General Concepts

Target: Lightstreamer Server v. 7.4 or greater
Last updated: 8/5/2023

Table of Contents

About This Document....4
1 Architecture Overview....5
1.1 Architecture Components....5
1.1.1 Lightstreamer Server....6
1.1.2 Lightstreamer Adapters....7
1.1.3 Lightstreamer Clients....7
1.2 Summary of the Components....8
2 High-level Concepts....10
2.1 Multichannel Transport....10
2.2 Message Routing....11
2.3 Optimized Delivery....13
2.3.1 Dynamic Throttling....13
2.3.2 Other Optimizations....14
2.4 Scalability....15
2.5 Security....16
2.6 Monitoring....16
3 Main Concepts....18
3.1 Glossary....18
3.2 Data Model and Subscription Modes....20
3.2.1 MERGE Mode....21
3.2.2 DISTINCT Mode....23
3.2.3 COMMAND Mode....24
3.2.4 RAW Mode....25
3.2.5 Compatibility of Modes....26
3.3 Bandwidth and Frequency Management....26
3.3.1 Bandwidth Management....26
3.3.2 Frequency Management....27
3.3.3 Applying the Limits....27
3.3.4 Filtered and Unfiltered Modes....27
3.3.5 The ItemEventBuffer....28
3.3.6 ItemEventBuffer in RAW Mode....29
3.3.7 ItemEventBuffer in MERGE Mode....29
3.3.8 ItemEventBuffer in DISTINCT Mode....30
3.3.9 ItemEventBuffer in COMMAND Mode....30
3.3.10 Resume of Settings for the ItemEventBuffer....31
3.3.11 Considerations on the Usage of the Buffer....31
3.3.12 Selectors....32
3.4 The Preprocessor....32

3.4.1 Snapshot....32
3.4.2 Prefilter....33
3.4.3 Preprocessor and Modes....34
3.5 Roles in the Choice of Parameters....34
4 The Adapters....36
4.1 The Metadata Adapter....36
4.1.1 Authentication and Authorization Examples....38
4.2 The Data Adapter....39
4.2.1 An Internal Data Adapter: The Monitoring Data Adapter....42
5 Mobile and Web Push Notifications....49
5.1 Data Streaming vs. Push Notifications....49
5.2 The Lightstreamer MPN Module....50
5.3 The MPN Device....51
5.3.1 Lifecycle....52
5.4 MPN Subscriptions....53
5.4.1 The Notification Format....53
5.4.2 The Trigger Expression....55
5.4.3 Coalescence....56
5.4.4 Notification Limits....57
5.4.5 Lifecycle....58
5.5 The Internal MPN Data Adapter....58
5.6 The Database Provider....60
5.6.1 The Database in a Clustered Configuration....61
5.6.2 DBMS Compatibility....61
5.7 The MPN Provider....63
5.7.1 APNs....63
Safari Push Notifications....64
The Push Package File....64
5.7.2 FCM....65
The Service JSON File....66
Chrome and Firefox Push Notifications....66
5.8 Workflow Examples....67
5.9 Special Considerations on the GCM to FCM Transition....68
5.9.1 How to Update Your Android MPN App to Server 7.1....69

About This Document

This document provides a comprehensive picture about the general architecture of Lightstreamer, by describing its main features and design principles.

The document is intended for Architects, Developers, System Integrators and System administrators who have the responsibility of design, develop, integrate, deploy and administer every Lightstreamer based solution.

This document is structured in the following sections:

- Architecture Overview outlines the building blocks of the Lightstreamer Architecture.
- High-level Concepts Introduces some of the characteristics of the Lightstreamer technology.
- Main Concepts provides fundamental notions needed to understand the internal mechanisms of Lightstreamer.
- The Adapters illustrates how custom Adapters work.
- Mobile and Web Push Notifications describes role and functioning of the special module designed to integrate both Apple Push Notification Service and Google Cloud Messaging.

1 Architecture Overview

Lightstreamer is a real-time engine that delivers real-time data over HTTP and WebSockets to/from any types of clients (HTML pages, native mobile applications, desktop applications, etc.), as well as native push notifications to mobile clients.

By implementing unique features such as **Bandwidth & Frequency Throttling** and **Adaptive Streaming**, Lightstreamer handles the distribution of live data to the clients in a reliable, light, and efficient way.

Lightstreamer is able to pass through any firewall and proxy by using many sophisticated mechanisms without the risk of security policy blockages.

This chapter provides an overview of the Lightstreamer architecture.

NOTE: Some of the features described in this document may be available on some Lightstreamer editions only. Please refer to the product matrix to know the details (https://www.lightstreamer.com/ls-features).

1.1 Architecture Components

The Lightstreamer Server is the core component of the Lightstreamer technology. It is a stand-alone process that runs in a Java Virtual Machine. It handles the connections with the clients and dispatches real-time data back and forth between the clients and any back-end system.

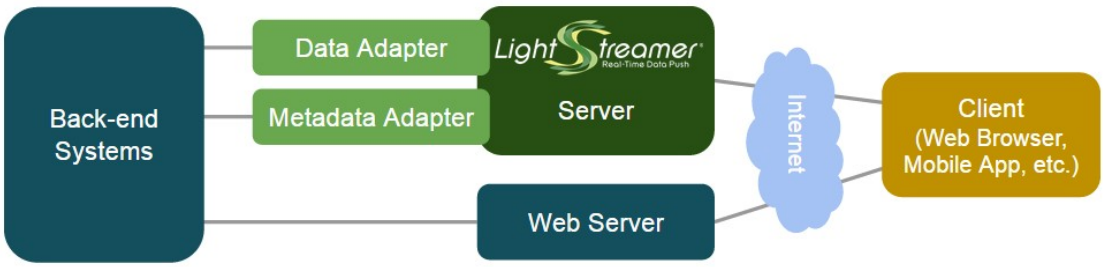


Figure 1- Lightstreamer General Architecture.

Although Lightstreamer comes with an embedded Web server to serve static resources, which makes the product immediately useful for demo purposes, the deployment architecture shown above is normally employed in real production scenarios.

Typically, the Lightstreamer Server and a common Web Server (Apache, nginx, IIS or any other web server) are deployed in the DMZ network, accessible over the Internet by clients. Lightstreamer Server gets the data and metadata from the back-end systems that are protected by the second firewall.

The clients (web browser, mobile applications, desktop applications, etc.) get synchronous data from the Web server (*pulled data*, such as web pages or any request/response interaction) and asynchronous real-time data from the Lightstreamer Server (*pushed data*). The clients can also send real-time data to the Lightstreamer Server. This way, bi-directional streaming (or two-way push) is fully supported.

In addition, native push notifications (Apple APNs and Google FCM) are supported for mobile clients (see §5).

It is possible to create a **cluster** of Lightstreamer Servers by using any Web load-balancing appliance, in order to achieve **load-balancing** and **fail-over**. If one of the machines in the cluster fails, the clients will reconnect to another machine. The **recovery** is transparent to the users, while full data coherency is maintained. See the "Clustering" documentation for more details

1.1.1 Lightstreamer Server

The Kernel of Lightstreamer Server is the "off-the-shelf" part of the system. It takes direct control over the operating system's TCP/IP stack and handles the real-time flow of data for each client to optimize the data transmission as much as possible. The Kernel implements several important features to offer an efficient and reliable push service, such as:

- **Scalability**. The architecture of the Kernel is based on a staged event-driven architecture, that allows many thousands of concurrent connections to be sustained by a single server CPU.
- **Firewall and Proxy Friendly**. The use of HTTP and HTTPS over standard ports to deliver data to the Clients allows the traffic to pass through the strictest firewalls, proxies and routers. Some proxy servers exist that block any form of streaming, because they try download the full content of a document before forwarding it to the client. In this cases, the **Stream-Sense** features of Lightstreamer automatically detects the issue and switches to a high-efficiency smart polling mode.
- **Bandwidth Control**. For each user a maximum bandwidth can be allocated, dedicated to the streaming channel. For example, if it is required that a certain user cannot exceed a bandwidth of 10 kbps, the Kernel will filter the data in such a way as to ensure that the streaming connection for that user always remains below 10 kbps.
- **Frequency Control**. A maximum update frequency can be allocated for each user/item combination. For example, it is possible to configure the profile of a certain user so that they do not receive more than 2 updates per second for a certain subscribed item (piece of data).
- **Adaptive Streaming and Congestion Control**. The Kernel automatically detects situations of congestion on the network, heuristically slowing down or suspending the dispatch of data until the connection is again fully serviceable. Data aging is avoided.
- **TCP-Level Optimization**. The Kernel has direct control over the composition of the TCP packets. Instead of delegating the aggregation of data in packets to the operating system (using the *Nagle* algorithm), the Kernel itself decides on each occasion the optimum composition of each TCP packet, with the objective/trade-off of reducing the latency before data is dispatched as well as minimizing the number of dispatched packets.
- **Prefiltering, Selector Mechanism, Event Customization**. Advanced features to exploit the data flow before it is delivered to the clients.
- **JMX Manageability** Any management console compliant with Java Management Extensions can communicate with the JMX agent included in Lightstreamer Kernel to collect thorough data about the Server status and to manage the system.

1.1.2 Lightstreamer Adapters

Lightstreamer Adapters are custom server-side components attached to the Lightstreamer Server, whose role is to interface the Kernel with any data source, as well as to implement custom authentication and authorization logic. An *Adapter Set* is made up of a Metadata Adapter and one or multiple Data Adapters, both developed by implementing provided interfaces.

Adapter APIs are currently provided for **Java**, **.NET**, **Node.js**, and **Python**. In the case of Java, Adapters can run either in-process with the Lightstreamer Server or in an external process (on the same machine or on a different machine). .NET and Node.js Adapters always run in external processes. In addition, Adapters can be developed in any language to talk to the Lightstreamer Server via plain **TCP sockets** through the **Adapter Remoting Infrastructure** (see the specific documentation for full specification of the protocol).

Lightstreamer Metadata Adapter

The role of a Metadata Adapter is to manage the policies applied to a user's session. It controls the life cycle of user sessions, performing the following tasks:

- validation of authentication credentials sent by the client
- assigning of the granted quality of service to each user, in terms of allocated bandwidth and update frequency
- interpretation, validation and authorization of subscription requests, made up of items and field names

Lightstreamer Data Adapter

The Kernel of Lightstreamer is integrated with any data feed (the source of real-time information) by means of a Data Adapter.

Each Data Adapter is a connector that interfaces Lightstreamer Server with a data source. It receives a flow of data from the back-end systems (information providers, data feeds, databases, application servers, CRMs or any other legacy platforms) and injects it into the Lightstreamer Kernel for controlled delivery to individual users.

The system integrator that implements the Data Adapter is free to use any technology to integrate the Kernel with the data feed. However, it is preferable to use middleware that provides asynchronous paradigms, such as message-oriented systems, so as not to break the asynchronous chain that goes from the feed to the user's client. In any case, it is also possible to use polling techniques to get updates from the data feed (e.g. polling a file or a database).

Because Lightstreamer is able to manage an arbitrary number of different Data Adapters and Metadata Adapters (grouped together into Adapter Sets), *it is possible to integrate heterogeneous sources of information while maintaining a single point of access to the streaming/push channel.*

1.1.3 Lightstreamer Clients

Lightstreamer client libraries are provided for many different platforms, which expose APIs to integrate any application with the Lightstreamer Server. The available client APIs include:

- Web (compatible with any browser, including older browsers and mobile browsers; supports frameworks like AngularJS and React, as well as hybrid frameworks, such as PhoneGap and Electron)

- Android
- Apple (iOS, macOS, tvOS, watchOS)
- Microsoft (.NET, Excel, Silverlight, Windows Phone, WinRT)
- Java SE
- Node.js (for both server-side code and React Native apps)
- Python
- Unity
- Adobe (Flash, Flex, AIR)
- Java ME and BlackBerry
- Generic client (open protocol)

1.2 Summary of the Components

To resume, Lightstreamer comprises the following main components:

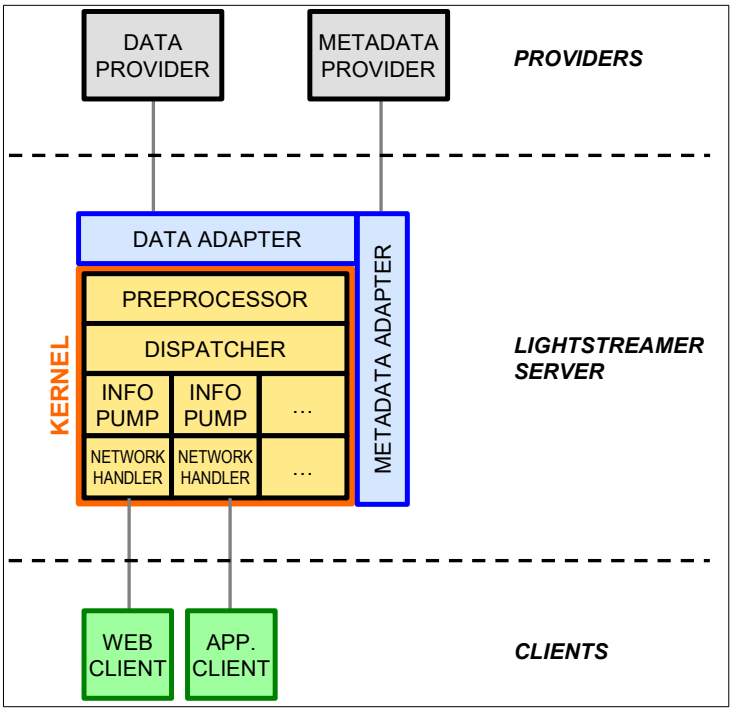


Figure 2- The Components of Lightstreamer Server.

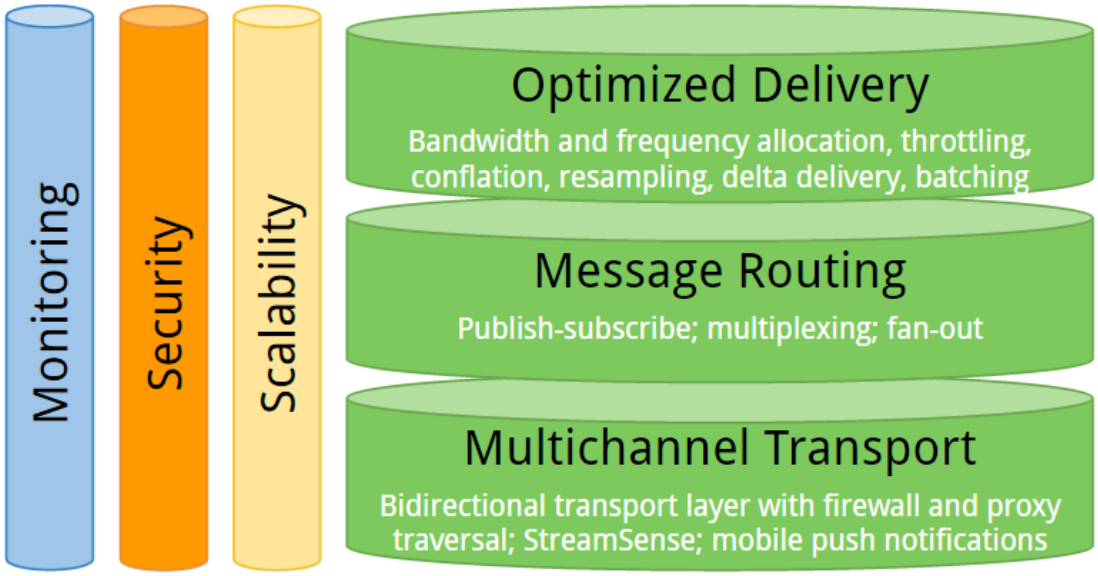
- **Data Provider –** Data feed that provides Lightstreamer with real-time data updates.
- **Metadata Provider –** Metadata feed that provides Lightstreamer with authentication information, permissioning policies and description of data (for example, items and fields).
- **Lightstreamer Data Adapter –** Custom component that integrates Lightstreamer Kernel with the Data Provider.
- **Lightstreamer Metadata Adapter –** Custom component that integrates Lightstreamer Kernel with the Metadata Provider.

- **Lightstreamer Kernel –** The core of the server, which in turns is comprised of the following sub-components:
  - **Preprocessor** – performs Prefiltering and pre-elaboration on update events.
  - **Dispatcher** – forwards real-time updates from the Data Adapter.
  - **InfoPump** – data structure allocated by the Kernel for each activated session.
- **Lightstreamer Client –** Any kind of application (Web-based, mobile, tablet, desktop, as well as server processes) that receives/sends real-time data from/to Lightstreamer Server through a TCP/IP network (using only HTTP and/or WebSockets). A Client subscribes to a set of data that the Server publishes.

2 High-level Concepts

From a functional perspective, Lightstreamer is made up of three logical layers: Multichannel Transport, Message Routing, and Optimized Delivery.

Scalability, Security, and Monitoring are three essential properties.



2.1 Multichannel Transport

The transport layer offers the upper layer the abstraction of a reliable network transport that works in any situation. Web protocols (HTTP and WebSockets) are used. This, together with several sophisticated mechanisms, enables bi-directional real-time communication through any kind of proxies, firewalls, and other network intermediaries.

The **Stream-Sense** mechanism implements a fast detection of the best transport on a per-client basis, with a sequence of automatic fallbacks. For example, the JavaScript client library behaves like this:

Whatever is the transport automatically chosen for the communication with each client, the same exact set of features is guaranteed. In particular, a high-performance bidirectional channel is provided in all the cases, with in-order guaranteed message delivery and automatic batching from client to server. In other words, Lightstreamer enriches HTTP when sending messaged from the client to the server:

- Messages are acknowledged explicitly
- Lost messages are retransmitted automatically
- Out-of-order messages are reordered automatically
- Underlying socket is kept open for reuse via reverse heartbeats
- Multiple requests are automatically batched, to highly reduce the number of HTTP round trips

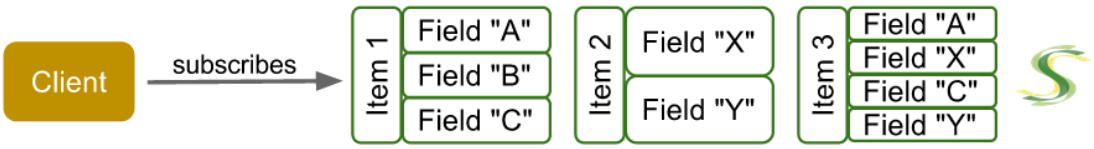
In addition to Web transports, Lightstreamer supports native transports for **push notifications** to mobile clients, where the protocol and infrastructure provided by Apple™ (**APNs**) and Google™ (**FCM**) are leveraged (see §5).

