has a public IP address), besides being reachable through the Load Balancer. Here, we prefer to access the Server instances from the Load Balancer on a local port. We emphasize this in our sample configuration by leveraging the possibility to specify different listening interfaces for the two listening ports.

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
   <listening_interface>10.0.0.1</listening_interface>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
   <listening_interface>200.0.0.1</listening_interface>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:
```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
   <listening_interface>10.0.0.1</listening_interface>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
   <listening_interface>200.0.0.1</listening_interface>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

As anticipated, the separation between ports 80 and 81 is optional; a S+CR port 80, accessible on both public and private address, can be used, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.

WebSocket: This option is particularly useful If the Balancer does not support WebSocket. Only the "S" connection (which is always HTTP-based) passes through the Balancer, while the "CR" connections (which can be upgraded to WS) go straight to the Lightstreamer Servers.

3 HTTPS-Based Scenarios

With HTTPS connections, the Balancer cannot see the content of the application HTTP request, because it is encrypted and only the final server can decode it.
In practice, the LB must switch traffic at Layer 4 (TCP), with all related restrictions; so, for example, Option 1.B.1 is not possible.

However, as will be shown in some of the options below, it is also possible to use an **SSL Acceleration** (a.k.a. **SSL offloading**) module, a hardware component that off-loads the encryption/decryption operations from the Lightstreamer Server and then uses plain HTTP to communicate with the Lightstreamer Server.
In practice, an off-loading LB switches traffic at Layer 7 (HTTP), with all related limitations. So, please consider that sometimes SSL offload/acceleration products introduce proxy connections at some level. This could have a double impact on Lightstreamer's features: 1) If the SSL module buffers the full response, before forwarding it to the client, the client will automatically switch from streaming mode to polling mode. 2) The connection that the Lightstreamer Server monitors is not the actual client connection, but is the connection with the SSL Accelerator; if the Accelerator introduces a TCP buffer, the adaptive streaming algorithms of Lightstreamer could take a longer time to detect a network congestion.

But HTTPS has a further complication with respect to HTTP, as the TLS/SSL handshake requires that the server sends a SSL Certificate consistent with the hostname used by the client to access the server. Since the TLS/SSL handshake is usually performed before the application protocol (HTTP) can start[2], the server has no way of determining the hostname used (usually reported in the "Host" HTTP header) and must assume it in advance.
In case the same server should be accessed with two or more different hostnames, it can't but issue a certificate which is consistent with all such names. This is possible by using Wildcard SSL Certificates or Multi-Domain SSL Certificates.
When such a Multi-Domain certificate cannot be obtained, it is still possible to stick to multiple Single-Domain certificates and leverage different ports. We will show this technique as well, whereas both instances of Lightstreamer Server and off-loading Balancers may be affected.

3.1 Leverage LB Stickiness – Options 2.A

The "**sticky**" property of the Load Balancer is leveraged. Usually Load Balancers offer at least two types of stickiness mechanisms, which leads to two distinct options.

3.1.1 Option 2.A.1 – IP-Address Based

The Balancer routes the requests based on the source IP address. After a request from a new IP address is received, all the subsequent requests from the same IP address are routed to the same node chosen for the first request. The same considerations as in section 2.1.1 apply from a theoretical point of view (potentially making this option unreliable), but client IP switching does not usually happen with HTTPS.

A simple SSL certificate for hostname "push.mycompany.com" is needed.

Conn	Full Hostname	Reached	Public IP	Public	Final Box	Final IP	Final

[2] See:
- http://en.wikipedia.org/wiki/Secure_Sockets_Layer#Support_for_name-based_virtual_servers
- https://www.switch.ch/pki/meetings/2007-01/namebased_ssl_virtualhosts.pdf

		Box	Address	TCP Port		Address	Port
S, CR	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing + sticky)	10.0.0.1 or 10.0.0.2	443

The lightstreamer_conf.xml file will be the same for both Servers and will contain the following element:

```
<https_server name="HTTPS Server">
   <port>443</port>
   <keystore>
      <keystore_file>push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
```

No clustering configuration through the `<control_link_address>` element is necessary.

WebSocket: If the Balancer does not support WebSocket, it is possible to configure it at level-4 (TCP), rather than at level-7 (HTTP). This is already the case if SSL offloading is leveraged.

3.1.1.1 Example of Multihosting

By using a Multi-Domain SSL Certificate (see section 3.2.1 below for details) the configuration remains similar to the Option 2.A.1 scenario.

If only two Single-Domain SSL Certificates are available, we must extend the configuration in order to allow each Server instance to provide different certificates based on the hostname with which it is accessed.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S, CR	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing + sticky)	10.0.0.1 or 10.0.0.2	443
S, CR	push.ourcompany.com	Load Balancer	200.0.0.20	443	LS Server 1 or LS Server 2 (balancing + sticky)	10.0.0.1 or 10.0.0.2	444

The Lightstreamer_conf.xml file should contain the following elements, for both the Lightstreamer Server 1 and Lightstreamer Server 2:

```
<https_server name="My HTTPS Server">
   <port>443</port>
   <keystore>
      <keystore_file>mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
```

```
<https_server name="Our HTTPS Server">
   <port>444</port>
   <keystore>
      <keystore_file>ourpush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
```

Similar considerations apply to all the other cases in which individual SSL certificates have to be provided directly by Lightstreamer Server.

Only in case of a browser-based client, a multihosting environment may lead to some browsers using less than optimal connections to the Lightstreamer servers. See the remarks on how to ensure that an "optimal connection" is established in the introduction page of the Web Client SDK's API docs.

3.1.2 Option 2.A.2 – Cookie Based

The Balancer must support a "Sticky Cookie-Insert" mode, provided that the Load Balancer is extended with an **SSL Acceleration** module (otherwise the encrypted cookies would not be writeable and readable by the Balancer). The same considerations as in section 2.1.2 apply.

A simple SSL certificate for hostname "push.mycompany.com" is needed for the off-loading Balancer.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S, CR	push.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing + sticky)	10.0.0.1 or 10.0.0.2	80

The lightstreamer_conf.xml file will be the same for both Servers and will contain the following element:

