Clustering for Load Balancing and Fail Over

Last updated: 16/11/2016

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. It is an HTTP POST request, by which the client authenticates and gets its client-id. In some cases, this connection is short lived. In other cases, it is long lived and implements the actual streaming channel. 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 some cases. These connections require to be sticky to the server that has served the corresponding session creation connection. These connections can be based either on HTTP or WebSocket.

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

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

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. The Lightstreamer Web Client Guide clarifies the various cases and summarizes the related configuration requirements (see “Deployment and Browsers”).
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 the 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 level-4 (TCP), rather than at level-7 (HTTP).

2.1.2 Option 1.A.2 – Cookie Based

The Balancer must support a "Sticky Cookie-Insert" mode, that means that the a 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 the 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 2965 or 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.
The Web Client Guide document illustrates various cases (see Deployment and Browsers).

For clients implementing the Unified Client API and only for startup performance reasons, the following should also be added to the client initialization code, before connecting:

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

WebSocket: As this option works at level 7 (level 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 most 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, BUT PARTIAL	Managed by LS library, but not for WebSocket; Remember to issue setCookieHandlingRequired(true)
Java SE	TO BE ENFORCED	Just issue: java.net.CookieHandler.setDefault(new java.net.CookieManager(null, java.net.CookiePolicy.ACCEPT_ALL));

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

For clients implementing the Unified Client API and only for startup performance reasons, the following should also be added to the client initialization code, before connecting:

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

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

The final nodes can be directly addressed by the Client through the 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	80
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 VIPs are 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">
   <port>80</port>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
<http_server name="HTTP Server">
   <port>80</port>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

Notice that the same solution as Option 1.B.2 could be used too (use three different VIPs on the Load Balancer rather than doing L7 switching).

WebSocket: If the Balancer supports WebSocket, the connections will be automatically upgraded from HTTP when necessary, otherwise the Client will detect WebSocket failure and will keep using HTTP.

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. The Web Client Guide illustrates such cases and contains pointers to possible solutions (see Deployment and Browsers).

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	200.0.0.10	80	LS Server 1	200.0.0.1	80

		Balancer			or LS Server 2 (balancing)	or 200.0.0.2	
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.

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

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

On Lightstreamer Server 2:

```
<http_server name="HTTP Server">
   <port>80</port>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

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

HTTPS has a limitation with respect to HTTP. Since the TLS/SSL handshake is usually performed before the application protocol (HTTP) can start[2], the Balancer cannot detect the hostname indicated in the application request, because it is encrypted and only the final Server can decode it (unless SSL Acceleration is used), so, for example, Option 1.B.2 is not possible. Furthermore, the same instance of the Lightstreamer Server should be accessed with two different hostnames; but a traditional SSL certificate associated to the server socket can contain only one hostname, resulting in a security warning on the Client when the second name is used. Two solutions exist to this issue: using Wildcard SSL Certificates, or using multiple certificates on different sockets.

Note that some options below 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. 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.

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 Option 1.A.1 apply from a theoretical point of view (potentially making this option unreliable), but client IP switching does not usually happen with HTTPS.

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

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

```
<https_server name="HTTPS Server">
   <port>443</port>
   <keystore>
```

[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

```
      <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).

3.1.1.1 Example of Multihosting

In order to cope with this case, we must extend the configuration in order to allow each Server instance to provide different SSL 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

Two traditional SSL certificates are needed for the two different hostnames.

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.

As an alternative, a Multi-Domain SSL Certificate could be used, so that one certificate would be enough for all the Lightstreamer Server instances. But please consider that Multi-Domain Certificates have not been tested with Lightstreamer yet, though they should work[3].

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. The Web Client Guide illustrates such cases and contains pointers to possible solutions (see Deployment and Browsers).

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 Option 1.A.2 apply.

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 the 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);
```

The Web Client Guide document clarifies the various cases (see Deployment and Browsers).

For clients implementing the Unified Client API and only for startup performance reasons, the following should also be added to the client initialization code, before connecting:

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

For non-browser clients, see the additional notes provided in section Error: Reference source not found.

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

[3] For more information, see for example:
- http://www.whichssl.com/mdc.html
- http://www.jguru.com/forums/view.jsp?EID=1358409

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.

For clients implementing the Unified Client API and only for startup performance reasons, the following should also be added to the client initialization code, before connecting:

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

3.2.1 Leverage Wildcard 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 using the Common Name "***.mycompany.com**" may be used to secure all the following domains: "push.mycompany.com", "push1.mycompany.com", "push2.mycompany.com".

The use of Wildcard Certificates is addressed in RFC 2818[4], section 10. Browser compatibility is quite high, where typically only older micro-browsers do not support wildcard certificates[5]. For more information on wildcard certificates, including possible limitation, you may visit the specific pages of some well-known certification authorities sites[6].

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

An SSL certificate is issued using the Common Name "***.mycompany.com**".

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	443
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).

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

[5] For example, see http://www.geocerts.com/ssl/browsers

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

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
<https_server name="HTTPS Server">
   <port>443</port>
   <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">
   <port>443</port>
   <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>
```

WebSocket: As this option works at level 4 (level 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. The Web Client Guide illustrates such cases and contains pointers to possible solutions (see Deployment and Browsers).

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”).

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	443
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	444
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

Two wildcard SSL certificates are needed for the two different domains.

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
<https_server name="My HTTPS Server">
   <port>443</port>
   <keystore>
      <keystore_file>wildcard_mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="Our HTTPS Server">
   <port>444</port>
   <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">
   <port>443</port>
   <keystore>
      <keystore_file>wildcard_mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="Our HTTPS Server">
   <port>444</port>
   <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>
```

As an alternative, the same considerations on Multi-Domain SSL Certificate reported in section 3.1.1.1 apply.

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

An SSL certificate is issued using the Common Name "***.mycompany.com**".
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)	200.0.0.1 or 200.0.0.2	443
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.

The lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
<https_server name="HTTPS Server">
   <port>443</port>
   <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">
   <port>443</port>
   <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>
```

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 level 4 (level 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 SSL certificate is issued using the Common Name “***.mycompany.com**”.
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.