2.2 Message Routing

The Multichannel Transport layer offers reliable message delivery on the Internet. But this is not enough to build flexible and complex applications. Some message routing facility is needed to provide a convenient way to deliver the right data to the right recipients. For this reason, on the top of the Web Transport layer, Lightstreamer offers a powerful Message Routing layer.

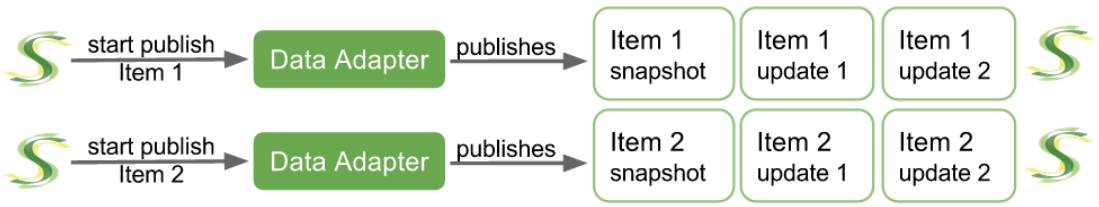
Lightstreamer provides a **publish and subscribe** (pub-sub) paradigm, to decouple message producers from message consumers.

A client subscribes to items (the basic piece of information) while specifying a schema (a set of fields for each item):



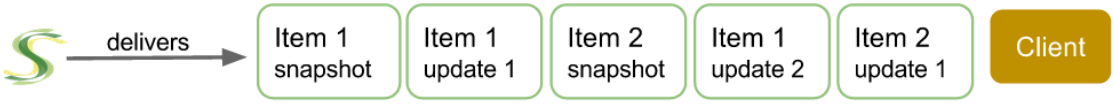
(The "S" logo represents the Ligthstreamer Server in this set of figures)

When an item is subscribed to by a client for the first time, the subscription is propagated to the Data Adapter, which should start publishing updates for that item. Any subsequent client subscribing to the same item does not cause any new request to the Data Adapter. When the last client unsubscribes from that item, the Data Adapter is notified it can stop publishing updates for that item:



Thanks to **on-demand publishing**, no data needs to be produced when no actual recipient exists, saving on systems resources.

The Lightstreamer Server receives from the Data Adapter the initial snapshot for each item (for example, the current value for all its fields), followed by the real-time updates. It then sends the data to each client, based on its subscriptions, in a multiplexed fashion, on the top of a single connection:



Thanks to the flexibility of the subscription-based mechanism, any message routing scenario is actually supported. From **unicasting** (where a message targets only one client), to **broadcasting** (where a message reaches all the clients), passing through **multicasting** (where a message reaches a group of clients).

For example, the top part of the figure above shows a case where the same item is subscribed to by one million clients. The Data Adapter needs to publish updates to that item only once, while the **fan-out** is done by the Lightstreamer Server.

The bottom part of the same figure shows the opposite case. An item is subscribed to by one client only, so that every client gets its own **personal message flow.**

Broadcast, multicast, and unicast items can be freely mixed as part of the same client session.

Lightstreamer implements an **asymmetric pub-sub** paradigm:

This means that the publishers are on the server side (connected to the Data Adapters) and the subscribers are on the client side. This highly optimizes the cases where there is some massive publishing from typical data feeds. In many scenarios, the data feed is completely different from the data consumer (due to topology, to protocol, or to the business model).

That being said, clients can still publish data, by sending direct messages to the Data Adapter (intermediated by the Metadata Adapter for security reasons):

This way, the Data Adapter has always the responsibility of publishing the actual updates for each item, irrespective of the fact that such updates are coming from a server-side data feeds or from the clients. Furthermore, in some cases the Data Adapter will want to validate, reformat, and process the data before publishing it. This is the extreme flexibility provided by the asymmetric pub-sub model, compared to traditional symmetric pub-sub systems, where publisher and subscribers are all peers (and some complex ESB might be needed to validate and adapt the data flows).

2.3 Optimized Delivery

Given a reliable and scalable Multichannel Transport abstraction and a flexible and secure Message Routing facility, a higher-level layer can be implemented, to optimize as much as possible the data delivery. The Delivery Optimization layer of Lightstreamer provides a very powerful set of features to save on bandwidth, reduce latency, and cope with the Internet unpredictability.

2.3.1 Dynamic Throttling

Based on the nature of the data, series of updates to an item can be filtered, to reduce their frequency (message rate), via:

- **Queueing** (buffer all the messages and deliver them at the proper rate)
- **Resampling** (discard some of the messages)
- **Conflation** (discard some of the messages while preserving the latest value for each field).

Imagine you have a sensor that measures the current temperature 100 times per second and you want to deliver the measurements in real time to a software application. If you just want to display the value to a human user, perhaps 10 updates per second are more than enough. Or, even better, you might want to deliver as many updates as possible without saturating the link bandwidth. This is where resampling and conflation are extremely useful.

For each subscription of each client, Lightstreamer allows to define how data can be filtered, with several parameters. Filtering is then applied on the fly to the data stream based on a number of static and dynamic variables.

There are three main variables that Lightstreamer takes into consideration to decide how to throttle the data flow:

1) Bandwidth allocation

For each client, a maximum bandwidth can be allocated to its multiplexed streaming connection. Lightstreamer guarantees that such max bandwidth is never exceeded, whatever is the original data update rate. Max bandwidth can even be changed dynamically during the life of a client session.

2) Frequency allocation

For each subscription of each client, a maximum update frequency (message rate) can be allocated. Lightstreamer guarantees that such max frequency is never exceeded, whatever is the original data update rate. Max frequency can even be changed dynamically during the life of a client session.

3) Real bandwidth detection

Internet congestions are automatically detected by the Lightstreamer Server and the actual available bandwidth at any point in time for each client is estimated.

In addition to these, there are other variables taken into considerations, either directly or indirectly (CPU cycle availability, batching configuration, etc.).

➔ A live demo is available, which shows Bandwidth and Frequency allocation in action: http://demos.lightstreamer.com/BandwidthDemo/

In a nutshell, Lightstreamer is able to continuously adapt the data flow to the network conditions. With data that can be resampled, Lightstreamer can send a different stream to each client, where the number of samples depends on the available bandwidth.

Suppose a mobile client connected over a tiny-bandwidth network with terrible packet loss and a desktop client connected with a broad-bandwidth connection subscribe to the same item. This item might contain the 3D coordinates of a user in a multiplayer game, or the telemetry data of an airplane, or the prices of a financial instrument. In all the cases, each client will receive an amount of data compatible with its network capacity. The first user above will probably receive less updates per second than the second user. But both will receive up-to-date data, not aged data that stacked up in a queue. Resampling done on the fly for each individual user is one of the most powerful features of Lightstreamer.

2.3.2 Other Optimizations

Several other techniques are employed by Lightstreamer to optimize the data delivery.

Data is aggregated efficiently within TCP packets.

The TCP Nagle's algorithm is disabled (TCP_NODELAY option) and Lightstreamer uses its own algorithms to decide how to pack data into TCP segments. A trade-off between latency and efficiency can be configured. What is the maximum latency you can accept to forge bigger packets and decrease the overheads (both network overheads and CPU overheads)? You can answer this question for each application and game and configure it in Lightstreamer via the *max_delay_millis* parameter. The higher the acceptable delay, the more data can be stuffed into the same TCP segment (**batching**), increasing overall performance. For extreme cases, you can use 0 for *max_delay_millis*.

Delta delivery.

With delta delivery, for subsequent updates to an item, only the actually changed fields (delta) are sent. For example, if a user subscribes to 20 different fields for each stock (last, bid, ask, time, etc.), only a few of them really change at every market tick. Lightstreamer automatically extracts the delta and the provided client-side library is able to rebuild the full state.

Moreover, the delta extraction can also regard the field value contents. For changed fields, the Server may send either the actual value or the difference between the previous and the new value. Several "diff" formats can be supported. While sending each value, Lightstreamer will decide if any "diff" algorithm is suitable and will use it to compute the difference to be sent, provided that the client-side library in use is able to rebuild the full value.

Upon initial connection, Lightstreamer sends a full snapshot for all the subscribed items.

Lightweight protocol.

Lightstreamer avoids using JSON, XML, or any other verbose protocol, which carries large amounts of redundant data with each event (for example, the field names), resulting in increased bandwidth usage. Lightstreamer uses a position-based protocol, which reduces the overhead to a minimum. The actual payload accounts for most of the bandwidth.

Nevertheless, custom field values can be of any type. Currently, if the JSON format is used for the values of some field, it is possible to specify whether the field is suitable for the use of JSON Patch as a "diff" format in delta delivery.

A description of these and other techniques, with some examples, is available in a Lightstreamer article, which describes a gaming use case but has general validity: http://blog.lightstreamer.com/2013/10/optimizing-multiplayer-3d-game.html

2.4 Scalability

The internal architecture of Lightstreamer Server makes it extremely scalable. It is based on a concurrent staged event-driven architecture with non-blocking I/O.

Performance and scalability depend on many different variables, but as a very rough figure, Lightstreamer has been tested on a single box with: one million connections with low frequency traffic; tens of thousands connections with very high frequency traffic. In other words, the actual scalability depends most of all on the message throughput.

Lightstreamer provides graceful degradation of the quality of service. If the server CPU saturates, due to wrong capacity planning or to an unexpected spike, the Lightstreamer Server will not block. It will simply start to reduce the update rate evenly on all the connections, while avoiding data ageing at the same time (thanks to adaptive throttling; see below).

The Lightstreamer Server can scale both vertically and horizontally.

Vertical scalability: The Lightstreamer Server automatically leverages all the CPUs and cores that are available on that machine, by spawning the right number of threads and avoiding any critical global synchronization point.

Horizontal scalability: Multiple Lightstreamer Servers can be clustered by using any standard Web Load Balancer. See the "Clustering" documentation for more details.

2.5 Security

Lightstreamer offers pluggable security policies via the Metadata Adapter. Basically, every authentication and authorization decision can be delegated to the Metadata Adapter, in a very fine-grained fashion. For example, upon the client session creation, the authentication credentials received from the client, together with further information (http headers, network properties, etc.) are passed to the Metadata Adapter for validation. The Metadata Adapter is free to query whatever back-end system it wants to decide whether to authenticate a user or not.

Similarly, each subscription done by a client, as well as any request for some quality of service (QoS), such as granted bandwidth or update frequency, is authorized through the Metadata Adapter.

See §4.1 for more details.

2.6 Monitoring

Lightstreamer offers several means to monitor the server.

A world-class **monitoring dashboard** is available since Lightstreamer 6.0. Based on HTML, it is configured to work out of the box and provides many real-time metrics and statistics on the behavior of the Lightstreamer Server. The dashboard is organized in five tabs:

- The first tab shows a visual representation of the Lightstreamer architecture, with several overlays containing the metrics (connections, sessions, throughput, memory, thread pools, etc.).
- The second tab displays real-time graphs of the main metrics.
- The third tab contains the logs.
- The fourth tab reports the details of the features allowed by the current license in use.
- The fifth tab hosts the "JMX Tree", where the monitoring objects can be browsed and inspected.

The Monitoring Dashboard is a convenient means of watching the server for both developers and systems administrators. By default, it is available at the "/dashboard/" URI of the Lightstreamer Server.

A very fine-grained configurable **logging** facility is available, with several categories, levels, and appenders. It is based on logback (http://logback.qos.ch/).

Extensive metrics are exposed via a **JMX interface**, to hook into any third-party application management tool. See the JMX Extension SDK for full details.

3 Main Concepts

3.1 Glossary

The picture below shows a graphical representation of the hierarchy between a Session and an Item. Definitions of each term are given below the figure.

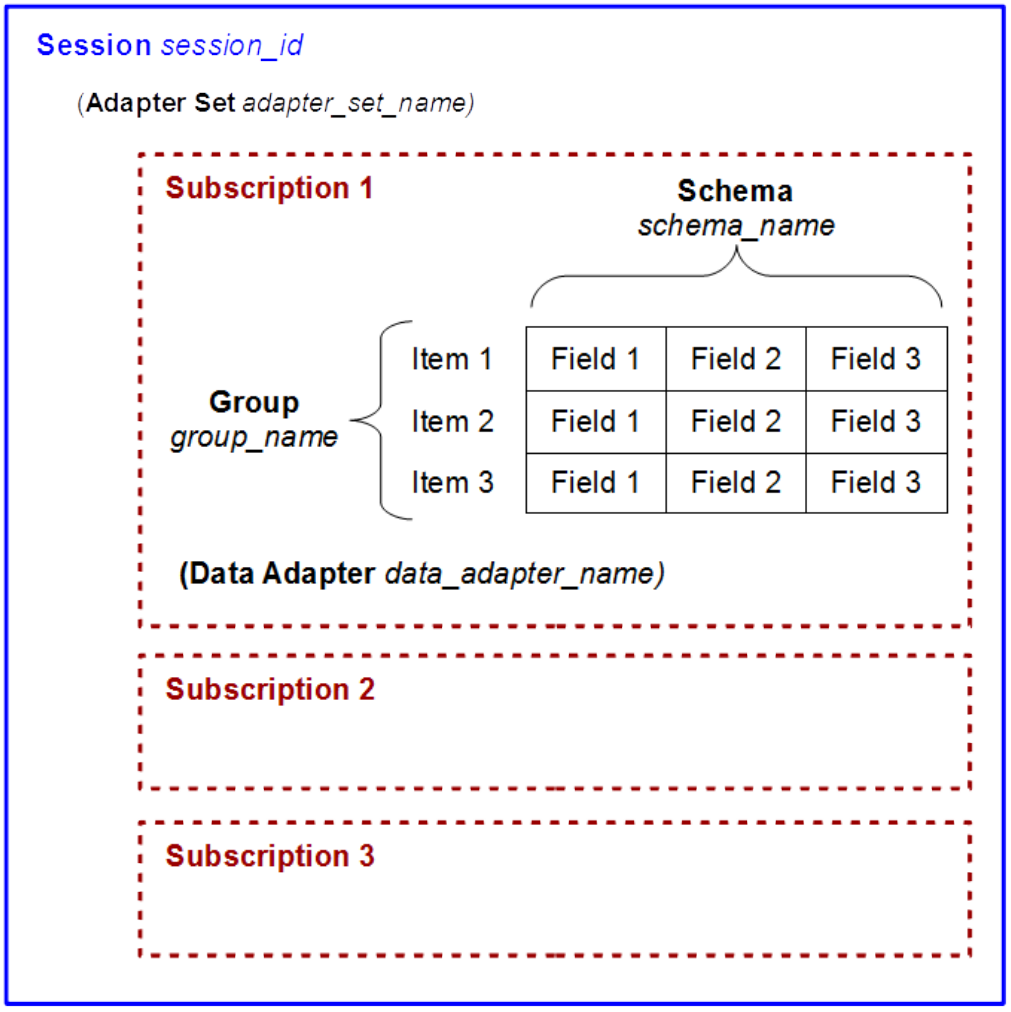


Figure 3- The logical hierarchy between a Session and an Item.

Adapter Set

It is made up of one Metadata Adapter and one or multiple Data Adapters. Multiple Adapter Sets can be plugged onto Lightstreamer Server.

Session

An entity created in Lightstreamer Server targeted to serve a specific Client. Each session is identified by a unique **sessionId**. Each session is associated to a single **Adapter Set**.

User

A user to whom a session is bound. Each session is bound to one and only one user.

Item

Information entity whose state is defined by a set of **fields** whose values can change over time. In other messaging systems, items are often referred to as "subjects", "topics", or "channels".

Each item is identified by a unique **item name** (whose syntax depends on the supplier Data Adapter). There exist both "general items", which can be subscribed by any user, and "personal items", each of which can be subscribed only by a specific user or by a group of users. This enables unicast messaging and multicast messaging, as opposed to broadcast messaging. The distinction between these categories of items is enforced by the Metadata Adapter, which can validate each subscription based on the user identity.

Examples:

- item name = "MSFT.NASDAQ" (a full quote of Microsoft stock on NASDAQ exchange)
- item name = "user123-prv-msg" (a private user message).

Field

A string pair comprised of a **field name** and a **field value**.

Examples:

- field name = "bid"; field value = "3.05" (the bid element of a stock quote)
- field name = "ask"; field value = "3.04" (the ask element of a stock quote)
- field name = "message"; field value = "Hello World!" (a message in a chat room)

Snapshot

The current full state of an **item**. For example, in MERGE mode, the snapshot is the set of **fields** the define the complete state of an item at that time.

ItemEvent

A set of **fields** sent from a Data Adapter to Lightstreamer Kernel to update the state of an **item** at a certain time.

ItemUpdate

A set of **fields** sent from Lightstreamer Kernel to a Lightstreamer Client to update the state of an **item** at a certain time.

UpdateEvent

A set of **ItemUpdates** sent from Lightstreamer Kernel to a Lightstreamer Client to update the states of all the subscribed items.

Field Schema

A set of **field names** that have been subscribed to by a Client in relation to an **item**. A Field Schema is identified either by an array of field names or by a unique string that should be known to the Client and to the Metadata Adapter.

Item Group

A set of **items** subscribed to by a Client with a common **field schema** and a common supplier **Data Adapter**. An Item Group is identified either by an array of item names or by a unique name that should be known to the Client and to the Metadata Adapter.

Subscription

An association among an **Item Group**, a **Field Schema**, and a **Mode**. It may also be referred to as a **Table** in the context of some client libraries.

Mode

The way the **itemEvents** of an **item** are handled in Lightstreamer Kernel for a Client. It can be: "**MERGE**", "**DISTINCT**", "**COMMAND**", and "**RAW**". The mode is indicated by the Client during subscription.

Selector

A predicate to be applied to the **itemEvents** in order to filter them. It can be associated to a **Subscription**, so that it will be applied for all **items** in the subscribed **Item Group**. The selector is identified by a symbolic name that should be known to the Client and to the Metadata Adapter, which is wholly responsible for the filtering.

3.2 Data Model and Subscription Modes

The role of a Data Adapter is to map the original *data model* of a Data Provider onto the Lightstreamer data model. As anticipated, the Lightstreamer data model is based on items, which are made up of fields. Items can be interpreted in different ways, depending on the **subscription mode**.