```
<http_server name="HTTP Server">
   <port>80</port>
</http_server>
```

No clustering configuration through the `<control_link_address>` element is necessary.

For browser-based clients, depending on the environment, the following setting may need to be added to the client initialization code, before connecting:

```
lsClient.connectionOptions.setCookieHandlingRequired(true);
```

See the remarks on how to ensure that an "optimal connection" is established in the introduction page of the Web Client SDK's API docs.

For clients implementing the Unified Client API in which the optional `setEarlyWSOpenEnabled` setting is available, the following should also be added to the client initialization code, before connecting:

```
lsClient.connectionOptions.setEarlyWSOpenEnabled(false);
```

However, this is also the default in the most recent versions of these APIs.

For non-browser clients, see the additional notes provided in section 2.1.2.

WebSocket: As this option works at Layer 7 (Layer 4 is impossible because it does not allow cookie insertion), it is required that the Balancer supports WebSocket.

3.2 Leverage Application Stickiness – Options 2.B

With these options, stickiness is provided by application-level mechanisms, rather than leveraging the stickiness mechanisms offered by the Load Balancer.

As we will show, this involves the setting of the `<control_link_address>` Server configuration element. This setting specifies a hostname that addresses the Server instance directly. As such, it should be different for each instance and from the Balancer hostname. This allows the instance reached through the "S" connection to notify itself to the client, which, in turn, will issue the "CR" connections directly to that instance.

Because of this mechanism, during the life of a Session, different connections pertaining to the Session may be issued to different hostnames and take different routes. In particular, the "S" connection will be issued to the Load Balancer, whereas any subsequent "CR" connection will be issued to the "control link" and may or may not pass through the Balancer, depending on the server-side architecture (see also the examples below). As a consequence, the streaming flow may also take different routes.
The latter depends on client-side policies. On this matter, we can distinguish three kinds of clients:

- **NewSS**: Web / Node.js clients since SDK version 8.1.
  These clients perform a "Stream-Sense" procedure by trying, as a first attempt, to open the streaming directly on the "S" connection on a WebSocket. If this succeeds (which should be the most common case), then they may never need to open a "CR" connection to the "control link". In this scenario, the main reason for opening a "CR" connection is to recover from an unresponsive connection. If this happens, the streaming flow will migrate to the "control link".

- **BaseSS**: Other clients featuring the "Stream-Sense" algorithm.
  These clients use the "S" connection to perform a pre-flight request in HTTP to open the Session. Then, they use a new "CR" connection for the streaming. Hence, all the streaming flow comes from the "control link".

- **NoSS**: Clients from legacy SDKs or very old versions of other SDKs.
  These clients always try to open the streaming in HTTP on the initial "S" connection. However in case of long polling or in case of rebind, they open "CR" connections. If this happens, the subsequent streaming flow will come from the "control link".

These client policies also determine how the TLS handshakes are distributed among the two endpoints. In case of Servers "partially behind an SSL Accelerator", as described below, this also determines how the burden due to TLS handshakes is distributed between the Load Balancer and the Server instances.
As you can infer from the above description, for **NewSS** clients the connections opened toward the "control link" should be fewer than those opened toward the Load Balancer, whereas for the other clients, they should be at least as many (note that **NoSS** clients need to reach the "control link" for control requests).

The possibility to address "S" and "CR" connections to different hostnames also allows for the possibility to receive them on different listening ports on the Server instances. This is necessary in some of the scenarios analyzed below; in other scenarios, both possibilities (i.e. single port or two ports) can be configured. However, the two-ports configuration is strongly recommended in all cases,

because it enables the use of the `<port_type>` configuration setting on the ports, which, in turn, leverages the backpressure features provided by the Server.
In the examples below, we will always show and suggest the two-ports configuration.

Note that a consequence of the two-ports configuration is that the TLS handshakes for “S” and “CR” connections are performed on different ports, so it is important that the TLS configurations of the two ports are the same; or, at least, the configuration of the “CR” port should not be more restrictive than the configuration of the “S” port, to prevent that, for some clients, a loop of successful session establishments followed by bind failures builds up.
In principle, this is easy to achieve, but it may not be that easy when the Server is “partially behind an SSL Accelerator”, as described below. In this case, port “S” is handled by the accelerator while port “CR” is handled by Lightstreamer Server, so care should be taken to ensure that the two TSL configurations are consistent.

For clients implementing the Unified Client API in which the optional `setEarlyWSOpenEnabled` setting is available (and only for startup performance reasons), the following should also be added to the client initialization code, before connecting:

```
        lsClient.connectionOptions.setEarlyWSOpenEnabled(false);
```

However, this is also the default in the most recent versions of these APIs.

3.2.1 Leverage Multi-Domain SSL Certificates – Options 2.B.1

A Wildcard SSL Certificate helps enable SSL encryption on multiple sub-domains using a single certificate as long as the domains are controlled by the same organization and share the same second-level domain name. For example, a certificate issued for name “***.mycompany.com**” may be used to secure all the following domains: “push.mycompany.com”, “push1.mycompany.com”, “push2.mycompany.com”.

The use of wildcards is restricted to domains that share the same second-level domain name. However, the Subject Alternative Name (SAN) field of a SSL Certificate allows for the specification of multiple independent domain names, as long as they are controlled by the same organization. The only drawback is that they must be determined in advance (instead, with a Wildcard Certificate, any newly created sub-domain is automatically allowed).
Multiple domain names can be specified with a syntax of the SAN field like: “**dns:push.mycompany.com,dns:push.ourcompany.com**”.

The two above techniques can be combined: a Wildcard Certificate issued with the SAN “**dns:*.mycompany.com,dns:*.ourcompany.com**” may be used to secure all the following domains: “push1.mycompany.com”, “push2.mycompany.com”, “push1.ourcompany.com”, “push2.ourcompany.com”.

The use of Wildcard Certificates and the SAN field is addressed in RFC 2818[3], section 3.1. Browser compatibility is quite high, where typically only older micro-browsers do not support wildcard certificates or SAN[4]. For more information on wildcard certificates and the SAN field, including possible limitation, you may visit the specific pages of some well-known certification authorities sites[5].

[3] http://www.ietf.org/rfc/rfc2818.txt

[4] For example, see:
- http://www.geocerts.com/ssl/browsers
- https://www.digicert.com/subject-alternative-name-compatibility.htm

[5] For example, see:
- http://www.verisign.com/ssl-certificates/wildcard-ssl-certificates/
- https://www.thawte.com/ssl-digital-certificates/wildcardssl/index.html

3.2.1.1 Option 2.B.1.1 – Servers Behind the Load Balancer

A Wildcard SSL Certificate is issued for “***.mycompany.com**”.
Alternatively, an SSL Certificate with the SAN “**dns:push.mycompany.com,dns:push1.mycompany.com,dns:push2.mycompany.com**” could be issued.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	444
CR	push1.mycompany.com	Load Balancer	200.0.0.1	443	LS Server 1	10.0.0.1	443
CR	push2.mycompany.com	Load Balancer	200.0.0.2	443	LS Server 2	10.0.0.2	443

The Load Balancer needs to be exposed on multiple IP addresses (in this example three) in order to distinguish the final destination of the request, because it cannot read the encrypted hostname that is part of the HTTPS request (however it is necessary to use multiple hostnames, because the Web Client always needs to access the Server with a logical name and not an IP address, due to security restraints).

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<https_server name="HTTPS Server S">
   <port>444</port>
   <port_type>CREATE_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:
```
<https_server name="HTTPS Server S">
   <port>444</port>
   <port_type>CREATE_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="HTTPS Server CR">
```

```
      <port>443</port>
      <port_type>CONTROL_ONLY</port_type>
      <keystore>
         <keystore_file>wildcard_push.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <control_link_address>push2.mycompany.com</control_link_address>
```

The separation between ports 443 and 444 is optional here; a single S+CR port can be used, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.

WebSocket: As this option works at Layer 4 (Layer 7 is impossible because SSL offloading is not being used), WebSocket should always work with no issues.

3.2.1.1.1 Example of Multihosting

As already pointed out in section 3.1.1.1, only in case of a browser-based client a multihosting environment may lead to some browsers using less than optimal connections to the Lightstreamer servers. See the remarks on how to ensure that an “optimal connection” is established in the introduction page of the Web Client SDK’s API docs.

In this example, Option 2.B.1.1 is used to implement a multihosting scenario, where the same Lightstreamer Servers are used to serve clients connected to different domains (e.g. “mycompany.com” and “ourcompany.com”).

By using a single Wildcard SSL Certificate with the SAN “**dns:*.mycompany.com,dns:*.ourcompany.com**”, the configuration remains similar to the Option 2.B.1.1 scenario.

Let’s examine the case in which Wildcard SSL Certificates are available but different second-level domains cannot be combined in a single certificate.
Hence, Two Wildcard SSL Certificates for, respectively “***.mycompany.com**” and “***.ourcompany.com**” are needed for the two different domains.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	445
CR	push1.mycompany.com	Load Balancer	200.0.0.1	443	LS Server 1	10.0.0.1	443
CR	push2.mycompany.com	Load Balancer	200.0.0.2	443	LS Server 2	10.0.0.2	443
S	push.ourcompany.com	Load Balancer	200.0.0.20	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	446
CR	push1.ourcompany.com	Load Balancer	200.0.0.11	443	LS Server 1	10.0.0.1	444
CR	push2.ourcompany.com	Load Balancer	200.0.0.12	443	LS Server 2	10.0.0.2	444

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
<https_server name="My HTTPS Server S">
   <port>445</port>
   <port_type>CREATE_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="My HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="Our HTTPS Server S">
   <port>446</port>
   <port_type>CREATE_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_ourpush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="Our HTTPS Server CR">
   <port>444</port>
   <port_type>CONTROL_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_ourpush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
<https_server name="My HTTPS Server S">
   <port>445</port>
   <port_type>CREATE_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="My HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <keystore>
      <keystore_file>wildcard_mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="Our HTTPS Server S">
   <port>446</port>
   <port_type>CREATE_ONLY</port_type>
```

```
      <keystore>
         <keystore_file>wildcard_ourpush.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <https_server name="Our HTTPS Server CR">
      <port>444</port>
      <port_type>CONTROL_ONLY</port_type>
      <keystore>
         <keystore_file>wildcard_ourpush.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <control_link_address>push2.mycompany.com</control_link_address>
```

The separations between ports 443 and 445 and between ports 444 and 446 ars optional here; single S+CR ports can be used in the two cases, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.