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	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">
   <port>80</port>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

```
<http_server name="HTTP Server">
   <port>80</port>
</http_server>
<control_link_address>push2.mycompany.com</control_link_address>
```

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

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

An SSL certificate is issued using the Common Name "***.mycompany.com**".
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. The Load Balancer mounts the SSL certificate associated to "push.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>
   <listening_interface>10.0.0.1</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <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:

```
<http_server name="HTTP Server S">
   <port>80</port>
   <listening_interface>10.0.0.2</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <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>
```

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

In case wildcard certificates cannot be used, individual certificates should be issued for each hostname used in the cluster.

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 443 will handle the "push.mycompany.com" certificate, while the server socket on port 444 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	443
CR	push1.mycompany.com	Load Balancer	200.0.0.1	443	LS Server 1	10.0.0.1	444
CR	push2.mycompany.com	Load Balancer	200.0.0.2	443	LS Server 2	10.0.0.2	444

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>443</port>
   <keystore>
      <keystore_file>push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="HTTPS Server CR">
   <port>444</port>
    <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>443</port>
   <keystore>
      <keystore_file>push.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="HTTPS Server CR">
   <port>444</port>
    <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 level 4 (level 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. The Web Client Guide illustrates such cases and contains pointers to possible solutions (see Deployment and Browsers).
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”).

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	443
CR	push1.mycompany.com	Load Balancer	200.0.0.1	443	LS Server 1	10.0.0.1	444
CR	push2.mycompany.com	Load Balancer	200.0.0.2	443	LS Server 2	10.0.0.2	444
S	push.ourcompany.com	Load Balancer	200.0.0.20	443	LS Server 1 or LS Server 2 (balancing)	10.0.0.11 or 10.0.0.12	443
CR	push1.ourcompany.com	Load Balancer	200.0.0.11	443	LS Server 1	10.0.0.11	444
CR	push2.ourcompany.com	Load Balancer	200.0.0.12	443	LS Server 2	10.0.0.12	444

Six SSL certificates are needed for the six different hostnames.

The Lightstreamer_conf.xml file should contain the following elements:

On Lightstreamer Server 1:

```
<https_server name="My HTTPS Server S">
   <port>443</port>
   <listening_interface>10.0.0.1</listening_interface>
   <keystore>
      <keystore_file>mypush.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="My HTTPS Server CR">
   <port>444</port>
   <listening_interface>10.0.0.1</listening_interface>
    <keystore>
      <keystore_file>mypush1.keystore</keystore_file>
      <keystore_password>mypassword</keystore_password>
   </keystore>
</https_server>
<https_server name="Our HTTPS Server S">
   <port>443</port>
   <listening_interface>10.0.0.11</listening_interface>
   <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>
    <listening_interface>10.0.0.11</listening_interface>
     <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>443</port>
    <listening_interface>10.0.0.2</listening_interface>
    <keystore>
        <keystore_file>mypush.keystore</keystore_file>
        <keystore_password>mypassword</keystore_password>
    </keystore>
</https_server>
<https_server name="My HTTPS Server CR">
    <port>444</port>
    <listening_interface>10.0.0.2</listening_interface>
     <keystore>
        <keystore_file>mypush2.keystore</keystore_file>
        <keystore_password>mypassword</keystore_password>
    </keystore>
</https_server>
<https_server name="Our HTTPS Server S">
    <port>443</port>
    <listening_interface>10.0.0.12</listening_interface>
    <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>
    <listening_interface>10.0.0.12</listening_interface>
     <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, can be bound to the same ports of two different IP addresses. For example, the server socket on 10.0.0.1:443 (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. This option is certainly less common and less convenient than Option 2.B.2.1, though technically doable.

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	443
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>443</port>
   <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>
   <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>443</port>
   <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>
   <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 level 4 (level 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	80
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">
   <port>80</port>
</http_server>
<control_link_address>push1.mycompany.com</control_link_address>
```

On Lightstreamer Server 2:

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

WebSocket: If the Balancer does not support WebSocket, configuring it at level-4 (TCP) can in some cases make WebSocket work, provided that the Balancer supports level-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. 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>
   <listening_interface>10.0.0.1</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <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:

```
<http_server name="HTTP Server S">
   <port>80</port>
   <listening_interface>10.0.0.2</listening_interface>
</http_server>
<https_server name="HTTPS Server CR">
   <port>443</port>
   <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: 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.

This case must be handled on the client side in a suitable way.

1. When a *rebind connection* reaches a wrong Server instance, it fails and the whole session is interrupted by “Sync Error”. This event is similar to a connection interruption and its handling should fall into that case, hence a new session should be opened by the client.
   How this is done depends on the specific client SDK in use. In some SDKs, recovery from interruptions is managed by the SDK library; in the remaining ones, the session interruption is notified to the application, which should take care of opening a new session.

2. On the other hand, When a *control connection* reaches a wrong Server instance, it fails by reporting a “Sync Error”, but the session is not interrupted. This event should be handled by the client by forcing a session close, in order to fall, then, into the previous case.
   Again, how this is done depends on the specific client SDK in use. In some SDKs, recovery from errors in subscriptions or other control request attempts is managed by the SDK library; in the remaining ones, the “Sync Error” is notified to the application, which should take care of closing the session and opening a new one.

Finally, note that when application stickiness 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. In this case, 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.

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, having the Load Balancer 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 interruption 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 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.

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.