With **MERGE** subscription mode, an item represents a row in a table. Real-time updates to that item are used to update the contents of the cells (fields) for that row.

With **DISTINCT** subscription mode, an item represents a list of events. Real-time updates to that item are used to add lines to that list (where each line is made up of fields).

With **COMMAND** subscription mode, an item represents a full table. Real-time updates to that item are used to change the contents of that table, by adding rows, removing rows, and updating cells.

By combining these subscription modes, it is possible to map any kind of data. It is always advised to try to leverage the Lightstreamer fields as much as possible, so that the field-level delivery optimizations can take place. But there is total freedom on what items and fields should represent. So, if tabular data representation is not suitable and some complex data structure should be delivered, it is possible to embed it into a single field.

In addition to these modes, there exists **RAW** subscription mode, where no assumption is made on the nature of the data, leading to less optimization in data delivery.

The following sections delve into the details of each subscription mode.

3.2.1 MERGE Mode

Many data sources can be defined as filterable. **Data filtering** is a possibility offered by the intrinsic nature of some types of information. All information sources that over time generate updated versions of the same data, more or less frequently, can quite easily be subjected to filtering algorithms.

The following are examples of filterable data feeds:

- **Stock prices.** The price of a stock ("last price") varies continuously during the course of a stock market session. Once the market has generated a new price, the previous one automatically becomes obsolete. So, in the case where prices are generated with high frequency, one could decide not to communicate each and every variation. It is obviously indispensable to maintain the consistency of the data presented in the application, despite the filtering. For example, if on the same page there is the price of a covered warrant that usually gets updated once every hour, together with the price of a stock listed on the NASDAQ, which generates 30 updates a second, it is obviously unthinkable to lose the one update of the first in order to communicate more updates of the second.
- **Measurements (monitoring, metering, telemetry, etc.).** A probe of any sort (e.g. a physical probe to measure temperature or a software probe for network management) produces a whole series of measurements which are samples of the quantity that the probe is designed to monitor. These samples can be subsequently re-sampled to reduce their frequency even further.

Lightstreamer manages the filterable data feeds by means of the **MERGE mode**.

In this case, the source provides updates to a persisting state of an item. The absence of a field in an itemEvent is interpreted as "unchanged data". Any "holes" must be filled by copying the latest available value for each field.

It is important to note that filtering is not mandatory, even in MERGE mode (see §3.3.4).

Conflation (Merging)

Not all the itemEvents from the Data Adapter need to be sent to the Client (depending on throttling configuration). With conflation, if a previous itemEvent is discarded, the changes brought by it are still included in the newer itemEvents.

Let's imagine that each update is a small sheet divided into cells, with the current field values written in each cell. By referring to the stock price example, the fields are "last", "last size", "bid", and "ask". The sheets fall down one after another from the data source and stack on the ground, as shown in the following picture:

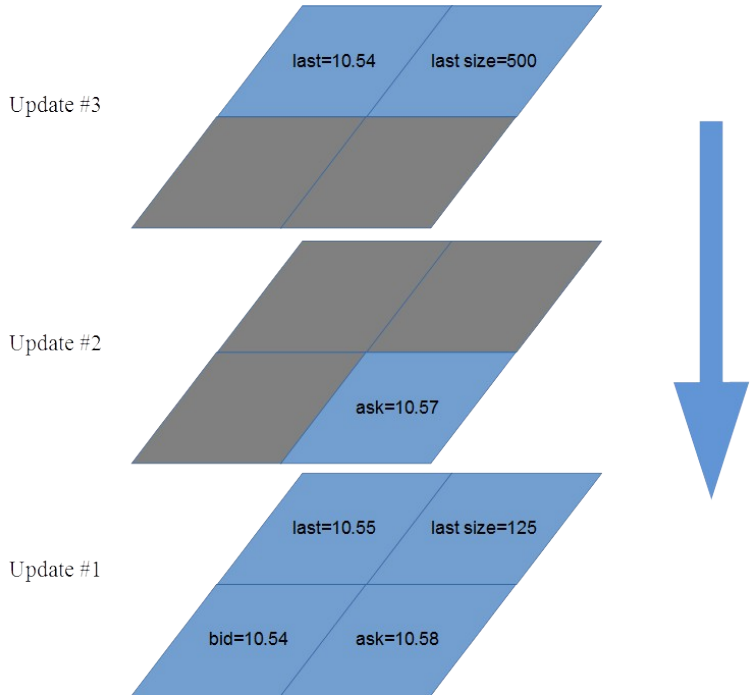


Figure 4 - The updates falling down and stacking on the ground.

Each update carries only the fields whose value has changed. The other cells are transparent. The first sheet (Update #1) carries all the field values. The second one (Update #2") carries only "ask". The third one (Update #3) carries only "last" and "last size".

Upon the first update, the current item state is given as follows:

Item = [last=10.55, last size=125, bid=10.54, ask=10.58]

When the second sheet stacks upon the first one, a new sheet is created, that carries "last", "last size", and "bid" from the first sheet and "ask" from the second sheet. This leads to an updated current item, whose state is given by merging old (black) and new (red) values:

Item = [last=10.55, last size=125, bid=10.54, ask=10.57]

This means that if for any reason (bandwidth constraint, frequency allocation, adaptive streaming, etc.) the first two updates cannot be sent to the client, only the resulting *merged* update ("sheet") will be sent.

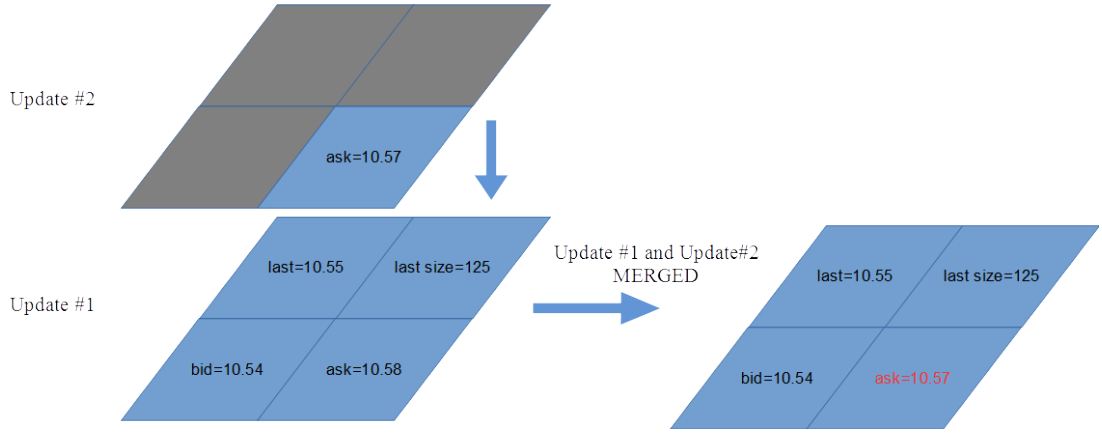


Figure 5 – The first and second updates are merged.

When the third sheet stacks upon the first two, a resulting sheet carrying "last" and "last size" from the third sheet, "ask" from the second sheet, and the "bid" from the first sheet is created. The final item state is:

Item = [last=10.54, last size= 500, bid=10.54, ask=10.57]

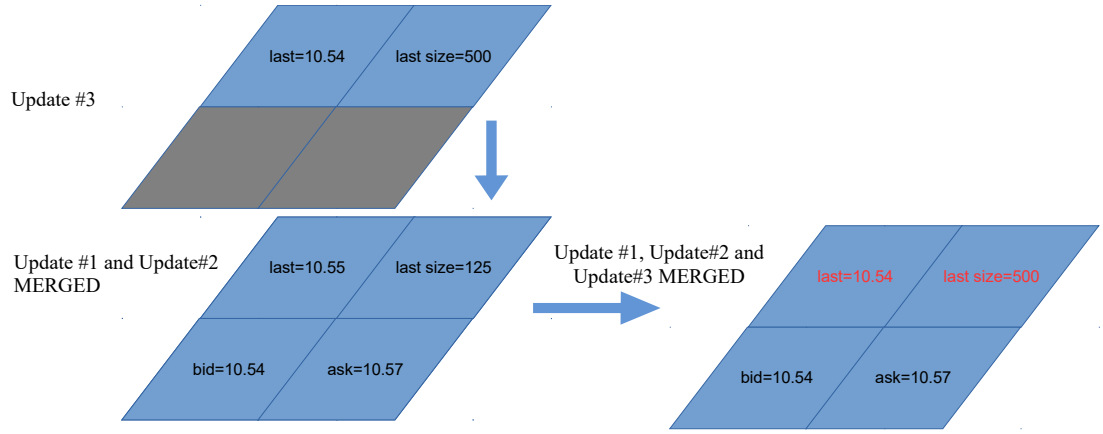


Figure 6– All the three updates are merged together.

In this case, the server has the possibility of sending one update only to the client instead of three. But as mentioned before, such filtering only occurs when actually necessary, based on several combined conditions that determine throttling.

3.2.2 DISTINCT Mode

There are data sources that do not allow conflation. In these cases, the feed does not produce data that replaces the previous one, but each event is distinct from the previous one.

The following are examples of distinct data feeds:

- **News headlines.** Press agencies generate fresh news items throughout the day. Typically, a real-time news visualization system shows a series of headlines in chronological order (from the latest to the oldest) which runs on receipt of each new headline. So when a fresh item of news is received, it does not eliminate the previous one; it gets listed alongside it.
- **Chat messages.** Messages in a chat room should follow one each other without being "conflated" or resampled.

Lightstreamer manages distinct data feeds by means of the **DISTINCT mode**.

In this case, the source provides events of the same type. The itemEvents coming from the Data Adapter must be sent to the client unchanged. DISTINCT data can be either unfilterable or filterable (see §3.3.4).

3.2.3 COMMAND Mode

There exist many application scenarios where it is required not only to push changes to an item content in terms of updating one or more fields, but even changes to the list of items itself, in terms of adding and removing rows. This is the so called "**meta-push**".

The following are examples of meta-push feeds:

- **Stock portfolio.** The value of stocks in a portfolio keeps changing based on the market, so some fields keep being updated. But, at the same time, selling or buying some stocks results in removing or adding rows to the portfolio table.
- **Chat participants.** The list of users in a given chat room keeps changing, due to users entering and exiting the room. The state of users in the room can change as well, for example from "active" to "idle".

Lightstreamer manages the meta-push feeds by means of **COMMAND mode**

In COMMAND mode, usually a single item is used for the whole list or table (e.g. the whole portfolio). The itemEvents are interpreted as **commands** that indicate how to progressively modify a list. In the schema there are two fields that are required to interpret the itemEvent.

1. The "**command**" field contains the command associated with the itemEvent, having a value of "ADD", "UPDATE" or "DELETE".
2. The "**key**" field contains the key that unequivocally identifies a line of the list generated from the Item.

The "key" and "command" fields must never be missing from an itemEvent.

In the following figure, a graphical representation of an Item handled in COMMAND Mode is shown:

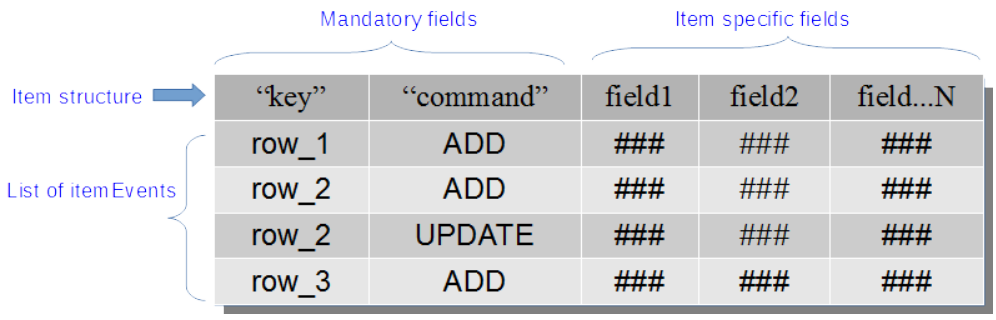


Figure 7 - Item in COMMAND Mode.

Not all the itemEvents coming from the Data Adapter need to be sent to the client (because filtering is possible). Successive events regarding the same "key" (but not necessarily received consecutively from the Data Adapter) can be merged according to the following rules:

- an UPDATE merges with previous ADD or UPDATE;
- a DELETE removes the previous ADD or UPDATE (when removing ADD, the DELETE itself is removed; when removing UPDATE, the DELETE remains);
- an ADD removes the previous DELETE and becomes an UPDATE.

For the itemEvents with commands of "ADD" or "DELETE" the "holes" (in the other fields) are propagated to the Client. For the itemEvents with commands of "UPDATE", the "holes" are filled by copying the values of the field corresponding to the last message of ADD or UPDATE with the same key.

Note that in COMMAND mode the "Delta Delivery" mechanism computes the delta of an update with respect to the previous update of the same item, which usually is not an update of the same key. This reduces the efficiency of the compression. To enforce Delta Delivery at key level, use Two-level Push (see below).

Two-level Push

To achieve a better separation of concerns, the COMMAND mode enables the so called "Two-level" push behavior too. In this way, it is possible to decouple the ADD/DELETE logic from the UPDATE logic, by means of two different kinds of items:

- the first-level item, based on COMMAND mode, which will manage the "first-level" subscriptions in terms of changes to the list of keys;
- the second-level items, which will manage the "second-level" subscriptions in terms of updates to the values for each single key.

Each time the Data Adapter pushes an "ADD" command for the first-level item with a new key to add a row, the client uses this information to automatically subscribe to an **underlying second-level item** in MERGE mode. Similarly, when a "DELETE" command is received to remove a row from the list, the client automatically unsubscribes from the underlying item. This mechanism is natively supported by some client libraries.

This way, the first-level item represents the full table (determines which rows are part of the table at every moment); for each row, there is a specific second-level item that provides the field values for such row.

The first- and second-level items may be supplied by different Data Adapters. For example, in the stock portfolio mentioned above, it is easier to have one Data Adapter managing the changes in the portfolio contents, and another Data Adapter managing the updates to the prices coming from the market.

3.2.4 RAW Mode

In RAW Mode, all the itemEvents coming from the Data Adapter must be sent to the client unchanged. The only constraint that applies is the bandwidth allocation. RAW mode acts as a "dumb pipe", where no aggregation of events is done.

Note that this constraint may pose restrictions also on the use of "diff" algorithms for "Delta Delivery". Only algorithms that ensure to rebuild the original value at text level (and not just at semantic level) will be enabled. This, for instance, disables the use of JSON Patch for RAW mode.

3.2.5 Compatibility of Modes

In principle, the same item can be subscribed to in more modes by the same client or by different clients. However, each item can only be handled in one among the MERGE, DISTINCT, and COMMAND modes, depending on the kind of source, which restricts the possibilities somewhat. On the other hand, every item can be handled in RAW mode. This gives rise to 4 possible logical families of items, as follows:

1. RAW
2. RAW | MERGE
3. RAW | DISTINCT
4. RAW | COMMAND

The expected modes for each Item have to be specified by the Metadata Adapter, so that client requests for other modes can be rejected. This is done by implementing the *modeMayBeAllowed* method, which must obey the above restrictions. Failing to do so will cause the Server to handle each item according to the mode specified in the first subscription request it receives for it, and to just ignore subsequent requests for conflicting modes.

The Metadata Adapter can further restrict the allowed modes for each Item, either overall or on a per-client basis. For instance, when either MERGE or DISTINCT or COMMAND Mode is supported, it could disallow RAW mode in order to enforce filtering.

3.3 Bandwidth and Frequency Management

3.3.1 Bandwidth Management

Each stream connection is characterized by a maximum bandwidth (**maxBandwidth**), which is an absolute and binding limit. The bandwidth limit is ensured even across consecutive polling connections.

The maxBandwidth depends on the user and the client and is derived as follows:

maxBandwidth = min(allowedMaxBandwidth, requestedMaxBandwidth)

where:

- **allowedMaxBandwidth** is the maximum bandwidth the Metadata Adapter allocates for a specific user's session.
- **requestedMaxBandwidth** is the maximum bandwidth the client can request at the opening of the stream connection.

It is possible to omit these restraints (or set them as "unlimited").

3.3.2 Frequency Management

Each item subscribed to by a client is characterized by a maximum frequency (**maxFrequency**) to send updated events to that client. The frequency limits are ensured even across consecutive polling connections.

The maxFrequency is determined as follows:

maxFrequenxy=min(allowedMaxFrequency, requestedMaxFrequency, grantedMaxFrequency)

where:

- **allowedMaxFrequency** is the maximum frequency the Metadata Adapter allocates to the item/user pair.
- **requestedMaxFrequency** is the maxmimum frequency the client can indicate at the moment of the subscription of the Item Group containing that item.
- **grantedMaxFrequency** is an upper limit on the frequency that may be imposed as a limitation in some specific editions of the product.

It is possible to omit these restraints (or set them as "unlimited").

3.3.3 Applying the Limits

Maximum bandwidth and maximum frequency constraints act on different levels: while the bandwidth constraint is applied to the whole session, the frequency constraint is applied to each item individually. Both constraints set an upper bound, which is dynamically managed by the Lightstreamer Server.

A Client can specify the requestedMaxBandwidth and/or the requestedMaxFrequency limits in order to protect itself from update rates too high to be sustainable. A client may also need to ensure that the itemUpdates for an item don't exceed a maximum frequency for application-level reasons (think to updates that are meant to be displayed in a cell). By setting a requestedMaxFrequency, the client will have the Server apply the limit before sending the updates, hence saving bandwidth. Moreover, a client may also specify the "unlimited" values to mean that the maximum actual parameters can be decided on the Server side.

On the other hand, the Metadata Adapter can specify limits (i.e. allowedMaxBandwidth and allowedMaxFrequency) in order to protect the Server from peaks of activity. While doing this, it has the opportunity of granting different service levels to different clients based on the involved user.

Finally, the license terms may impose limits (i.e. grantedMaxFrequency) that are only meant to force a reduced quality of service on limited versions of Lightstreamer.

When bandwidth and frequency limits are set, some ItemEvents originated by the Data Adapter may have to be *filtered*. This is done for each Item independently, and in different ways depending on the subscription mode. However, even with no limits set, filtering can occur, at Server discretion, in case of resource limitations or for uninformative updates (again, depending on the mode).