3.2.1.2 Option 2.B.1.2 – Servers Partially Behind the Load Balancer

A Wildcard SSL Certificate is issued for "***.mycompany.com**", to be used for both the two Server instances and the off-loading Balancer.
Alternatively, an SSL Certificate with the SAN "**dns:push.mycompany.com,dns:push1.mycompany.com,dns:push2.mycompany.com**" could be issued.

The final nodes can be directly accessed through public IP addresses assigned to the Lightstreamer Server boxes.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	444
CR	push1.mycompany.com	LS Server1	200.0.0.1	443	-	-	-
CR	push2.mycompany.com	LS Server2	200.0.0.2	443	-	-	-

The two instances of the Lightstreamer Server should be directly reachable from the Internet (i.e. each of them has a public IP address), besides being reachable through the Load Balancer. Here, we prefer to access the Server instances from the Load Balancer on a local port. We emphasize this in our sample configuration by leveraging the possibility to specify different listening interfaces for the two listening ports.

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
   <https_server name="HTTPS Server S">
      <port>444</port>
      <port_type>CREATE_ONLY</port_type>
      <listening_interface>10.0.0.1</listening_interface>
      <keystore>
         <keystore_file>wildcard_push.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
```

```
     </https_server>
     <https_server name="HTTPS Server CR">
        <port>443</port>
        <port_type>CONTROL_ONLY</port_type>
        <listening_interface>200.0.0.1</listening_interface>
        <keystore>
           <keystore_file>wildcard_push.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
     <https_server name="HTTPS Server S">
        <port>444</port>
        <port_type>CREATE_ONLY</port_type>
        <listening_interface>10.0.0.1</listening_interface>
        <keystore>
           <keystore_file>wildcard_push.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <https_server name="HTTPS Server CR">
        <port>443</port>
        <port_type>CONTROL_ONLY</port_type>
        <listening_interface>200.0.0.1</listening_interface>
        <keystore>
           <keystore_file>wildcard_push.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <control_link_address>push2.mycompany.com</control_link_address>
```

The separation between ports 443 and 444 is optional here; a S+CR port 443, accessible from both private and public address, can be used, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.

WebSocket: Only the “S” connection (which is always HTTPS-based) passes through the Balancer, while the “CR” connections (which can be upgraded to WSS) go straight to the Lightstreamer Servers. Furthermore, this option works at Layer 4 (Layer 7 is impossible because SSL offloading is not being used). So, WebSocket should always work with no issues.

3.2.1.3 Option 2.B.1.3 – Servers Behind an SSL Accelerator

An external SSL Accelerator is used to offload the SSL operations, though the limitations outlined at the beginning of section 3 should be taken into consideration.

The Load Balancer is extended with an **SSL Acceleration** module, that takes care of the encryption/decryption operations and uses HTTP to communicate with the final cluster nodes.
A Wildcard SSL Certificate is issued for “***.mycompany.com**”, to be used on the off-loading Balancer. Alternatively, an SSL Certificate with the SAN “**dns:push.mycompany.com,dns:push1.mycompany.com,dns:push2.mycompany.com**” could be issued.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	81
CR	push1.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 1	10.0.0.1	80
CR	push2.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 2	10.0.0.2	80

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:
```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

The separation between ports 80 and 81 is optional here; a single S+CR port can be used, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.

WebSocket: If the Balancer does not support WebSocket, configuring it at Layer 4 (TCP) can in some cases make WebSocket work, provided that the Balancer supports Layer-4 SSL offloading.

3.2.1.4 Option 2.B.1.4 – Servers Partially Behind an SSL Accelerator

A Wildcard SSL Certificate is issued for "***.mycompany.com**", to be used for both the two Server instances and the off-loading Balancer.
Alternatively, an SSL Certificate with the SAN "**dns:push.mycompany.com,dns:push1.mycompany.com,dns:push2.mycompany.com**" could be issued.

The final nodes can be directly accessed through public IP addresses assigned to the Lightstreamer Server boxes.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	80
CR	push1.mycompany.com	LS Server 1	200.0.0.1	443	-	-	-
CR	push2.mycompany.com	LS Server 2	200.0.0.2	443	-	-	-

The two instances of the Lightstreamer Server listen to HTTP requests on a private-IP socket AND to HTTPS requests on a public-IP socket. We emphasize this in our sample configuration by leveraging the possibility to specify different listening interfaces for the two listening ports.
The Balancer balances the incoming “S” connections, while the “CR” connections are directly sent to the final Lightstreamer Servers.

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<http_server name="HTTP Server S">
   <port>80</port>
   <port_type>CREATE_ONLY</port_type>
   <listening_interface>10.0.0.1</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <listening_interface>200.0.0.1</listening_interface>
   <keystore>
      <keystore_file>wildcard_push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<control_link_address>push1.mycompany.com</control_link_address>
<prestarted_max_queue>1000</prestarted_max_queue>
```

On Lightstreamer Server 2:
```
<http_server name="HTTP Server S">
   <port>80</port>
   <port_type>CREATE_ONLY</port_type>
   <listening_interface>10.0.0.2</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <listening_interface>200.0.0.2</listening_interface>
   <keystore>
      <keystore_file>wildcard_push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<control_link_address>push2.mycompany.com</control_link_address>
<prestarted_max_queue>1000</prestarted_max_queue>
```

The `<prestarted_max_queue>` setting is recommended here, in order to leverage the backpressure features provided by the Server. In fact, in this particular configuration, the “CR” connections, being the only ones in TLS/SSL, may be slower than the “S” connections in the establishment phase. This may pose problems, in particular, with **BaseSS** clients (as defined above), whereas, in case of massive client arrival, this may originate a queue of session establishment requests waiting for the final *rebind connection*, with growing session establishment latencies. The `<prestarted_max_queue>` setting puts a limit on this queue; the best value should be determined experimentally.

WebSocket: This option is particularly useful If the Balancer does not support WebSocket. Only the "S" connection (which is always HTTPS-based) passes through the Balancer, while the "CR" connections (which can be upgraded to WSS) go straight to the Lightstreamer Servers.

3.2.2 Leverage Individual Certificates – Options 2.B.2

If only Single-Domain SSL Certificates are available, individual certificates should be issued for each hostname used in the cluster. The configurations shown for the previous options 2.B.1 have to be extended as follows.

3.2.2.1 Option 2.B.2.1 – Servers Behind the Load Balancer

Two different SSL certificates (for two different hostnames) need to be handled by each node of the Lightstreamer Server, which needs to create two server sockets associated to two different keystores. For example, the server socket on port 444 will handle the "push.mycompany.com" certificate, while the server socket on port 443 will handle the "push1.mycompany.com" certificate. A total of three certificates are needed in this example.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	444
CR	push1.mycompany.com	Load Balancer	200.0.0.1	443	LS Server 1	10.0.0.1	443
CR	push2.mycompany.com	Load Balancer	200.0.0.2	443	LS Server 2	10.0.0.2	443

The Load Balancer needs to be exposed on multiple IP addresses (in this example three) in order to distinguish the final destination of the request, because it cannot read the encrypted hostname that is part of the HTTPS request (however it is necessary to use multiple hostnames, because the Web Client always needs to access the Server with a logical name and not an IP address, due to security restraints).

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<https_server name="HTTPS Server S">
   <port>444</port>
   <port_type>CREATE_ONLY</port_type>
   <keystore>
      <keystore_file>push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
```

```
         <keystore>
            <keystore_file>push1.keystore</keystore_file>
            <keystore_password>mypassword</keystore_password>
         </keystore>
      </https_server>
      <control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
      <https_server name="HTTPS Server S">
         <port>444</port>
         <port_type>CREATE_ONLY</port_type>
         <keystore>
            <keystore_file>push.keystore</keystore_file>
            <keystore_password>mypassword</keystore_password>
         </keystore>
      </https_server>
      <https_server name="HTTPS Server CR">
         <port>443</port>
         <port_type>CONTROL_ONLY</port_type>
         <keystore>
            <keystore_file>push2.keystore</keystore_file>
            <keystore_password>mypassword</keystore_password>
         </keystore>
      </https_server>
      <control_link_address>push2.mycompany.com</control_link_address>
```

WebSocket: As this option works at Layer 4 (Layer 7 is impossible because SSL offloading is not being used), WebSocket should always work with no issues.

3.2.2.1.1 Example of Multihosting

As already pointed out in sections 3.1.1.1 and 3.2.1.1.1, only in case of a browser-based client, a multihosting environment may lead to some browsers using less than optimal connections to the Lightstreamer servers. See the remarks on how to ensure that an “optimal connection” is established in the introduction page of the Web Client SDK’s API docs.

In this example, Option 2.B.2.1 is used to implement a multihosting scenario, where the same Lightstreamer Servers are used to serve clients connected to different domains (e.g. “mycompany.com” and “ourcompany.com”).

Six SSL certificates are needed for the six different hostnames involved.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	445
CR	push1.mycompany.com	Load Balancer	200.0.0.1	443	LS Server 1	10.0.0.1	443
CR	push2.mycompany.com	Load Balancer	200.0.0.2	443	LS Server 2	10.0.0.2	443
S	push.ourcompany.com	Load Balancer	200.0.0.20	443	LS Server 1 or LS Server 2	10.0.0.1 or 10.0.0.2	446

					(balancing)		
CR	push1.ourcompany.com	Load Balancer	200.0.0.11	443	LS Server 1	10.0.0.1	444
CR	push2.ourcompany.com	Load Balancer	200.0.0.12	443	LS Server 2	10.0.0.2	444

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
     <https_server name="My HTTPS Server S">
        <port>445</port>
        <port_type>CREATE_ONLY</port_type>
        <keystore>
           <keystore_file>mypush.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <https_server name="My HTTPS Server CR">
        <port>443</port>
        <port_type>CONTROL_ONLY</port_type>
        <keystore>
           <keystore_file>mypush1.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <https_server name="Our HTTPS Server S">
        <port>446</port>
        <port_type>CREATE_ONLY</port_type>
        <keystore>
           <keystore_file>ourpush.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <https_server name="Our HTTPS Server CR">
        <port>444</port>
        <port_type>CONTROL_ONLY</port_type>
        <keystore>
           <keystore_file>ourpush1.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:
```
     <https_server name="My HTTPS Server S">
        <port>445</port>
        <port_type>CREATE_ONLY</port_type>
        <keystore>
           <keystore_file>mypush.keystore</keystore_file>
           <keystore_password>mypassword</keystore_password>
        </keystore>
     </https_server>
     <https_server name="My HTTPS Server CR">
        <port>443</port>
        <port_type>CONTROL_ONLY</port_type>
        <keystore>
           <keystore_file>mypush2.keystore</keystore_file>
```

```
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <https_server name="Our HTTPS Server S">
      <port>446</port>
      <port_type>CREATE_ONLY</port_type>
      <keystore>
         <keystore_file>ourpush.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <https_server name="Our HTTPS Server CR">
      <port>444</port>
      <port_type>CONTROL_ONLY</port_type>
      <keystore>
         <keystore_file>ourpush2.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <control_link_address>push2.mycompany.com</control_link_address>
```