3.3.4 Filtered and Unfiltered Modes

With respect to the maximum frequency, the management Modes can further be specialized in "filtered" and "unfiltered" modes:

- The **unfiltered mode** is activated on the client side by requesting **unfiltered dispatching** at the moment of the subscription. This is accomplished by specifying the value "unfiltered" for the requestedMaxFrequency parameter. Such mode prevents filtering at all for an item and obviously implies requesting no frequency limits. However, the request is accepted only if the allowedMaxFrequency allocated by the Metadata Adapter for an item/user pair is "unlimited"; otherwise the user cannot subscribe the with unfiltered dispatching.
- The **filtered mode** is activated on the client side by specifying for the requestedMaxFrequency parameter a number or the value "unlimited" at the moment of the subscription.

The management of requestedMaxFrequency and allowedMaxFrequency is provided only for the following subscription modes: **MERGE**, **DISTINCT**, **COMMAND** (in COMMAND the frequency limits apply to the UPDATE commands sent for each key). For **RAW** mode, unfiltered dispatching (and so unfiltered mode) is always implied.

On the other hand, grantedMaxFrequency applies to all items, even if subject to unfiltered dispatching.

The following table shows a final resume of the allowed combined settings to determine the maxFrequency parameter:

Filtered/Unfiltered	requestedMaxFrequency	allowedMaxFrequency	Output maxFrequency
Filtered mode	Number	Number	min(requestedMaxFrequency, allowedMaxFrequency)
	"unlimited"	Number	allowedMaxFrequency
	Number	"unlimited"	requestedMaxFrequency
	"unlimited"	"unlimited"	Unlimited
Unfiltered mode	"unfiltered"	"unlimited"	Unlimited

Table 1 - Allowed combinations for maxFrequency parameter.

➔ An **online example** with full source code, showing dynamic changes to bandwidth and frequency is available at:
https://github.com/Lightstreamer/lightstreamer-example-BandwidthandFrequency-client-javascript

3.3.5 The ItemEventBuffer

For each item subscribed to in a session, an **ItemEventBuffer** (which is a FIFO buffer) is created in the InfoPump instance which is dedicated to that session (an InfoPump is a data structure allocated by the Lightstreamer Kernel for each session). More precisely, distinct itemEventBuffers are set up for each single subscription: for instance, if the same item is subscribed twice in the same session, two buffers will be prepared.

The itemEventBuffer contains the itemEvents extracted from the supplier Data Adapter that are still to be forwarded to the Client, because of bandwidth and/or frequency constraints. Only the requested fields are retained, according to the field schema indicated during subscription.

Each element will be sent to the Client as part of an updateEvent (except for the case when the buffer is full, more details further on).

An itemEventBuffer is characterized by the following parameters

- **mode**: mode of interpreting the new data in arrival (**RAW, MERGE, DISTINCT, COMMAND**).
- **dispatching switch: "filtered" or "unfiltered",** as described in paragraph 3.3.4
- **bufferSize**: maximum buffer dimension (whose value can also be "unlimited"); it can be specified for MERGE and DISTINCT modes, unless unfiltered dispatching is in place. It refers to the number of events waiting for dispatching and is relative to each single subscription.

The choice of mode is affected by the nature of the data and is indicated during item subscription. It impacts on the way events can be packed into updateEvents and on the way buffer size limits apply.

The unfiltered mode presupposes that all events participate in the evolution of the client state and no event can be suppressed. In this case, more events can be released in a single updateEvent. However, grantedMaxFrequency and bandwidth limits can still lead to the need for a buffer, in case of event bursts. For this reason, when in unfiltered mode, an **unlimited** bufferSize is always granted.

In all the cases where the bufferSize is unlimited, the Kernel is free at runtime to leverage a "robustness mechanism" which puts a limit on the maximum dimension of the buffer. This means that the Kernel can block a further growth of buffer size to protect itself, by discarding events. When this happens for items in RAW mode or in unfiltered dispatching, the Kernel must also signal to the Client that one or more events were lost; it is up to the Client to decide whether to abort the subscription to that item, or to maintain it with the knowledge that not all events are being received.

In general, when the buffer is full and a new itemEvent arrives from the supplier Data Adapter, this will still enter the buffer, forcing the rejection of the element at the top of the buffer (the first element that entered, i.e. the oldest).

3.3.6 ItemEventBuffer in RAW Mode

In RAW Mode, more than one itemUpdate can be release in a single updateEvent, depending on bandwidth limits. The bufferSize is always **unlimited.** Any data loss is signaled to the Client.

3.3.7 ItemEventBuffer in MERGE Mode

In **MERGE filtered mode**, only one itemUpdate event can be released in a single updateEvent (unless the allowed maxFrequency is very high or "unlimited"). This way, as a matter of fact, the frequency of releasing of the updateEvents (which depends on various factors, including a maxBandwidth limit and the network roundtrip time) acts as a further frequency constraint and may enforce further filtering.

A buffer dimension greater than 1 allows it to absorb event bursts without overlaying them, in cases where it is preferred to receive as many updates as possible without filtering. Exceeding the buffer size limits leads to the silent suppression of some events, applying the merging algorithm. The default size is 1.

In **MERGE unfiltered mode**, more than one itemUpdate can be released in a single updateEvent. In this case the buffer size must be **unlimited**. Any data loss is signaled to the Client.

3.3.8 ItemEventBuffer in DISTINCT Mode

In **DISTINCT filtered mode**, only one itemUpdate event can be released in a single updateEvent (unless the allowed maxFrequency is very high or "unlimited").

A buffer with a size greater than 1 allows event bursts to be absorbed, in order to send them to the Client in such a way that it is compatible with the assigned maximum frequency and without loss. Exceeding the buffer size limits leads to the silent suppression of some events. The default size is unlimited.

In **DISTINCT unfiltered mode**, more than one itemUpdate can be released in a single updateEvent. In this case the buffer size must be **unlimited.** Any data loss is signaled to the Client.

3.3.9 ItemEventBuffer in COMMAND Mode

In **filtered COMMAND mode**, for each "key", the events belonging to the "key" are released in a way similar to the MERGE mode with a buffer size of 1. Exceeding this 1-sized buffer leads to event "matching" and merging. Hence, as said, in COMMAND mode frequency limits apply to the various keys independently. If the "robustness mechanism" must be leveraged, it will try to filter out events of "UPDATE" type; if any events of "ADD" or "DELETE" type must be discarded, then this is also signaled to the client, as some problems may occur, particularly during snapshot management (see below). However, note that as long as the number of active keys is less than the protection limit, the buffer cannot grow beyond the limit, regardless of the event rate.

In **unfiltered COMMAND mode**, each single event is granted not to be suppressed and is delivered to the client as is, unless the buffer has grown over the protection limit; in such case, any data loss for "ADD", "UPDATE" and "DELETE" events is notified to the Client.

On the other hand, the unfiltered COMMAND mode does not ensure that events pertaining to different keys are sent to the clients in the same order in which they are received from the Data Adapter (this may be particularly evident when grantedMaxFrequency limits, which apply to the various keys independently, are involved). In fact, usually, one should expect that updates for different keys are not related to one another.

Ordering

There are cases in which strict ordering could be useful; one case is handling ordered lists (classifications, charts, results, etc.). If the sort criterion were based on a field representing a "score", then we would only need to sort the column of the table on the client side. If, however, the sort criterion were more complex, then the client could be sent a field that contains the current position in the list. But the move of a single element could imply the need for an UPDATE for each line of the table and this could be very heavy and redundant. To optimize the handling, we can assume that the client will be sent only the UPDATE for the element that is moved (or that enters or exits), leaving the client the task of modifying the logical and physical position of the other elements. This would not work properly if update events could be filtered out or even flipped across different keys.

At the current stage, the further restriction of preserving update order across keys for unfiltered COMMAND mode can only be activated on a Server-level scope through a proper configuration flag. However, in that case, any grantedMaxFrequency limit would apply to the item as a whole; in fact, in cases like the one above, the benefits of using COMMAND mode instead of RAW mode would be minimal.

3.3.10 Resume of Settings for the ItemEventBuffer

The table below summarizes the possible combined settings as detailed in the previous paragraphs:

Mode	Buffer Size	ItemUpdate per UpdateEvent relative to the same item
RAW	Unlimited	1..N (but 1 with grantedMaxFrequency)
MERGE filtered	1..unlimited (default =1)	1 (but 1..N if frequency is unlimited)
DISTINCT filtered	1..unlimited (default = unlimited)	1 (but 1..N if frequency is unlimited)
COMMAND filtered	Unlimited	1..N with different keys
MERGE unfiltered	Unlimited	1..N (but 1 with grantedMaxFrequency)
DISTINCT unfiltered	Unlimited	1..N (but 1 with grantedMaxFrequency)
COMMAND unfiltered	Unlimited	1..N (but only with different keys with grantedMaxFrequency)

Table 2 - Settings for the ItemEventBuffer.

As said, if a grantedMaxFrequency limit is set because of the use of a limited edition, it applies to all itemEventBuffers, regardless that unfiltered dispatching is in place. Moreover this also extends to all itemEventBuffers the limitation of 1 ItemUpdate per UpdateEvent relative to the same item (or to the same "key", for unfiltered COMMAND).

Consider that the presence of a tight maxBandwidth or grantedMaxFrequency limit poses further and possibly strong restrictions to the ItemUpdate throughput, even on items in RAW or unfiltered mode.

In particular, for RAW or unfiltered items, the internal buffer may grow and since the bufferSize is unlimited, the ItemUpdates may be released with significant delays and the "robustness mechanism" may eventually take place. It is an application responsibility to ensure that the update rate from the supplier Data Adapter cannot exceed the outbound rate.

3.3.11 Considerations on the Usage of the Buffer

In normal cases, the Server should not need to use the buffer, as it should be able to send all events in real time. When subscribing to an item in filtered mode, the subscription settings should also request a small buffer size.

On the other hand, if a subscription is made in a mode that does not allow filtering, then the event rate for the item should not be higher than the expected dispatch rate. Hower, if the incoming event rate for an item is steadily higher than the dispatch rate, then the buffer usage grows with negative impacts both on dispatching delays and on Server heap memory usage.

3.3.12 Selectors

A predicate on the itemEvents (called a Selector) can be associated to the ItemEventBuffer. It is applied before itemEvents enter the ItemEventBuffer and can prevent some itemEvents to enter the buffer.

Typically, selectors can be used to extract subsets of events, based on the "key" value in COMMAND mode or on some special attribute in RAW and DISTINCT modes. Selectors may also be used to suppress unimportant updates in all modes. Selector can be applied on a per-user basis.

NOTE: In COMMAND mode, events suppression should be consistent with the key lifecycle, ensuring that ADDs and DELETEs on the same key are interlaced (always starting with an ADD, if the snapshot has been requested) and that UPDATEs follow ADDs for the same key, but not DELETEs.

NOTE: In MERGE mode, the suppression of a snapshot event may cause the client to receive the first event for the Item only some time after the subscription, despite the snapshot information has been requested.

3.4 The Preprocessor

The Preprocessor has the task of filling the empty fields in the events, maintaining snapshots and performing Prefiltering.

3.4.1 Snapshot

When a client subscribes to an item, it may request the **initial snapshot**, that is, a way to bootstrap the current item state without having to wait for the asynchronous real-time updates.

Only for the items handled in MERGE, DISTINCT, and COMMAND modes, the Preprocessor can maintain snapshots and make them available to the InfoPump when it requests them (when the Client requests a snapshot to the InfoPump). RAW mode does not support the snapshot.

MERGE: The snapshot is the current value for each field. Consisting of an itemEvent which is progressively modified or extended by itemEvents coming from the Data Adapter.

DISTINCT: The snapshot is the recent history of updates to the item. Consisting of a list with a predefined maximum length, which contains the latest itemEvents received from the supplier Data Adapter. All the events belonging to the snapshot are sent to the client in an *unfiltered dispatching* fashion, despite the frequency and buffer size limits operating for normal updates.

COMMAND: The snapshot is the shortest sequence of commands needed to build the current state of the table from scratch (where the full table is represented by a single item). it consists of a set of itemEvents, one for each "key". Each one of these itemEvents is progressively modified or extended by "UPDATE" itemEvents that come from the supplier Data Adapter. The set itself is progressively extended or shortened by the "ADD" or "DELETE" itemEvents coming from the Data Adapter. If the itemEvents in the set are sent with the "ADD" command, they allow the Client to reconstruct the list.

Note that the snapshot is not a simple playback of the commands received so far, but an optimization of them. Moreover, the order of the keys involved in the snapshot itemEvents is unspecified. This should be taken into consideration, in particular, when unfiltered dispatching is desired.

When the Preprocessor implements the subscription to a new item toward the supplier Data Adapter, if the snapshot management for the item is supported by the Data Adapter, then the Data Adapter must use an asynchronous call-back to immediately send the snapshot, before any updates. However, if the snapshot for the item is not available (because the Data Adapter does not support it), the Preprocessor behaves as though the state of the item at the time of the subscription was empty.

The Client can indicate, at the moment of the subscription to a particular item (provided that it is of the type MERGE, DISTINCT or COMMAND), whether it intends to receive the snapshot first or to receive only the updates (as a default, the snapshot is not requested). In order to facilitate the handling of this differentiation, when the Preprocessor implements the subscription to a new item toward the supplier Data Adapter, it needs to distinguish the snapshot from successive updates utilizing a flag in the asynchronous call-back.

The snapshot returned by the Data Adapter should be calculated in the same way as the Preprocessor does. If this is not the correct way to determine the initial state of the item, then the snapshot for this particular item is not supported by Lightstreamer Server; the snapshot should not be requested by the clients and should not be supported by the Data Adapter at all. A typical situation of this kind is an item that represents an ordered list in which the new position of a "key" is carried by a field (it was described in the previous discussion of the unfiltered COMMAND mode); for such an item, the order and the contents of the received "ADD", "UPDATE" and "DELETE" events is important to maintain the ordering of the "keys" within the snapshot, while Lightstreamer Server Preprocessor ignores these information.

3.4.2 Prefilter

Each ItemEventBuffer implements an event filter, as a function of the assigned maximum frequency. In cases where the external Data Provider generates events with a frequency far greater than that sent to the Client, *only for the items handled in MERGE or DISTINCT mode with filtered dispatching* it is possible to implement a Prefiltering of the data at Preprocessor level, common to all users, in order to avoid a heavy task of filtering of each single ItemEventBuffer. In this way each ItemEventBuffer applies its own filtering policy on a set of data already "refined", with a notable increase in server performance.

The **MERGE Prefilter** works like an ItemEventBuffer with a size of 1 and of MERGE type, characterized by a maximum send frequency (**preFilterFrequency**).

The **DISTINCT Prefilter** works like an ItemEventBuffer with a size of 1 and of DISTINCT type, characterized by a maximum send frequency (**preFilterFrequency**). The dimension is kept at 1 in order to avoid delays on the sending of data.

The decision whether to apply or not the Prefiltering for each item is up to the Metadata Adapter, as well as the choice for preFilterFrequency.

NOTE: if a preFilterFrequency for an item is set, then *the item cannot be subscribed with unfiltered dispatching* in MERGE or DISTINCT mode; it can, however be subscribed in RAW mode.

3.4.3 Preprocessor and Modes

The following table summarizes the allowed behaviors of the Preprocessor (with respect to snapshot management, prefiltering, and filling) for the different subscription modes.

Mode	Allow Snapshot	Allow Prefilter	Allow Filling
RAW	No	No	No
MERGE	**Yes**	**Yes**	**Yes**
DISTINCT	**Yes**	**Yes**	No
COMMAND	**Yes**	No	**Yes**

Table 3 - Allowed behaviors of the Preprocessor.

Filling means that the Preprocessor will fill any empty field coming from the Data Adapter with the latest available value for such field.

Filling is a logical process. It does not imply that when any field is unchanged across subsequent updates in MERGE mode, each time the same field will be redundantly delivered to the Client. The network protocol of Lightstreamer is highly optimized and the whole system guarantees the data coherency while saving on used bandwidth.

3.5 Roles in the Choice of Parameters

Different actors contribute in the choice of the previously discussed parameters. For each session the actors involved in the choice are the Client, the Kernel, the Data Adapter and the Metadata Adapter. The following table summarizes the different roles of the actors in the choice of each parameter.

Parameters	**Client**	**Kernel**	**Data Adapter**	**Metadata Adapter**
maxBandwidth	can ask	-	-	must decide
maxFrequency	can ask (only for MERGE, DISTINCT, COMMAND modes)	Can limit furtherly, based on license terms	-	must decide
unfiltered dispatching	can ask (only for MERGE, DISTINCT, COMMAND modes)	must refuse the request, if not allowed	-	must decide
Mode	must ask	must refuse the request, if not allowed	-	must decide
bufferSize	can ask (only for MERGE, DISTINCT)	need not follow the decision	-	must decide
Snapshot	can ask (only for MERGE, DISTINCT, COMMAND modes)	-	should collaborate by sending the initial snapshot	-
Snapshot length in DISTINCT mode	can ask (only for DISTINCT mode)	-	should collaborate by sending enough data for the initial snapshot	must decide the DISTINCT history length
preFilterFrequency	-	-	-	must decide

Table 4 - Choiche of parameters.

NOTE: The Client can ask for different bandwidth and item frequency at a later time; the request is reexamined by the Metadata Adapter.

NOTE: Metadata Adapter decisions can be overridden through JMX settings. Then, in case of a request for different bandwidth or item frequency by the Client, the last JMX settings are used for the decision.

4 The Adapters

Each Lightstreamer session requires the presence of an Adapter Set, which is made up of one Metadata Adapter and one or multiple Data Adapters. The roles of these two types of Adapters are very different and enable for a good separation of concerns.

A Data Adapter only knows about individual items. It has no knowledge about Subscriptions, Item Groups, and Field Schemas. Furthermore, it has no notion at all of users.

A Metadata Adapter knows about Subscriptions, users, Item Groups and Field Schemas. It has a very fine-grained control over the life cycle of user sessions, including authentication, authorization and entitlement.

This chapter shows some sequence diagrams to help clarify the typical workflows of the interactions between Lightstreamer Server (the Kernel) and a Metadata Adapter, and between Lightstreamer Server (the Kernel) and a Data Adapter.

4.1 The Metadata Adapter

Figure 8 illustrates the typical workflow generated by a user's activities with respect to the session-related Metadata Adapter, from authentication (session start) to log off (session end). (Adapter initialization is not shown).