3.2.2.2 Option 2.B.2.2 – Servers Partially Behind the Load Balancer

The two server sockets of the Lightstreamer Server, associated to two different keystores, should be bound to different (or the same) ports of two different IP addresses. We emphasize this in our sample configuration by leveraging the possibility to specify different listening interfaces for the two listening ports. For example, the server socket on 10.0.0.1:444 (private IP address) will handle the "push.mycompany.com" certificate, while the server socket on 200.0.0.1:443 (public IP address) will handle the "push1.mycompany.com" certificate.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	444
CR	push1.mycompany.com	LS Server1	200.0.0.1	443	-	-	-
CR	push2.mycompany.com	LS Server2	200.0.0.2	443	-	-	-

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
   <https_server name="HTTPS Server S">
      <port>444</port>
      <port_type>CREATE_ONLY</port_type>
      <listening_interface>10.0.0.1</listening_interface>
      <keystore>
         <keystore_file>push.keystore</keystore_file>
         <keystore_password>mypassword</keystore_password>
      </keystore>
   </https_server>
   <https_server name="HTTPS Server CR">
      <port>443</port>
      <port_type>CONTROL_ONLY</port_type>
      <listening_interface>200.0.0.1</listening_interface>
      <keystore>
```

```
            <keystore_file>push1.keystore</keystore_file>
            <keystore_password>mypassword</keystore_password>
         </keystore>
      </https_server>
      <control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
      <https_server name="HTTPS Server S">
         <port>444</port>
         <port_type>CREATE_ONLY</port_type>
         <listening_interface>10.0.0.2</listening_interface>
         <keystore>
            <keystore_file>push.keystore</keystore_file>
            <keystore_password>mypassword</keystore_password>
         </keystore>
      </https_server>
      <https_server name="HTTPS Server CR">
         <port>443</port>
         <port_type>CONTROL_ONLY</port_type>
         <listening_interface>200.0.0.2</listening_interface>
         <keystore>
            <keystore_file>push2.keystore</keystore_file>
            <keystore_password>mypassword</keystore_password>
         </keystore>
      </https_server>
      <control_link_address>push2.mycompany.com</control_link_address>
```

WebSocket: Only the “S” connection (which is always HTTPS-based) passes through the Balancer, while the “CR” connections (which can be upgraded to WSS) go straight to the Lightstreamer Servers. Furthermore, this option works at Layer 4 (Layer 7 is impossible because SSL offloading is not being used). So, WebSocket should always work with no issues.

3.2.2.3 Option 2.B.2.3 – Servers Behind an SSL Accelerator

An external SSL Accelerator can be used to offload the SSL operations, though the limitations outlined at the beginning of section 3 should be taken into consideration.

The Load Balancer is extended with an **SSL Acceleration** module, that takes care of the encryption/decryption operations and uses HTTP to communicate with the final cluster nodes. In order for this option to work, the SSL Accelerator must be able to manage different SSL certificates based on the IP address on which it is reached.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	81
CR	push1.mycompany.com	Load Balancer + SSL Accel.	200.0.0.1	443	LS Server 1	10.0.0.1	80
CR	push2.mycompany.com	Load Balancer + SSL Accel.	200.0.0.2	443	LS Server 2	10.0.0.2	80

The two instances of the Lightstreamer Server do not need to have any public IP addresses and they are configured to listen only on an HTTP socket. The Load Balancer is exposed to the Clients through three different IP addresses. The Balancer mounts three SSL certificates (associated to the three different hostnames) and chooses the certificate to use based on the target IP address of the Client request. So, all of the requests will reach the Load Balancer, which must adopt a routing algorithm based on the target IP address or –same result– on the hostname with which the request is received (the hostname is discovered after the request has been decrypted). If the target IP is 200.0.0.10, or the hostname is “push”, the Balancer will use the balancing algorithm (e.g. round robin). If the target IP is 200.0.0.1, or the hostname is “push1”, it will send the request to Server 1. If the target IP is 200.0.0.2, or the hostname is “push2”, it will send the request to Server 2.

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
<http_server name="HTTP Server S">
   <port>81</port>
   <port_type>CREATE_ONLY</port_type>
</http_server>
<http_server name="HTTP Server CR">
   <port>80</port>
   <port_type>CONTROL_ONLY</port_type>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

The separation between ports 80 and 81 is optional here; a single S+CR port can be used, although this would prevent the possibility to specify the port type and take advantage of the associated benefits.

WebSocket: If the Balancer does not support WebSocket, configuring it at Layer 4 (TCP) can in some cases make WebSocket work, provided that the Balancer supports Layer-4 SSL offloading.

3.2.2.4 Option 2.B.2.4 – Servers Partially Behind an SSL Accelerator

The final nodes can be directly accessed through public IP addresses assigned to the Lightstreamer Server boxes.

Conn	Full Hostname	Reached Box	Public IP Address	Public TCP Port	Final Box	Final IP Address	Final Port
S	push.mycompany.com	Load Balancer + SSL Accel.	200.0.0.10	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.1 or 10.0.0.2	80
CR	push1.mycompany.com	LS Server 1	200.0.0.1	443	-	-	-
CR	push2.mycompany.com	LS Server 2	200.0.0.2	443	-	-	-

The two instances of the Lightstreamer Server listen to HTTP requests on a private-IP socket AND to HTTPS requests on a public-IP socket. We emphasize this in our sample configuration by leveraging the possibility to specify different listening interfaces for the two listening ports.
The Load Balancer mounts the SSL certificate associated to “push.mycompany.com“. The two Lightstreamer Servers mount the certificates relative to “push1.mycompany.com“ and “push2.mycompany.com“. The Balancer balances the incoming “S” connections, while the “CR” connections are directly sent to the final Lightstreamer Servers.

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:
```
<http_server name="HTTP Server S">
   <port>80</port>
   <port_type>CREATE_ONLY</port_type>
   <listening_interface>10.0.0.1</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <listening_interface>200.0.0.1</listening_interface>
   <keystore>
      <keystore_file>push1.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<control_link_address>push1.mycompany.com</control_link_address>
<prestarted_max_queue>1000</prestarted_max_queue>
```

On Lightstreamer Server 2:
```
<http_server name="HTTP Server S">
   <port>80</port>
   <port_type>CREATE_ONLY</port_type>
   <listening_interface>10.0.0.2</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <port_type>CONTROL_ONLY</port_type>
   <listening_interface>200.0.0.2</listening_interface>