The complete specification of the methods invoked bidirectionally between the Server and the Metadata Adapter is available under the form of "API reference" within the Adapter SDKs. The interactions between the Client and the Server use a logical notation and does not correspond directly to the physical network protocol employed.

Authentication. When creating a new Lightstreamer session, the Client can pass the Server some credential information (two generic strings can be used, conventionally named "user" and "password"; in addition, cookies, custom HTTP headers, and client-side SSL/TLS certificates can be used). The Server validates the credentials against the Metadata Adapter, that throws a specific exception if the validation fails. If the user is authenticated, a LS_session token is sent back to the Client, that will be used for all subsequent interactions in order to identify the session.

QoS, Authorization, Entitlement. The Metadata Adapter is responsible for assigning the quality of service to each user (in terms of bandwidth, frequency, etc.). It also interprets the "metadata" coming from the Client (Item Group and Field Schema names) and transform them into a form that can be used with the proper Data Adapter (that is, a list of item names and a list of field names). Furthermore, for each request coming from the Client, it should grant it or deny it. This applies to each subscription request and to the properties of each subscription (subscription mode, buffering, etc.).

Log off. The Metadata Adapter is notified when a session is closed (usually due to the user closing the client application).

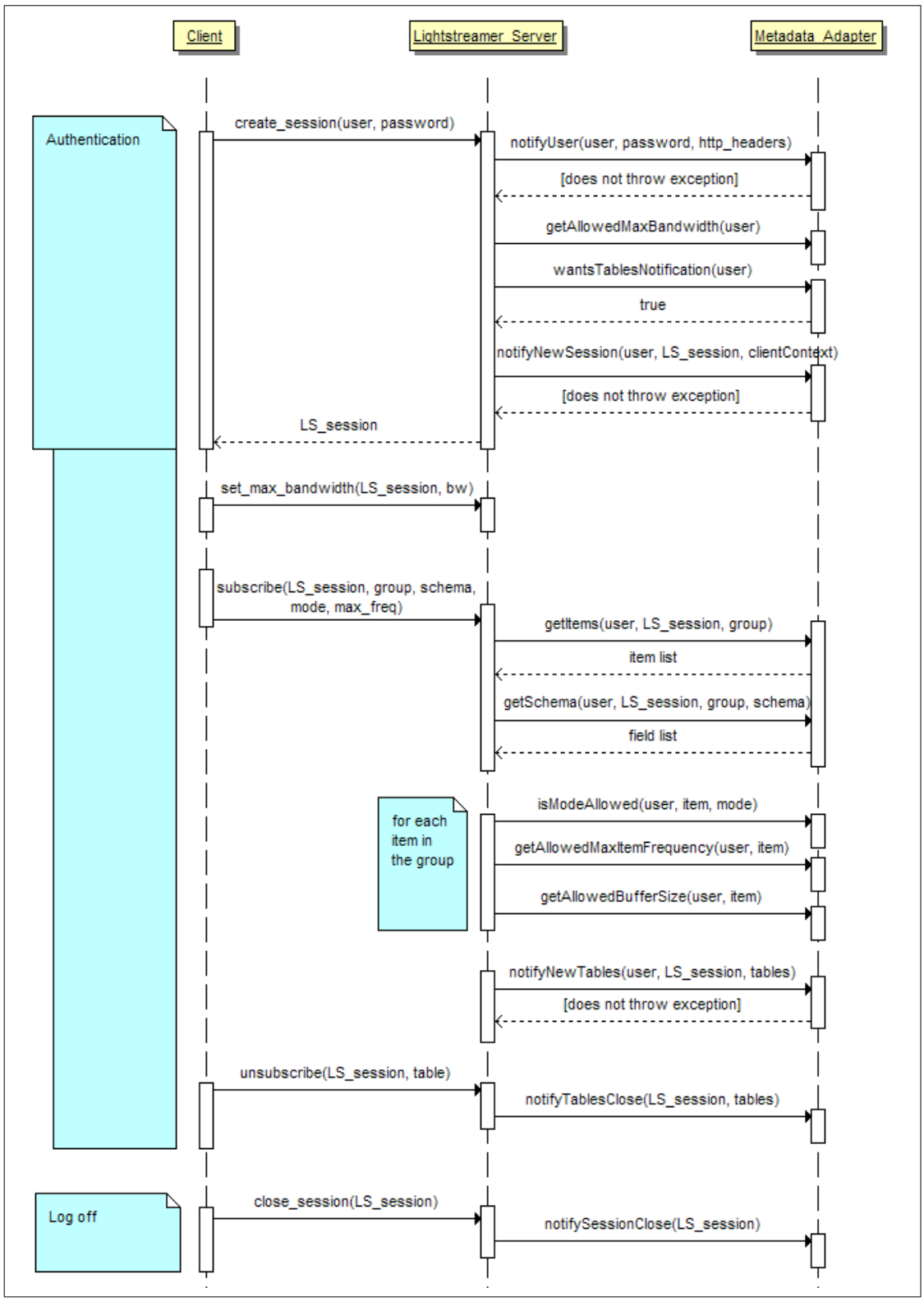


Figure 8- A typical Metadata Adapter workflow for a user.

Figure 9 focuses on the **authorization** process and illustrates the best practice used in typical scenarios. The actual authentication is usually handled by a legacy application server, irrespective of Lightstreamer. Some sort of session_id is sent back to the Client (through cookies, response payload or any other technique). When the Client creates the Lightstreamer session, instead of sending again the full credentials (usually involving a password) to Lightstreamer Server, it sends just the session_id (or the username and the session_id). The Metadata Adapter is passed this information and validates the session_id against the application server that generated it (or a database or whatever back-end system).

The advantage of this practice is that the user's password is never used outside the legacy system. Also, since the communication between the Client and Lightstreamer Server can leverage HTTPS, the session_id can be encrypted, so that it cannot be intercepted on the network.

This is just an example, because the Metadata Adapter is usually implemented in a custom way for each project, when integrating Lightstreamer into an existing or a new system.

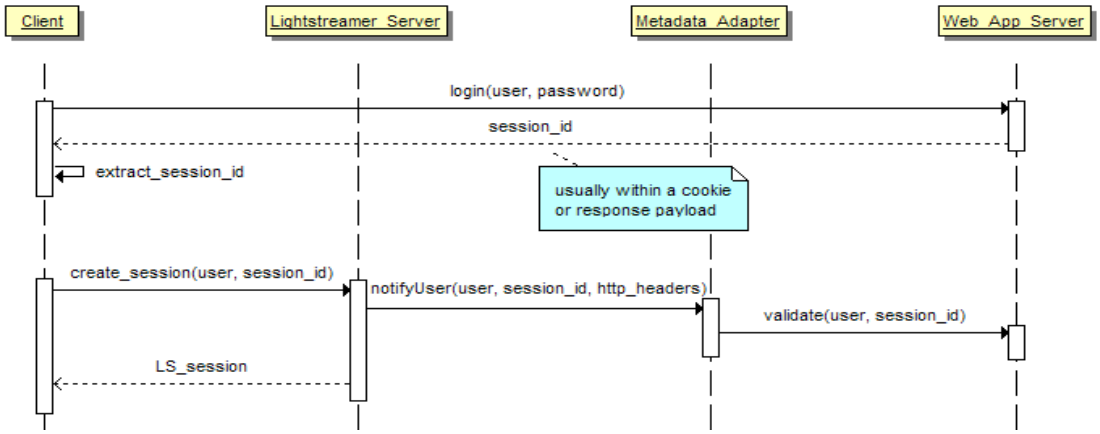


Figure 9- Authentication through the Metadata Adapter and an external server.

4.1.1 Authentication and Authorization Examples

A simple example showing authentication and authorization in action is available online:

- ➔ Adapter code: https://github.com/Lightstreamer/Lightstreamer-example-AuthMetadata-adapter-java
- ➔ Client Code and live demo: https://github.com/Lightstreamer/Lightstreamer-example-AuthMetadata-client-javascript

4.2 The Data Adapter

Figure 10 illustrates the typical workflow with respect to a Data Adapter. At Server's start-up the Data Adapter is initialized and a listener is passed for all subsequent asynchronous callbacks (that is, from the Data Adapter to the Kernel). When a new subscription targeted to this Data Adapter is received by the Server, after extracting the individual items through the session-related Metadata Adapter (see the previous section), the Server activates the new item subscription on the Data Adapter. Before that, the Server needs to know if the Adapter is able to provide an initial snapshot for an item. After the subscription has been activated, the Data Adapter will start sending back asynchronous events. The Server will use these updates to generate events for the Client. When no more Clients are subscribed to a given item, that item is unsubscribed form the Data Adapter too.

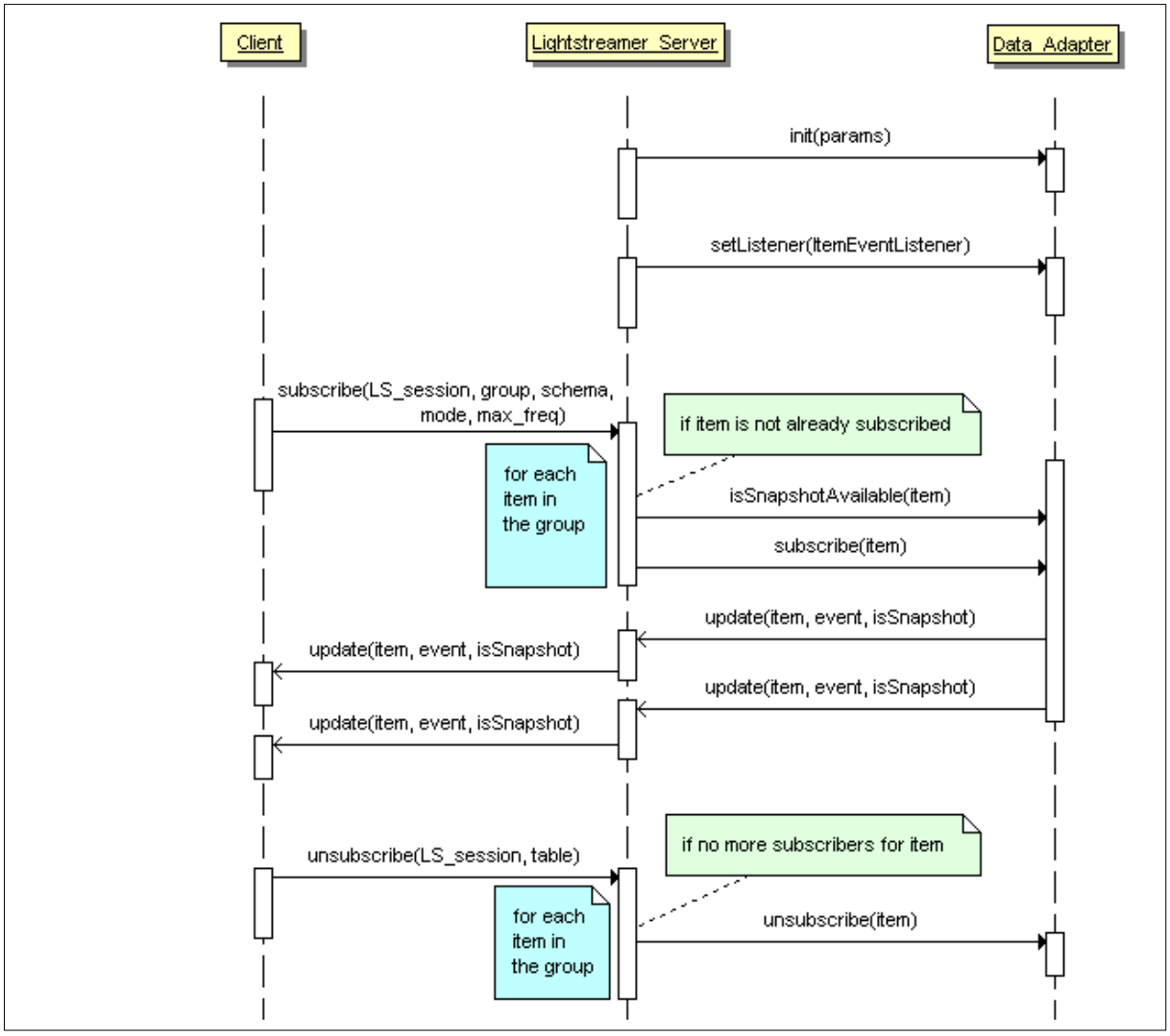


Figure 10- Data Adapter general workflow.

Figure 11 shows a concrete example. Two Clients are involved. Client B subscribes to item "item10", requesting a schema comprised of the fields "bid" and "ask". The Server will start serving that item after activating the subscription on the Data Adapter. Notice that the Data Adapter, as a first event, sends back a full snapshot for "item10". When Client A subscribes to "item10", the Server is able to serve it (including the initial snapshot) without asking anything to the Data Adapter. At this point the Server delivers the updates on "item10" to both the Clients. When both the Clients unsubscribe from "item10", the unsubscription is propagated to the Data Adapter, that should stop injecting events pertaining to "item10". When a Client subscribes to "item10", all the process starts again from scratch, that is, the Data Adapter receives the subscription and must send back the initial snapshot, followed by the real-time updates.

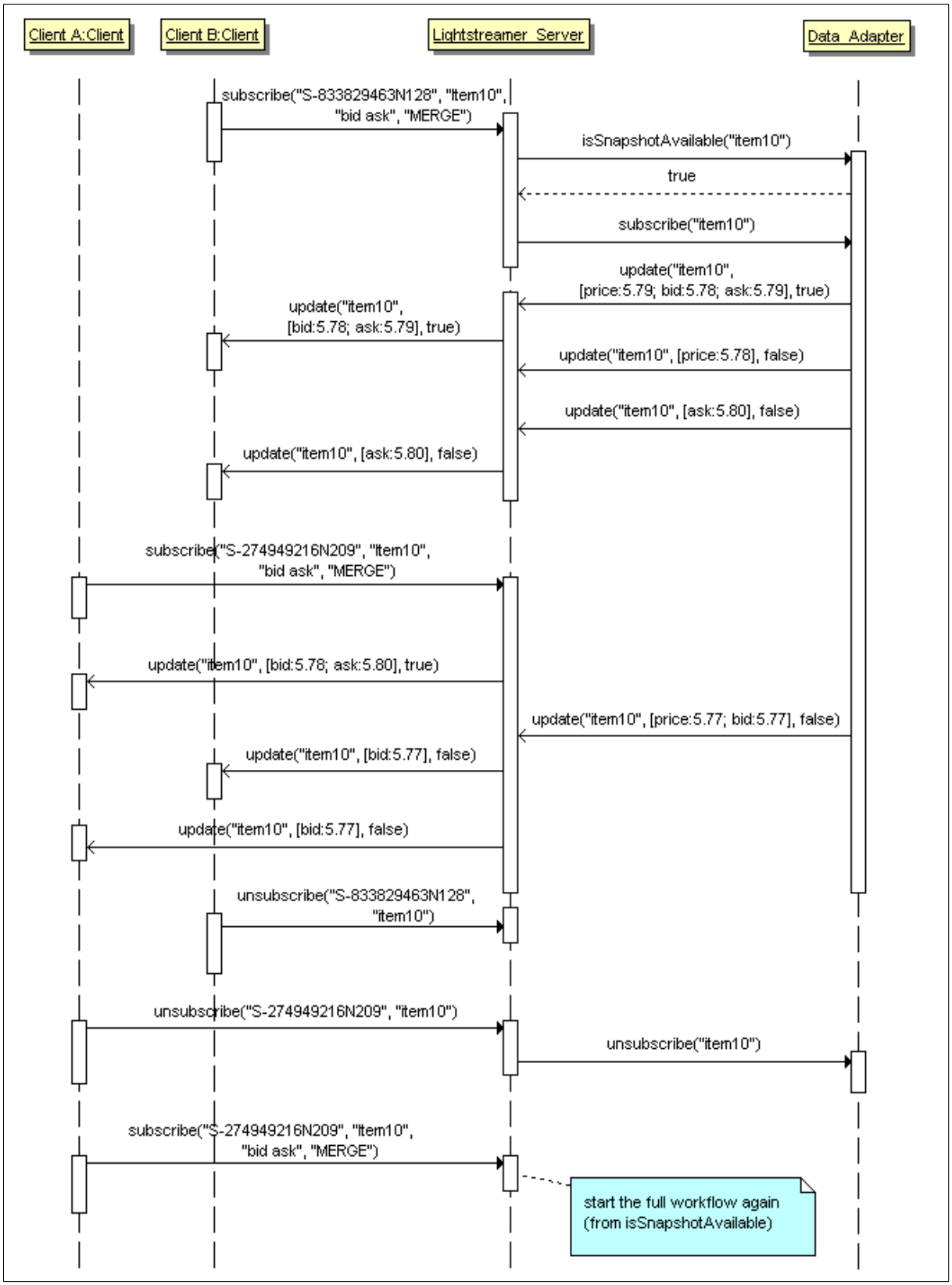


Figure 11- A concrete scenario of a Data Adapter workflow

The Data Adapter has the opportunity to enforce, for each Field of a subscribed Item, suitable "diff" algorithms (among the available ones) that should be tried for "delta delivery" purpose. Field "diff" information can be specified, through a dedicated message, after the item subscription and before sending the first data for the field.

4.2.1 An Internal Data Adapter: The Monitoring Data Adapter

The Server provides a special Data Adapter, the monitoring Data Adapter, which can be included in one or multiple custom Adapter Sets, by specifying the reserved "MONITOR" name in place of the Data Adapter class in the Adapter Set configuration.

This allows the clients to request internal Server statistics and monitoring data along with other application data. This is the same Data Adapter used by Monitoring Dashboard to provide real-time statistics on server's internals.

The custom Metadata Adapter included in the Adapter Set should identify requests directed to the monitoring Data Adapter and operate any desired access restrictions to these data.

The following items are supplied:

- **monitor_statistics**

  Internal load statistics, provided in MERGE mode without snapshot support.

  The available fields are the following:

  - **MEMORY.TOTAL** The total memory (expressed in Bytes) allocated for the heap.
  - **MEMORY.FREE** The free memory (expressed in Bytes) in the heap.
  - **THREADS** The total number of threads in the Java virtual machine.
  - **POOL.SIZE** The global number of threads belonging to one of the thread pools used by Lightstreamer Server to manage the various processing stages. This value is the sum of POOL.ACTIVE and POOL.WAITING, but for possible further threads currently in a transition phase.
  - **POOL.ACTIVE** The total number of threads belonging to one of the thread pools used by Lightstreamer Server which are currently executing a task.
  - **POOL.WAITING** The total number of threads belonging to one of the thread pools used by Lightstreamer Server which are currently unused and waiting for a new task to be enqueued.
  - **POOL.QUEUE** The total number of tasks waiting to be processed by any of the thread pools used by Lightstreamer Server. Note that the AUTHENTICATION and MSG pools are always defined and are used to invoke callbacks that are expected to behave asynchronously. For this reason, the queue count reported for these pools includes the requests currently in asynchronous execution on the Metadata Adapter.
  - **POOL.QUEUE_WAIT** The average waiting time in milliseconds experienced by the tasks scheduled for processing by any of the thread pools used by Lightstreamer Server.
  - **NIO.WRITE_QUEUE** The number of socket write operations currently waiting to be selected by the NIO selector.

- **NIO.WRITE_QUEUE_WAIT** The waiting time, in milliseconds, averaged over all the socket write operations currently waiting to be selected by the NIO selector.
- **NIO.WRITE_SELECTORS** The number of selectors and related threads currently used to handle write operation tasks.
- **NIO.TOTAL_SELECTORS** The total number of selectors and related threads currently in use.
- **CLIENTS.CONNECTIONS** The number of current Connection sockets; it comprises Stream, Polling, Control, Web Server and unrecognized requests.
- **CLIENTS.MAX_CONNECTIONS** The maximum number of concurrent Connection sockets reached in the life of the Server; it comprises Stream, Polling, Control, Web Server and unrecognized requests.
- **CLIENTS.SESSIONS** The number of current Sessions. Prestarted Sessions are not considered for this and the following session statistics. By **prestarted** Session we mean a session for which no related requests have been received yet. Binding a session with a new connection also gets it out of the prestarted state.
- **CLIENTS.MAX_SESSIONS** The maximum number of concurrent Sessions reached in the life of the Server.
- **CLIENTS.REFUSED_SESSIONS** The total (cumulative) number of Requests for opening new Sessions that have been refused in the life of the Server because of load limits, including thread pool queue limits.
- **CLIENTS.STREAMING_SESSIONS** The number of current Sessions that are associated with a Stream Connection.
- **CLIENTS.POLLING_SESSIONS** The number of current Sessions that are associated with Polling Connections.
- **CLIENTS.ITEM_SUBSCR** The total number of currently active Item subscriptions to Items. The value is cumulated over all client sessions, hence a single item, subscribed by multiple clients, ore even multiple times by the same client, will be counted multiple times.
- **CLIENTS.MAX_ITEM_SUBSCR** The maximum number of concurrently active subscriptions to Items reached in the life of the Server.
- **ITEMS.TOTAL** The total number of currently subscribed Items, covering all the Data Adapters from all the Adapter Sets plugged in the Server.
- **ITEMS.EVENTS_SEC** The frequency of inbound events flowing from all Data Adapters, within any Adapter Set, into LS Kernel in the last sampling period. It is expressed as events per second.
- **ITEMS.FILTERED_EVENTS_SEC** The frequency of outbound events flowing from the Preprocessor of LS Kernel in the last sampling period. It is expressed as events per second.
- **UPDATES.EVENTS_SEC** The frequency of outbound events flowing towards the clients. The value is cumulated over all client sessions, hence a single event produced by the Data Adapter could trigger multiple updates on one or multiple sessions. It is expressed as updates per second.
- **UPDATES.TOTAL_IN** The total (cumulative) number of inbound events flowing from all Data Adapters, within any Adapter Set, into LS Kernel.

- **UPDATES.TOTAL_OUT** The total (cumulative) number of outbound events flowing towards the clients. The value is cumulated over all client sessions, hence a single event produced by the Data Adapter could trigger multiple updates on one or multiple sessions.
- **UPDATES.TOTAL_LOST** The total (cumulative) number of updates lost. Events can only be lost because of buffer size restrictions posed for safety purpose. This applies to Items subscribed to in RAW mode, or in any mode with unfiltered dispatching specified. This also applies to items subscribed to in COMMAND mode with filtered dispatching, restricted to "ADD" and "DELETE" events only (note that those events can also be filtered through matching). Any lost event is also notified to the client.
- **BANDWIDTH.TOTAL_BYTES** The total (cumulative) number of bytes sent in the life of the Server (for Stream and Polling Connections only). The count includes all HTTP contents sent.
- **BANDWIDTH.CURRENT** The current global outbound bandwidth used by the Server (for Stream and Polling Connections only) in the last sampling period. It is expressed as Kilobits per second and it is comprised of network overheads.
- **BANDWIDTH.HIGHEST** The maximum global outbound bandwidth used by the Server in its life (for Stream and Polling Connections only). It is expressed as Kilobits per second and it is comprised of network overheads.
- **BANDWIDTH.ALLOCATED** The current outbound allocated bandwidth, calculated as the sum of the maximum outbound bandwidths allocated by the Server for each Session. It is expressed as Kilobits per second and it is comprised of network overheads.
- **BANDWIDTH.CLIENT_AVERAGE** The average outbound bandwidth used by the Server for each Session (in the last sampling period). It is expressed as Kilobits per second and is comprised of network overheads.
- **MESSAGES.EVENTS_SEC** The frequency of Client Messages submitted to the Metadata Adapter in the last sampling period. It is expressed as messages per second.
- **MESSAGES.TOTAL_HANDLED** The total (cumulative) number of Client Messages submitted to the Metadata Adapter in the life of the Server.
- **MESSAGES.TOTAL_BYTES** The total (cumulative) number of bytes submitted to the Metadata Adapter in the life of the Server and related to Client Messages associated to sendMessage requests. The count includes the byte length of the String objects carrying the messages, according with their internal UTF-16 encoding.
- **MESSAGES.CURRENT_THROUGHPUT** The current global throughput (measured in the last sampling period) related to the submission to the Metadata Adapter of Client Messages associated to sendMessage requests. It is expressed as Kilobits per second and it counts the byte length of the String objects carrying the messages, according with their internal UTF-16 encoding.
- **MESSAGES.HIGHEST_THROUGHPUT** The maximum global throughput (occurred in the life of the Server) related to the submission to the Metadata Adapter of Client Messages associated to sendMessage requests. It is expressed as Kilobits per second and it counts the byte length of the String objects carrying the messages, according with their internal UTF-16 encoding.

- **PUSH_NOTIFICATIONS.DEVICES** The number of Devices currently served by the MPN Module, when enabled. Actually, multiple apps can request Push Notifications for the same physical device; here, by Device we mean the combination of a physical device and an app.
- **PUSH_NOTIFICATIONS.MAX_DEVICES** The maximum number of Devices concurrently served by the MPN Module reached in the life of the Server.
- **PUSH_NOTIFICATIONS.NOTIFICATIONS_SEC** The frequency of Push Notifications submitted by the MPN Module to the proper external service for delivery in the last sampling period. It is expressed as notifications per second.
- **PUSH_NOTIFICATIONS.TOTAL_SENT** The total (cumulative) number of Push Notifications submitted by the MPN Module to the proper external service for delivery in the life of the Server.
- **PUSH_NOTIFICATIONS.TOTAL_BYTES** The total (cumulative) number of bytes related with Push Notifications submitted by the MPN Module to the proper external service for delivery in the life of the Server. The count only includes the notification payload.
- **PUSH_NOTIFICATIONS.CURRENT_THROUGHPUT** The current global throughput (measured in the last sampling period) related with Push Notifications submitted by the MPN Module to the proper external service for delivery. It is expressed as Kilobits per second and it only counts the notification payload.
- **PUMP_WAIT.SLEEP** The average extra-time experienced for sleep() system calls. It is expected that the difference between the expected time and the observed time increases when the available CPU decreases, though significant delays with plenty of CPU available have sometimes been observed.
- **PUMP_WAIT.NOTIFY** The average time experienced between a notify() system call and the exit from the related wait() in a producer/consumer interaction. It is expected that this time difference increases when the available CPU decreases, though significant time differences with plenty of CPU available have sometimes been observed.

- **monitor_identification**

  Basic data about the Server, provided in MERGE mode (but they only consist of a snapshot).

  The available fields are the following:

  - **EDITION** The edition in which Lightstreamer Server is running.
  - **VERSION** The version of Lightstreamer Server.
  - **LICENSE_TYPE** The license type used to run Lightstreamer Server: "FREE", "DEMO", "EVALUATION", "STARTUP", "NON-PRODUCTION-LIMITED", "NON-PRODUCTION-FULL", "PRODUCTION", or "HOT-STANDBY".
  - **CLIENT_ID** The client id (or contract id) of the license used to run Lightstreamer Server. For "FREE" license type "-" is returned; and for a "DEMO" license type "DEMO" is returned.
  - **LOCAL_IP** The local IP address of the host running Lightstreamer Server.
  - **LOCAL_HOST** The local hostname of the host running Lightstreamer Server.

- **STARTUP_TIME** The timestamp representing the number of milliseconds elapsed between 1 January 1970 00:00:00 UTC and the moment the Lightstreamer Server instance was started.

- **monitor_details**

  Information details about edition and license features enabled in the Server, provided in MERGE mode. The available fields are the following:

  - **MAX_SSNS** The maximum number of concurrent Sessions allowed. Unlimited in case of a Per-server license; contractually limited in case of a Per-user license.
  - **IS_MPN** Whether the Push Notifications (APNs and FCM) are enabled in the server.
  - **EXP_DATE** License expiry date, null if never expire.
  - **MAX_RATE** Max message rate allowed for each individual Item (can 1 update per second, 3 updates per second or unlimited).
  - **BAND** Whether the Bandwidth control for each client sessions is allowed in the server.
  - **TLS** Whether TLS/SSL connections are allowed in the server.
  - **JMX** Whether JMX Management API is allowed in the server.
  - **VAL_TYPE** Validation type requested for license check; can be "ONLINE" or "FILE". "None" in case of "FREE" or "DEMO" license type.
  - **MAX_NODEJSAPI** Max Node.js Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_ANDROIDAPI** Max Android Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_IOSAPI** Max iOs Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_OSXAPI** Max macOS Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_TVOSAPI** Max tvOS Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_WATCHOSAPI** Max watchOS Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_FLASHAPI** Max Flash Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_FLEXAPI** Max Flex Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_SLAPI** Max Silverlight Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_JSEAPI** Max Java SE Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
  - **MAX_DOTNETSTDAPI** Max .NET Standard Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).

- **MAX_PCLAPI** Max .NET PCL Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
- **MAX_DOTNETAPI** Max Unity Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
- **MAX_BBAPI** Max BlackBarry Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
- **MAX_JMEAPI** Max Java ME Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).
- **MAX_GENAPI** Max Generic Client API version allowed ("" means all version, null means disabled feature and "*" means disabled by configuration file).

- **monitor_client_libs**

  This Item provides a table handled in COMMAND mode in order to recap the Client APIs available and the max version supported for each of them. For each client API that is, a row in the table, available fields are the following:

  - **key** the short alphabetic code for the Client API.
  - **command** "ADD", "DELETE", or "UPDATE", this is a service field for the management of COMMAND mode.
  - **lib_ext** extended name for the Client API.
  - **free, demo, eval, startup, noprod-lim, noprod-full, prod, standby** one of these fields, depending on the license actually in use, provides the licensing information for the Client API: the maximum license version allowed ("" means all versions) or if the feature is disabled (null means disabled, "*" means disabled by configuration file); the other fields are null.
  - **ordinal** this filed should be considered as private.

- **monitor_socket_<name>**

  Information about a server socket, as configured through the <http_server> or <https_server> elements, recognized through the specified name. The data is provided in MERGE mode (but it only consists of a snapshot).

  The available fields are the following:

  - **SERVER_NAME** the name of the server socket, as configured through the "name" attribute.
  - **LOCAL_PORT** the configured value for the local interface IP address on this server socket, or null if no configuration has been supplied.
  - **LOCAL_INTERFACE** the local port where this server socket is listening.
  - **IS_SSL** "true" if the server socket is for https connections.

  Note that configuring a name that contains spaces is allowed in <http_server> and <https_server>. In that case, the name of this item would also contain spaces, hence the Metadata Provider to be used for the custom Adapter Set should not be the sample LiteralBasedProvider or equivalent.

- **monitor_log_error**, **monitor_log_warning**, **monitor_log_info**, **monitor_log_output**

  The log produced by the Server on the preconfigured "LSProducer" appender, provided in DISTINCT mode without snapshot support. The different items select messages based on the priority level, in the following way:

  - monitor_log_error: ERROR and FATAL
  - monitor_log_warning: WARN
  - monitor_log_info: INFO
  - monitor_log_output: all messages

  The available fields are the same for all these items; they are the following:

  - **TIME** a timestamp in the "HH:mm:ss" form.
  - **TIME_MS** a UTC timestamp.
  - **CLIENT.IP** the IP address of the related client (only available for client-related messages).
  - **CLIENT.NAME** the hostname of the related client, if the hostname lookup is enabled and successful; the IP address otherwise (only available for client-related messages).
  - **MESSAGE** the log message.
  - **THREAD** the name of the thread originating the log message.
  - **COUNTER** a progressive counter, started on item subscription.

5 Mobile and Web Push Notifications

5.1 Data Streaming vs. Push Notifications

When dealing with real-time messaging within mobile and web apps, the difference between real-time data streaming and push notifications might not be perfectly clear.

1. With **data streaming**, the server sends real-time data directly to the app, leveraging WebSockets and/or HTTP. This requires the app to be running and, with iOS, to be in foreground. In case of browser-based apps, the browser must be running and must be pointed to that app.
2. With **push notifications**, the server sends messages to Google's and Apple's servers, which deliver them to the mobile devices and web browsers. Notifications are received by the device even if the app that subscribed to them is not in foreground or is not even running. In case of browser-based apps, the browser must be running (except for Safari) but the app does not need to be loaded: the browser might be displaying a completely different site.

Push notifications are the perfect means to reach a user when they are not using the app.

Push notifications are natively supported by Android and iOS devices, as well as by the Chrome, Firefox, and Safari browsers for any platforms, based on two standards: Google's FCM and Apple's APNs.

There are several cases where it would make sense to use data streaming and push notifications together. You can send important messages to your users via push notifications, even if they are not using your app in that moment. Then, you continue via data streaming when they open the app, thus supporting larger data volumes, increased bandwidth, decreased latency, and smart throttling.

Some examples of typical uses of push notifications are:

- Notify the user when the price of a financial instrument hits a given threshold
- Notify the user when an order has been executed
- Let the user know a new chat message has been received
- Remind the user their favorite TV show is starting
- Involve the user, whatever they are doing and wherever they are, as soon as something happens that could bring them back to your app.

5.2 The Lightstreamer MPN Module

Lightstreamer supports native push notifications on **Apple platforms** (iOS, macOS, tvOS, watchOS, Safari) via **Apple Push Notification Service** (APNs), and on **Google platforms** (Android, Chrome) and **Firefox** via **Firebase Cloud Messaging** (FCM).

Mobile and web push notifications (MPN for short) are handled by a special Lightstreamer Server's module called **MPN Module**. Since MPNs are designed to reach the user even when their application is not running, they can't be bound to a common Lightstreamer session. For this reason, the MPN Module has the ability to manage special subscriptions, called MPN subscriptions, which survive the scope of a session and are persisted on an SQL database.

Thanks to the MPN Module, items that are already managed by Data Adapters can be pushed to mobile and web clients through native push notifications, in addition to Web based protocols. Making items available via push notifications does not require any further server-side development.

MPN subscriptions are kept alive by the MPN Module and restarted each time the Lightstreamer Server restarts. Multiple instances of the Server (e.g. in a clustered configuration) are able to pass MPN subscriptions to each other by leveraging the database. Whenever an MPN subscription produces an update, this is formatted in a native push notification according to user's specifications and sent to the appropriate MPN Provider (APNs or FCM).

An extended view of the system architecture, including the MPN Module, is the following:

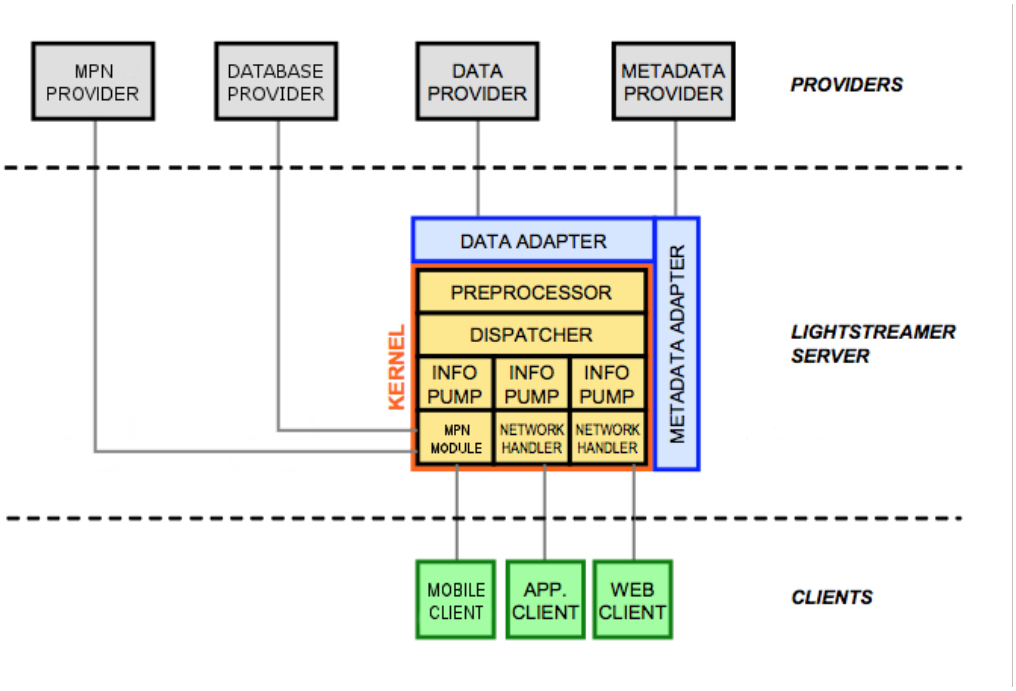


Figure 12-Extended view of the architecture with MPN Module

Mobile and web clients can activate MPN subscriptions by using special APIs provided by their Lightstreamer SDK. An MPN subscription activation requires an open Lightstreamer session and takes both the parameters of a common real-time subscription (data adapter, group ID, schema, mode, etc.) and the additional parameters needed to specify how (and when) notifications must be formatted and delivered.