   <keystore>
      <keystore_file>push2.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<control_link_address>push2.mycompany.com</control_link_address>
<prestarted_max_queue>1000</prestarted_max_queue>
```

The `<prestarted_max_queue>` setting is recommended here, in order to leverage the backpressure features provided by the Server. In fact, in this particular configuration, the “CR” connections, being the only ones in TLS/SSL, may be slower than the “S” connections in the establishment phase. This may pose problems, in particular, with **BaseSS** clients (as defined above), whereas, in case of massive client arrival, this may originate a queue of session establishment requests waiting for the final *rebind connection*, with growing session establishment latencies. The `<prestarted_max_queue>` setting puts a limit on this queue; the best value should be determined experimentally.

WebSocket: This option is particularly useful If the Balancer does not support WebSocket. Only the “S” connection (which is always HTTPS-based) passes through the Balancer, while the “CR” connections (which can be upgraded to WSS) go straight to the Lightstreamer Servers.

4 Stickiness Expiration

When LB stickiness (also known as *affinity*) is leveraged, it is possible that the Load Balancer sets a maximum duration for the association of a client to a Server instance, after which a new association is eventually established. This may be even a necessary choice, in order to prevent that any imbalance situation occasionally arisen may persist (as further discussed in section 5).

In this scenario, it is possible that the Server instance associated to a client changes during the life of a client session. This would compromise the correct behavior of the session, although the malfunctioning would not start immediately, but only upon the next [**CR**] connection that gets diverted to the other instance. This, in turn, might not occur at the very next connection, depending on transport details, as, for instance, if **WebSocket** were in use, or if **HTTP** connection reuse were operated by the user-agent.

So, what happens is that a request suddenly reaches a wrong Server instance and this implies that the LB, from that moment, will stick the client to the new instance. This case must be handled on the client side by opening a new session and migrating all the current activity to this session.
The issue manifests itself in several different ways, depending on the specific client SDK in use and its version, and the handling also differs in the various cases. Let's see the possible combinations:

1. When a *rebind connection* reaches a wrong Server instance, it fails and the whole session is interrupted. The client can react in various ways, depending on the specific client SDK in use:
   - The most recent clients SDKs will notify the session closure to the application with error code 21. The application will then be responsible for recovering with a new session. In most SDKs, by restoring the connection with the Server, the subscriptions will be resubmitted automatically.
   - Previous client SDKs will receive an internal "Sync Error" event and will handle the recovery with a new session automatically. Note that, in case of a misbehaving stickiness, this may lead to an endless loop of unsuccessful sessions.
     This automatic recovery with a new session may also occur with the most recent client SDKs, when the Server assumes that the previous instance was just replaced.
   - Old SDKs will notify session closure to the application with a "Session not found" cause. The application will then be responsible for recovering with a new session. In some SDKs, by restoring the connection with the Server, the subscriptions will be resubmitted automatically.
     This is still possible with the latest Generic Client SDK, when the Server assumes that the previous instance was just replaced.

2. On the other hand, When a *control connection* reaches a wrong Server instance, it fails, although the session is not interrupted and the request may be resubmitted. Nevertheless, the client is notified and it can react in various ways, depending on the specific client SDK in use:
   - The most recent clients SDKs will recognize the case and close the session, notifying the application with error code 21. The application will then be responsible for recovering with a new session. In most SDKs, by restoring the connection with the Server, the subscriptions will be resubmitted automatically.
   - Previous client SDKs will receive an internal "Sync Error" event and will close the session and handle the recovery with a new session automatically. Note that, in case of a misbehaving stickiness, this may lead to a loop of unsuccessful sessions.
     This automatic recovery with a new session may also occur with the most recent client SDKs, when the Server assumes that the previous instance was just replaced.
   - Old SDKs will notify the unsuccessful control request to the application with a "Session not found" cause. The application will then be responsible for closing the current session and recovering with a new session. In some SDKs, by restoring the connection with the Server, the subscriptions will be resubmitted automatically.

Clients based on the Generic Client SDK also fall into this case; note that with the latest Generic Client SDK, a different notification (error code 11) is used, unless the Server assumes that the previous instance was just replaced..

Finally, note that when “Application Stickiness” (as defined previously) is leveraged, the associated Server instance is kept by the client library for the whole life of the session. Hence, the stickiness is always guaranteed.

5 How to Force a Rebalancing

The stickiness requirement may lead to uneven balancing. A typical case is when an instance fails, or when an instance (or each instance in turn) is shut down for maintenance.
When this happens, the Load Balancer will reset all associations towards this instance, and all clients, upon their first reconnection attempt, will be assigned to other instances, possibly causing overload issues. When, later, the failed instance is restarted and added back to the pool, it becomes available to relieve the other instances from the extra load, but it cannot receive from the Load Balancer any connections related with the currently active clients, because they are constrained to the other instances.
Even if, in a cloud scenario, a new instance is added to the pool immediately, the reconnection attempts, for balancing reasons, may not be all resubmitted to the new instance.

If a similar imbalance arises, it could be an acceptable trade-off to interrupt part of the client sessions served by an overloaded instance and change their associations so as to migrate these clients to an idle instance.

Further considerations depend on the way stickiness is managed.

5.1 When LB Stickiness Is Leveraged