The MPN Module is disabled by default, but can be enabled and configured through the **<mpn>** section of the Lightstreamer Server configuration file.

An introductory tutorial and live examples with full source code are available online, to get started with mobile and web push notifications very easily:

➔ **Tutorial for Android and iOS mobile apps:**

blog.lightstreamer.com/2018/01/mobile-push-notifications-with.html

➔ **iOS client example:**

github.com/Lightstreamer/Lightstreamer-example-MPNStockList-client-ios

➔ **Android client example:**

github.com/Lightstreamer/Lightstreamer-example-MPNStockList-client-android

➔ **Web client example:**

github.com/Lightstreamer/Lightstreamer-example-MPNStockList-client-javascript

5.3 The MPN Device

The MPN Module organizes the *end-points* of native push notification delivery in **MPN devices**: each MPN device represents a specific app running on a specific mobile device on a specific platform, or alternatively a specific web app running on a specific browser (Safari, Chrome or Firefox) on a specific desktop device.

In other words, two separate apps running on the same device and platform are represented with separate MPN devices. In the same way, the same app running on two different platforms is again represented with separate MPN devices.

E.g., consider the following:

- an app identified with "com.mydomain.MyApp" running on an iOS device;
- the same app, identified with "com.mydomain.MyApp", running on an Android device;
- a second app, by the same developer, identified with "com.mydomain.MyOtherApp" and running on the same iOS device of the first app.
- a web app with URL "myapp.mydomain.com", running on Chrome on a desktop device.

They are all considered as separate MPN devices. Each one of these entities has, in fact, a separate and independent set of MPN subscriptions.

5.3.1 Lifecycle

To be used for an MPN subscription, an MPN device must first be **registered**. Registration is a two-step process where the platform's operating system first assigns a unique *token* to the mobile or desktop device. Subsequently, this token is passed to the MPN Module, where it is stored on the database together with the platform and app identifiers, and assigned a unique MPN device ID for later references. The MPN Module routes native push notifications to the appropriate device using the platform-app-token triplet.

Once registered, the lifecycle of an MPN device is composed of just two possible states:

- **ACTIVE**: the initial state, obtained upon registration, in which the MPN device accepts MPN subscriptions and sends notifications.
- **SUSPENDED**: a state of temporary suspension, usually reached as a result of an error while sending a notification. In this state no notifications are sent and no MPN subscriptions are accepted. A new registration of the same token or a token change can both restore the "ACTIVE" state.

The following diagram shows the possible states and transitions:

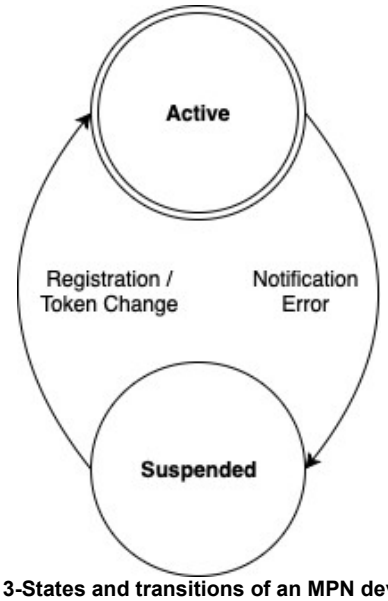


Fig 13-States and transitions of an MPN device

The registration is performed by Lightstreamer SDKs once the platform-specific token has been obtained. Each client instance of a Lightstreamer SDK may have *at most one* MPN device registered at any given time.

MPN devices are subject to **garbage collection**: devices left in "SUSPENDED" state, or with no active subscriptions, are deleted after a certain amount of time. However, the MPN device can be registered again in any moment. The interval between garbage collections and the amount of time for device expiration are specified with **<collector_period_minutes>** and **<device_inactivity_timeout_minutes>** tags, respectively, in the **<mpn>** section of the Lightstreamer configuration file.

5.4 MPN Subscriptions

MPN subscriptions are activated by using specific APIs made available on Lightstreamer SDKs, and require that an MPN device has been previously registered (see §5.3.1). Activating an MPN subscription requires an open session, even if the subscription itself is not bound to the session. A session is necessary so that the Metadata Provider may validate and authorize the MPN subscription request. To this purpose, an MPN subscription request can only refer to items served by the Adapter Set associated with the hosting session. Special considerations apply in case of a clustered configuration (see §5.6.1).

MPN subscriptions are subject to limitations, due to their nature of delivering updates directly to the end user (in most cases). They are composed of both the parameters of a common real-time subscription and a few more to specify how and when notifications must be delivered. On common real-time subscription parameters the following limitations apply:

- Only **MERGE** and **DISTINCT** modes are allowed.
- Unfiltered dispatch can't be used.
- Snapshot can't be used.
- Selectors and customizers can't be used (but additional parameters dedicated to MPNs may better serve the same purposes, see below).

Additional parameters, specific to MPNs, are the following:

- The **notification format** lets you specify *how to format* the notification message: it is an arbitrary structure, expressed in JSON format, that is interpreted by the MPN Provider. It can contain a special syntax that lets you compose the message with the content of the subscription updates (see §5.4.1).
- The optional **trigger expression** lets you specify *when to send* the notification message: it is a boolean expression, in Java language, that when evaluates to true triggers the sending of the notification (see §5.4.2). It also supports a special syntax that lets you evaluate the expression based on the content of each subscription update. If not specified, a notification is sent each time the Data Adapter produces an update.

Moreover, the subscription "group" should consist of a single item, although this is just for clarity, not mandatory. In fact, subscription attributes like the notification format and the trigger expression are meant to refer to a single update flow. If a group consisting of multiple items is used, their update flows will be merged into a single flow, where each update will be handled regardless of the item generating it.

5.4.1 The Notification Format

The notification format is specified as a provider-specific JSON structure. E.g., the notification format of a simple MPN subscription for Apple platforms could be the following:

```
{   "aps" : {
        "alert" : "Your stock price has changed"
    }
}
```

Lightstreamer SDKs provide a helper object, called *MPN Builder*, that builds the appropriate JSON structure required by the platform's MPN Provider.

Any part of the content of this structure accepts the embedding of *arguments*, that are later replaced by the MPN Module with the content of the corresponding subscription update.

These arguments can be expressed using a *named argument* or *indexed argument* syntax:

- A **named argument** is specified by its field name surrounded by **${** and **}**, e.g.: "${last_price}".
- An **indexed argument** is specified by its 1-based field position surrounded by **$[** and **]**, e.g.: "$[1]".

Thus, the notification above may be formulated as:

```
{  "aps" : {
      "alert" : "Your stock ${stock_name} is now ${last_price}"
   }
}
```

For instance, if the subscription update contains the value "Anduct" for the "stock_name" field and "13.63" for the "last_price" field, the actual notification would be:

```
{  "aps" : {
      "alert" : "Your stock Anduct is now 13.63"
   }
}
```

Note that the schema and group ID of the subscription are interpreted through the Metadata Adapter as usual. In particular, the schema ID is used to determine the array of subscribed fields and the **indexed arguments** refer to this array.

In order to identify a **named argument**, the name specified in the argument is submitted to the Metadata Adapter, by treating it as a schema ID to be interpreted, and it is expected to receive in return an array of a single field. Then, the received field name (which may or may not be identical to the submitted name) is expected to match the name of one of the subscribed fields (i.e. the ones determined by the subscription's schema ID).

For this reason, the use of named arguments is possible only if the Metadata Adapter supports the needed further request. This is ensured, for instance, if the Metadata Adapter is a subclass of *LiteralBasedProvider* or an equivalent class. On the other hand, the use of indexed arguments is always possible.

Once determined the subscribed fields referred to by the named arguments, for each named argument, an equivalent indexed argument can be determined and the whole notification format specification could be converted to only use indexed arguments. This **preliminary conversion** is, in fact, done and the converted form is the one used internally by the MPN Module, to store the subscription in the DB, to check coalescence (see §5.4.3), and to notify the subscription through the Internal MPN Data Adapter (see §5.5).

Note that the notification format specifications submitted to the Metadata Adapter for the authorization of the MPN Subscription are also the converted ones, hence they may only contain indexed arguments, not named arguments.

5.4.2 The Trigger Expression

In many cases the delivery of a notification for each update coming from the Data Adapter is not desirable. Instead, the delivery of a notification only when a specific condition is met may be preferable. E.g., send a notification when a stock value rises above a certain threshold. Trigger expressions are designed for this purpose.

A trigger expression is a **Java boolean expression** that is evaluated on the MPN Module against the update values, and, if found true, triggers the sending of the notification. An MPN subscription actually works differently whether a trigger expression is present or absent:

- In the presence of a trigger expression, the MPN subscription sends the notification **only when the expression evaluates to true for first time**, and no other notifications are sent after that.
- In the absence of a trigger expression, the MPN subscription sends a notification **each time the subscription produces an update**.

The trigger expression may contain **named arguments** and **indexed arguments** similarly to the notification format. See §5.4.1 for all remarks about the syntax and the **preliminary conversion** of named arguments to indexed arguments.

Note that, from a Java point of view, named and indexed arguments must be considered as *String* expressions. Hence, they must undergo appropriate type conversion if a comparison is needed. E.g.:

- "${stock_price} > 500.0" is an **invalid** trigger expression, since you can't compare a String with a number.
- "Double.parseDouble(${stock_price}) > 500.0" is a **valid** trigger expression, as it correctly converts the String to a number before comparison.

In fact, after the preliminary conversion of the trigger expression, indexed arguments are finally converted to variables of *String* type before the trigger expression is evaluated. Specifically, these variables have the prefix *LS_MPN_field* and a suffix which corresponds to the field index.

For instance, assuming that stock_price is the 3rd field in the schema,

```
Double.parseDouble(${stock_price}) > 500.0
```

is preliminarily converted to

```
Double.parseDouble($[3]) > 500.0
```

and finally converted to

```
Double.parseDouble(LS_MPN_field3) > 500.0
```

Trigger expressions may make use of any Java system class (by using their fully qualified name) but, for security reasons, no other classes. You can't access the Lightstreamer Server JMX APIs, for example, or your own provided *jars*. However, distinct executions of a trigger expression may share the same static context and perform side-effects on this context; this is not limited to the same underlying device.

To avoid that a hacker may compromise the Server integrity with a maliciously crafted trigger expression, the Metadata Adapter, when asked to authorize the client MPN subscription request, has the opportunity to check the submitted trigger expression and it is responsible for its acceptance.

The MPN Module also provides a prefiltering system, based on matching with regular expressions, that in most cases is sufficient to guarantee the Server's safety. This prefiltering is configured in the **<trigger_expressions>** section of the MPN Provider configuration files (see §5.7). Here, a list of (Java) regular expressions can be specified to validate triggers submitted by clients. For a trigger to be accepted, it must match at least one regular expression.

For instance, consider the following list:

```
<trigger_expressions>
    <accept>Double\.parseDouble\(\$\[\d+\]\)
       [&lt;&gt;] [+-]?(\d*\.)?\d+</accept>
</trigger_expressions>
```

It will let pass a trigger like "Double.parseDouble($[1]) > 500.0", but of course not one like "System.exit(0)". This simple example shows why an accept-all regular expression like ".*", even if possible, is strongly discouraged.

Note that the trigger expressions submitted for comparison with the list of regular expressions, like the ones submittted to the Metadata Adapter for authorization of the MPN Subscription, have already been subject to the **preliminary conversion**, hence they may only contain indexed arguments, not named arguments. So, always expect the "$[*digits*]" format, not the "${*name*}" format.

5.4.3 Coalescence

Once an MPN subscription has been activated, it may be examined, modified or unsubscribed from any session: the one where it was created or any subsequent one.

Consider the typical workflow of a Lightstreamer app:

1. Connect to Lightstreamer.
2. Activate a real-time subscription.
3. Receive updates.

Applied unchanged to MPN subscriptions it becomes the following workflow:

1. Connect to Lightstreamer.
2. Activate an MPN subscription.
3. Receive notifications.

This workflow has a problem: it will lead to multiple notifications for the same update. In fact, while real-time subscriptions are lost when the session is closed (i.e. when the app disconnects or is terminated), MPN subscription are not: they remain live on the MPN Module. Hence, each run of the app would then add *a new* MPN subscription.

The appropriate workflow for MPN subscriptions is the following:

1. Connect to Lightstreamer.
2. Wait for the currently active MPN subscription list to be available.
3. Check the list to see if the MPN subscription is present:
    - if it is, the MPN subscription is already active (and may be inspected for its details);
    - if it is not, the MPN subscription may be safely activated;
4. Receive notifications.

To simplify this workflow, the MPN subscription may be activated using the **coalescing** flag. This flag has been introduced for all cases where there is a fixed list of MPN subscriptions that must always be active. It may be specified during subscription activation, and its effect is that if an MPN subscription with the same **base parameters** exists (i.e.: adapter set, data adapter, group ID, schema and trigger expression), the activation is actually considered a modification of the existing subscription.

In other words, using the "coalescing" flag, just *one* MPN subscription is created per unique combination of base parameters, and subsequent activations just modify the other non-base parameters (e.g. the notification format).

Using coalescing, the simplified workflow then becomes:

1. Connect to Lightstreamer.
2. Activate the MPN subscription with **coalescing = true**.
3. Receive notifications.

Of course, coalescing makes sense if always used on the same set of MPN subscriptions, and since their initial activation. Use of this flag when multiple MPN subscriptions, with the same base parameters, are already present on the database, may lead the MPN Module to coalesce one at random.

5.4.4 Notification Limits

Additionally to the limitations of MPN subscriptions (see §5.4), there are also limitations on notifications themselves. In particular:

- **Frequency limit**: MPN Providers don't allow a high pace of notifications to be sent to the same device. If notifications are sent too frequently the provider may forcibly disconnect the MPN Module and ban it as a client, effectively causing a service interruption for all devices. For this reason, a specific maximum frequency is set in terms of a minimum delay between subsequent notifications. This limit is per MPN Provider and may be configured with the **<min_send_delay_millis>** tag in the provider's configuration file (see §5.7). The limit is enforced on a per MPN device basis, i.e.: notifications sent to two different apps on the same device are counted separately.
- **Size limit**: MPN Providers also set a limit to the maximum size of notifications. Check the documentation of your platform to know the current limit, as it may change with time. Remember that the notification format provided during subscription may not represent the final size of the notification, since the content of arguments also counts towards the size. The MPN Module enforces the limit and drops any notification of excessive size before it is sent to the provider.

5.4.5 Lifecycle

When an MPN subscription is activated by the MPN Module, it is stored on the database as a child entity of the MPN device it belongs to, and assigned an *MPN subscription ID* for later references.

The possible states of an MPN subscription are the following:

- **ACTIVE**: the initial state, in which the subscription either sends notifications, if it has no trigger, or monitors incoming updates for triggering.
- **TRIGGERED**: the state the subscription reaches when a trigger is present and it has been evaluated to true. In this state no other notifications are sent. A modification to the subscription restores the "ACTIVE" state (even if it does not specify a new trigger).

The following diagram shows the possible states and transitions:

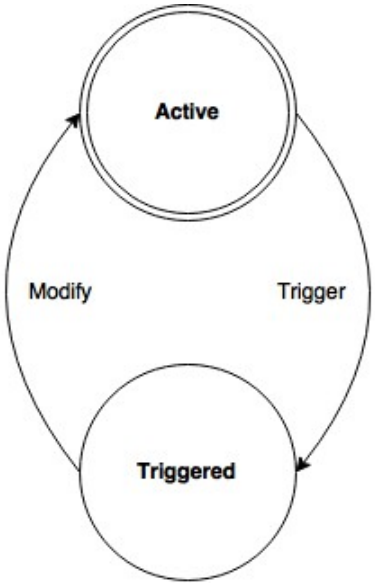


Fig 14-States and transitions of an MPN subscription

5.5 The Internal MPN Data Adapter

The MPN Module contains a special Data Adapter that provides the list and current status of all the MPN subscriptions of an MPN device, exposed as common real-time items. It is added automatically to any Adapter Set when the MPN Module is enabled. Its name can be configured through the **<internal_data_adapter>** tag in the **<mpn>** section of the Lightstreamer Server configuration file.

Under normal circumstances there is no need to know the adapter's items and fields, as it is exploited internally by Lightstreamer SDKs to provide the current MPN subscription list to the client (and all related events). However, if you need to implement your own client library, and have to support MPNs, following you can find all the details about its structure.

The internal MPN Data Adapter provides the following items:

- **Device status items**:
  - Item type: MERGE

- Item name: **DEV-*<MPN device ID>***
  - Where *<MPN device ID>* is the ID assigned by the MPN Module during registration. Typically it is a UUID, e.g.: "DEV-1e789ba3-4eda-429b-959d-07d7b2a36191".
- Fields:
  - **status**: the current status of the device, either "ACTIVE" or "SUSPENDED".
  - **status_timestamp**: the timestamp of the latest change of status, expressed as a Java timestamp (i.e. number of milliseconds since 1/1/1970 00:00:00 GMT).

- **Device subscription list items**:
  - Item type: COMMAND
  - Item name: **SUBS-*<MPN device ID>***
    - Where *<MPN device ID>* is the ID assigned by the MPN Module during registration. Typically it is a UUID, e.g.: "SUBS-1e789ba3-4eda-429b-959d-07d7b2a36191".
  - Fields:
    - **key**: the name of the item representing the details of this subscription, in the form SUB-<MPN subscription ID>.
    - **command**: the command for the key, either "ADD", "DELETE" or "UPDATE" (following the common rules for command mode items, see §3.2).
- **Subscription details items**:
  - Item type: MERGE
  - Item name: **SUB-*<MPN subscription ID>***
    - Where *<MPN subscription ID>* is the ID assigned by the MPN Module during activation. Typically is is a UUID, e.g.: "SUB-05357a23-fd10-4978-980b-32137fe70a90".
  - Fields:
    - **status**: the status of the MPN subscription, either "ACTIVE" or "TRIGGERED".
    - **status_timestamp**: the timestamp of the latest change of status, expressed as a Java timestamp.
    - **notification_format**: the notification format, represented as a JSON structure.
    - **trigger**: the trigger expression, represented as a Java expression.
    - **group**: the group ID.
    - **schema**: the schema.
    - **adapter**: the name of the Data Adapter.
    - **mode**: the subscription mode, either "MERGE" or "DISTINCT".
    - **buffer_size**: the buffer size, either "unlimited" or an integer number.
    - **max_frequency**: the max frequency, either "unlimited" or a decimal value.