Even without explicitly interrupting some sessions, enforcing the Load Balancer to discontinue and change the current association for some clients may be enough to cause their sessions to migrate in reasonable time. As explained in section 4, stickiness interruption must be managed by the application properly, to minimize the impact.

A simpler approach could be to configure the Load Balancer to set a maximum duration of the stickiness for any client, after which a new association would be established. This can lead to a continuous and gradual adjustment of the balance. On the other hand, this would cause occasional interruptions of the client sessions (which will be promptly recovered by the clients) also when no rebalancing needs are in place. So, the choice of the duration to be set should be a trade-off.

Note that, as also pointed out in section 4, there are cases in which the discontinuation of the stickiness may not be detected by the client for an arbitrarily long time. To cope with these cases, a maximum duration limit for all sessions can be configured, in addition, on the Server, through the `<max_session_duration_minutes>` setting. This ensures that the session is eventually interrupted with a specific error code, hence the client can recover and migrate in reasonable time. Again, the choice of the duration to be set should be a trade-off.

How the client recovers depends on the specific client SDK in use. In some SDKs, recovery from this kind of interruption is managed by the SDK library; in the remaining ones, the session interruption with code 48 is notified to the application, which should take care of opening a new session.
However, for SDKs released earlier than Server 6.0, code 48 is not available and the session interruption will appear to the client as either a socket interruption or a lack of data, depending on the transport; hence it will be managed accordingly, again depending on the SDK. To avoid complications, when `<max_session_duration_minutes>` is leveraged, sticking to client SDKs earlier than Server 6.0 is not advised.

5.2 When Application Stickiness Is Leveraged

When application stickiness is used, any time a new session is established, the associated Server instance is allowed to change. As a consequence, a continuous and gradual adjustment of the balance is always in place.

On the other hand, the associated Server instance is kept by the client library for the whole life of the session (i.e. no stickiness discontinuation is possible). So, in scenarios in which client sessions last for very long time, issues with persisting imbalance are still possible.
As in the previous case, this can be coped with by configuring on the Server a maximum duration limit for all sessions, through the `<max_session_duration_minutes>` setting. Needless to say, this would cause occasional interruption of the client sessions (which will be promptly recovered by the clients) also when no rebalancing needs are in place; hence the choice of the duration to be set should be a trade-off.

5.3 Notes on Instance Replacement

The approaches based on a maximum duration of balancing and/or of the Session itself may help in improving the balancing on the medium term, but prove insufficient to fully handle the case of instance failure or replacement. In particular, they don't help coping with the consequential massive reconnection requests, which stress the cluster. Moreover, after a massive reconnection, all those sessions which started at the same time may also expire at the same time, unneededly reiterating the stress.

In case of a controlled instance replacement, a more sophisticated technique is available through the use of the JMX interface, provided that JMX management is available (since full JMX features is an optional feature, available depending on Edition and License Type).
So, instead of shutting down the instance and closing all sessions immediately, the sessions can be closed gradually before issuing the shutdown, by invoking the `drainSessions` operation on the Server Mbean. The rate of session closes can also be configured.
The closures will be notified to the clients with code 48, which, as discussed above, instructs the client to reconnect.

If application stickiness is leveraged, during the draining phase, the instance can even be detached from the cluster, to keep it from new session creation requests, and still remain reachable for control requests for the trailing sessions.
Otherwise, the instance can be protected by also invoking the `block` operation on the Server Mbean. New session creation requests will be refused and, again, the client will be instructed to retry, hopefully to be assigned to a different instance.

6 Accessing Server Instances Directly and the Monitoring Dashboard Case

In some cases, mainly for testing purpose or for internal use, it may be needed that a browser-based client or an application client connects to a specific instance of the Server in the cluster, by addressing it through a direct hostname.
This is in general possible, but in all scenarios in which the `<control_link_address>` setting is leveraged, that setting would still be used for [**CR**] connections and this could cause the application not to work, in case it were running inside the organization and using a private hostname, whereas the public hostname specified in `<control_link_address>` were not visible.

This limitation can be overcome by instructing the application to ignore the `<control_link_address>` setting. This can be done for clients based on the Unified Client API, in which a proper configuration setting has been made available to application code. Some code similar to the following should be added to initialization code before connecting:

```
    lsClient.connectionOptions.setServerInstanceAddressIgnored(true);
```

Obviously, a client that ignores the `<control_link_address>` may not work properly if it connects to the Server through the Load Balancer address.

A special case is a demo or test web front-end whose pages are directly hosted by Lightstreamer Server through its basic Web Server support. For instance, the demo applications available in Lightstreamer distribution package are deployed in this way.
These front-ends, typically, don't specify a Server hostname, so they access the Server through the same hostname by which their pages have been requested.
These front-ends could be recalled either through the Load Balancer address or through a direct Server instance hostname. In the latter case, the above considerations hold.
By the way, note that, in both cases, if the `<control_link_address>` setting is leveraged, these front-ends may still be subject to using different hostnames to access the Server.

The Monitoring Dashboard is also a browser-based application whose front-end is hosted (and directly supplied) by Lightstreamer Server.
Moreover, the Monitoring Dashboard is available for production use.
As the purpose of the Monitoring Dashboard is that of inspecting single instances, it is configured to ignore any `<control_link_address>` setting; so it can be recalled through any public or private hostname of any Server instance, whereas recalling it through the Load Balancer address is not supported.