Access to the internal MPN Data Adapter requires an appropriate workflow:

1. A **session** is opened.
2. The **device is registered** to the Lightstreamer Server using the specific TLCP request. The Server responds with the MPN device ID and the name of the internal Data Adapter. Check the TLCP Specifications document for more information on the Lightstreamer Server underlying protocol.
3. Knowing the Data Adapter name and the device ID, a MERGE real-time subscription is activated to the **device status item**, using the item name composed by "DEV-" and the device ID.
4. A COMMAND real-time subscription is activated to the **device subscription list item**, using the item name composed by "SUBS-" and the device ID.
5. The **device subscription list snapshot** is received, which contains all the known MPN subscriptions for the device.
6. Finally, a MERGE real-time subscription is activated for each **subscription details item**, using the keys obtained at step 5 as item names.

With these real-time subscriptions in place, the client is kept up to date of any change in the subscriptions' details or status. E.g.:

- If a new MPN subscription is activated, an ADD command is sent on the COMMAND real-time subscription activated at step 4.
- If an MPN triggers, additionally to the corresponding notification, an update event is sent on its MERGE real-time subscription activated at step 6.

Note that the items of a specific device and its subscriptions are accessible only from a session where the device has been registered. Trying to access the items from a session without a previous registration will result in an error.

Also consider that the real-time updates of the items may not be guaranteed when a cluster of Server instances is in place. In that case, only changes requested by the same session (like subscription activations) are guaranteed to be notified back in real-time. This limitation will be addressed in future releases.

5.6 The Database Provider

Persistence in the MPN Module is managed through *Hibernate*, a well-known object-relational mapping tool that efficiently encapsulates the underlying SQL database's dialect and details. Hibernate has its own configuration file, which must be pointed to by the **<hibernate_config>** tag in the **<mpn>** section of the Lightstreamer Server configuration file.

The Hibernate configuration file contains options for:

- the JDBC connection parameters;
- the specific SQL dialect to be used;
- the JDBC connection pool parameters;
- debugging options like whether SQL queries have to be logged or not, and others.

The provided file is preconfigured for an HSQL database engine, suitable for testing, while sample parameters are also shown for MySQL and Oracle databases.

The Hibernate configuration file also contains pointers to the database entities mapping files. These files, in turn, contain options to set column names and sizes. The provided default values are compliant with predominant databases, but they may have to be tuned for particular deployments.

The database tables store the current status of active MPN subscriptions, and the provided column sizes affect the limits imposed upon single fields of subscription activation. Modifying the column size to change the limit is allowed, if supported by careful testing. The column meaning can be inferred from mapping files; on the other hand, a detailed description of the persisted data is currently not provided.

Once the Hibernate configuration file has been set, the appropriate libraries (e.g. the database's JDBC driver and its dependencies) must be added under the **lib/mpn/hibernate** directory of the Lightstreamer Server installation. The JDBC driver is typically distributed together with the database.

5.6.1 The Database in a Clustered Configuration

In case of a clustered configuration, **all Lightstreamer Server instances must point to the same database provider**. It is up to the Integrator to provide an SQL database with appropriate redundancy and load balancing.

Moreover, all nodes of the cluster must share the same Adapter Sets. In fact, nodes may pass MPN subscriptions to each other during events like fail-overs or restarts: in these cases the MPN Module must be able to start MPN subscriptions previously activated by another node.

In case of configuration changes with MPN subscriptions already in place (such as the adapter set changing name), the use of a custom SQL script to update existing rows may be necessary. The database structure has been designed to be inspectable and, when client APIs can't be used, updatable. An Integrator that finds themselves in this situation may contact support@lightstreamer.com for advices on how to proceed.

5.6.2 DBMS Compatibility

The MPN Module has been successfully tested with the following DBMS:

DBMS	Compatible	Version tested	JDBC driver tested	Notes
HSQLDB	✓	2.3.0	embedded	None.
MySQL	✓	5.5.35	5.1.28	None.
Oracle	✓	11g	ojdbc6 release 2	None.
MS SQL Server	✓	2008 R2	sqljdbc4 2008 R2	None.
PostgreSQL	✓	9.1.13	9.3-1101 jdbc4	None.
IBM DB2	✓	10.5 FP3	10.5 FP3	Requires a change in the table name for Module objects, may be set in the **Module.hbm.xml** file. Default name is "MODULES", which is a reserved keyword.

Other DMBS may result compatible or require some adjustments. Hibernate usually provides good compatibility with well-known DBMS.

5.7 The MPN Provider

At the time of writing, supported MPN Providers are:

- **Apple's APNs** (Apple Push Notification service), used by iOS, macOS, tvOS, watchOS and Web (see the support table below) Lightstreamer SDKs.
- **Google's FCM** (Firebase Cloud Messaging), used by the Android and Web (see the support table below) Lightstreamer SDK.

More providers will come with future releases.

Desktop-Windows	Desktop-macOS	Mobile-Android	Mobile-iOS
Chrome 22+	Safari 6+	Chrome 80+	Not Supported
Firefox 22+		Firefox 68+	
Firefox ESR 78+		Opera 46+	
Edge 14+			
Opera 25+			

Table 5: Browser support for Web Push Notifications

5.7.1 APNs

The Apple Push Notification Service Provider, or *Apple notifier* for short, is configured through a specific configuration file pointed to by the **<apple_notifier_conf>** tag in the **<mpn>** section of the Lightstreamer Server configuration file.

The sample configuration file, included with Lightstreamer Server's distribution, is enriched with comments that describe in detail each available parameter. Stripped of comments, a sample file would look like the following:

```
<apple_notifier_conf>
   <min_send_delay_millis>1000</min_send_delay_millis>
   <connection_timeout>30000</connection_timeout>

   <app id="com.mydomain.myapp">
      <service_level>development</service_level>
      <keystore_file>my_app_keystore.p12</keystore_file>
      <keystore_password>mypassword</keystore_password>

      <trigger_expressions>
         <accept>Double\.parseDouble\(\$\[\d+\]\)
            [&lt;&gt;] [+-]?(\d*\.)?\d+</accept>
      </trigger_expressions>
```

```
    </app>

    <app id="web.com.mydomain.myotherapp">
        <service_level>production</service_level>
        <keystore_file>my_other_app_keystore.p12</keystore_file>
        <keystore_password>myotherpassword</keystore_password>
        <push_package_file>push_package.zip</push_package_file>
    </app>
</apple_notifier_conf>
```

The file is subdivided in **<app>** sections, with each section specifying parameters for a served mobile or web app. To be able to send notifications through APNs, each app must specify its own certificate (obtainable through Apple's Developer Portal) and related password, in **p12** format. Recall that Apple enforces the service level: development (sandbox) certificates may be used when debugging apps, but to use a production certificate you a need a production (e.g. distribution or ad-hoc) provisioning profile for your app.

As discussed in §5.4.2, each app configuration may also specify a list of (Java) regular expressions to be used to validate trigger expressions. If omitted, no trigger will be accepted for the app.

Safari Push Notifications

To send notifications to a **Safari web app**, some special considerations apply:

- The **app ID** must begin with "web.", e.g. "web.com.mydomain.myotherapp", and match the website push ID.
- The **service level** must be either "test" or "production". There is no "development" service level for web apps, since there is no separate development environment on which to test them (there is just one World Wide Web, in fact).
- A **push package** zip file must be specified. See also below.
- The **notification format** must always include at least one URL argument, even if the push package website JSON specifies a URL format string with no arguments. If URL arguments are omitted, notifications will not be delivered.
    - Recall that URL arguments are composed in an URL, and hence they must be **URL-encoded** (i.e. they can't contain spaces or special characters).

You can find detailed information on how to prepare your web app on Apple's documentation:

- Configuring Safari Push Notifications

The Push Package File

The push package zip file is a descriptor of the web app that Safari requests and requires to enable notifications delivery. Refer to Apple's documentation for details on its content.

Note that the **authentication token field** is ignored by the MPN Module and can be filled with a dummy value. The MPN Module relies on the authentication initially performed by the Lightstreamer Server when the session was opened.

A quick way to prepare the push package zip file for your web app is to make use of available open source utilities, such as this one:

- github.com/SymmetricInfinity/push_package

Note that the **webServiceURL** field specifies the URL to be supplied by the client application to *window.safari.pushNotification.requestPermission*, and that should be handled by the Server in a proper way. To achieve this, the path part should correspond with what configured in the <**apple_web_service_path**> tag in the **<mpn>** section of the Lightstreamer Server configuration file.

5.7.2 FCM

The Firebase Cloud Messaging Provider, or *Google notifier* for short, is configured through a specific configuration file pointed to by the <**google_notifier_conf>** tag in the **<mpn>** section of the Lightstreamer Server configuration file.

The sample configuration file, included with Lightstreamer Server's distribution, is enriched with comments that describe in detail each available parameter. Stripped of comments, a sample file would look like the following:

```
<google_notifier_conf>
   <min_send_delay_millis>1000</min_send_delay_millis>

   <app packageName="com.mydomain.myapp">
      <service_level>dry_run</service_level>
      <service_json_file>service.json</service_json_file>

      <trigger_expressions>
         <accept>Double\.parseDouble\(\$\[\d+\]\)
            [&lt;&gt;] [+-]?(\d*\.)?\d+</accept>
      </trigger_expressions>
   </app>
</google_notifier_conf>
```

The file is subdivided in **<app>** sections, with each section specifying parameters for a served mobile or web app. To be able to send notifications through FCM, each app must specify its own service JSON file (obtainable through Google's Developer Console). For testing purposes, each app may be configured for *dry run*, i.e. send notifications to Google but avoid their delivery to the device.

As discussed in §5.4.2, each app configuration may also specify a list of (Java) regular expressions to be used to validate trigger expressions. If omitted, no trigger will be accepted for the app.

The Service JSON File

Every app or web app requires a special JSON descriptor to enable delivery of notifications. Since they are delivered via Firebase Messaging, you need to configure a corresponding project on the Firebase console, and from there a you can download the service JSON file.

At time of writing, the following procedure details the steps needed to obtain the file:

1. Open console.cloud.google.com
2. Select your project.
3. Go to "API and Services" → "Credentials".
4. Click on "Create credentials" → "Service Account Key".
5. Select "firebase-adminsdk" and "JSON", then click "Create".

The file will be created and the downloaded immediately. Once created, the service account key associated with the file will be indicated in the "Service Account Keys" list of the "Credentials" section on the project's console.

Chrome and Firefox Push Notifications

To send notifications to a Chrome or Firefox web app, some special considerations apply:

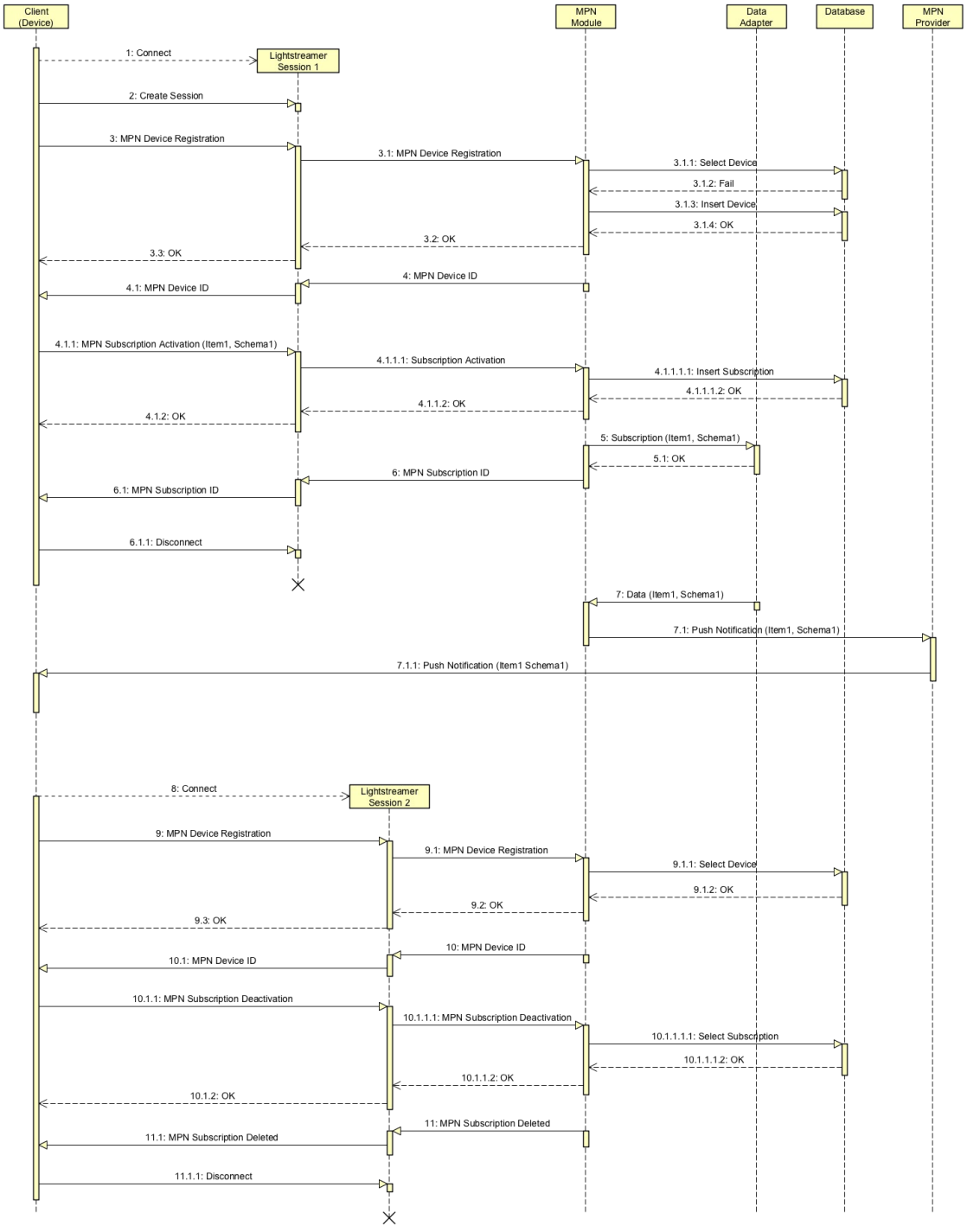
- You need to **generate a pair of VAPID keys** for the project.
- You need to **configure a manifest.json file** with appropriate settings.
- You need to **code a JavaScript service worker** that receives native push notifications while the web app is not running.
- The service worker is loaded only if the **web app HTTP connection is secure**. During development you may want to use a self-signed certificate, and in that case some settings must be tweaked on the browser to accept it (e.g. see this article for Chrome: Testing Service Workers locally with self signed certificates).
- Notifications are delivered to the service worker only when the **web app is in background**. When the web app is in the foreground, notifications are delivered directly to the app, on a specific event handler. This is relevant since service worker and web app have access to different UI APIs, and hence they will show notifications in different ways.
- For the web app to obtain a device token, the browser must be configured to **accept cookies**. Moreover, clearing the cookie storage will force the browser to obtain a new token the next the web app is opened.

You can find detailed information on how to prepare your web app on Google's documentation:

- Set up a JavaScript Firebase Cloud Messaging client app

5.8 Workflow Examples

The following diagram illustrates a full workflow of the MPN Module, including two separate sessions, device registration, subscription activation and push notification delivery. The diagram also includes interactions between the MPN Module, the database and the subscription data adapter. The internal data adapter, on the other hand, is not shown here.



5.9 Special Considerations on the GCM to FCM Transition

In April 2018, Google deprecated the Google Cloud Messaging (GCM) service in favor of the Firebase Cloud Messaging (FCM) service.

Version 7.1 of Lightstreamer Server abandons the use of GCM server libraries and switches to FCM server libraries for the delivery of native push notifications to Google platforms. This transition has the double purpose of:

1. Dismissing a deprecated service.
2. Introducing support for web push notifications on Chrome and Firefox (as documented earlier in this chapter).

This transition also has some unavoidable side effects:

- Previously obtained device tokens may not be compatible with FCM server libraries, resulting in an error when the Server tries to send a push notification to the device. Google states that tokens are interoperable between GCM and FCM server libraries, but our tests show a different situation.
- The JSON structure of the notification format is only partially compatible between GCM and FCM. Some properties, such as *icon*, *color*, *tag*, *sound* and others, are located in different substructures. Moreover, FCM introduces a substructure dedicated to Android notifications, previously absent in GCM notification formats.

The Android client library version 4.2 has been updated to be compatible with FCM server libraries. In particular:

- The MPN device now requires a pre-obtained device token upon creation. The previously existing factory method that obtained a device token automatically is no more available. Obtaining the token is now delegated to the Integrator, which can do so by leveraging standard Firebase APIs.
- The MPN builder now builds a notification format specific for FCM. In particular, all of the notification properties are set in the Android-specific substructure.

It is then advised to upgrade Android apps that make use of MPN features to a client library version 4.2 or newer. Note that apps developed for Apple platforms are not affected by this transition.

The table below summarizes client/server potential incompatibilities:

	Server Version < 7.1	**Server Version ≥ 7.1**
Android Client Version < 4.2	✓ **Compatible**	▲ **May not be compatible** Device token may not be compatible with FCM server libraries.
Android Client Version ≥ 4.2	▲ **May not be compatible** Notification format may not be compatible with GCM server libraries.	✓ **Compatible**

Note: the incompatibilities shown above pertain to MPN features only. Real-time features follow the traditional Lightstreamer compatibility rule that newer Server version always accept and support older clients.

5.9.1 How to Update Your Android MPN App to Server 7.1

Follow these steps:

1. Upgrade the Lightstreamer client library to a version greater than or equal to 4.2.
2. If you obtain the device token through the previously existing factory method, replace the call with the equivalent code that makes direct use of the Firebase APIs, in accordance with the Android API levels specific of your app.
    - If the next time the app is started it will probably obtain a different device token. The Server will automatically migrate all previous MPN subscriptions, no loss of subscriptions will occur.
3. If you are using the MPN builder to build the notification format, just check for deprecated properties (some properties are no more available under FMC).
4. If your are building your notification format manually, compare the old (GCM) and new (FCM) specifications and correct the format appropriately:
    - GCM: developers.google.com/cloud-messaging/http-server-ref
    - FCM: firebase.google.com/docs/cloud-messaging/admin/send-messages
5. Your upgrade is complete.

For any issue with the GCM to FCM transition contact support@lightstreamer.com.