weswit
Software Made in Italy


# Adapter Remoting Infrastructure

## Network Protocol Specification

Last updated: 18/04/2011

# Table of contents

# 1 The Architecture

The Lightstreamer **Adapter Remoting Infrastructure** (**ARI**) provides an easy way to develop Remote Adapters for Lightstreamer Server. “Remote” means that an Adapter does not run within the same process as the Java Virtual Machine running the Lightstreamer Server, but within a different process. Such process can be either local (on the same box) or actually remote (on a different box).

So, the advantages of a Remote Adapter with respect to a standard in-process Java Adapter are the following:

- Complete physical decoupling between the code of the Server and the code of the Adapter, avoiding possible interferences (memory usage, thread management, etc.).
- Possibility of deploying the Adapter on a different box and, if necessary, behind a second firewall.
- Possibility of implementing the Adapter in any language, rather than Java. The SDK for .NET Adapter development, provided as part of Lightstreamer distribution, is actually based on the ARI.

But there is also a drawback. Every interaction between Lightstreamer Server and a Remote Adapter requires information to be exchanged over TCP packets or local pipes, rather than communicating in process, resulting in potentially poorer performance. For this reason, some highly intensive callbacks of the Metadata Adapter interface are not supported by the ARI (for example, isSelected()).

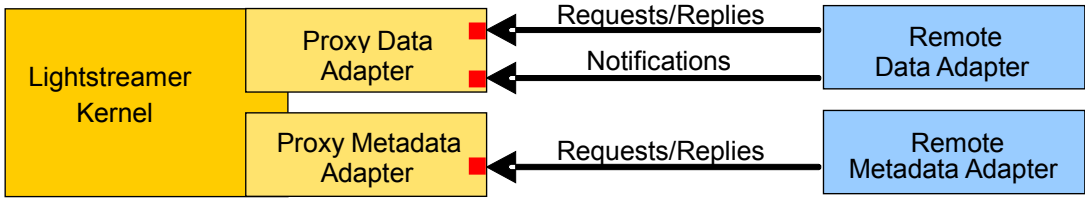
The communication between Lightstreamer Server and a Remote Adapter is based on unidirectional channels:

- **request channel** (data flow from Lightstreamer Server to the Remote Adapter)
- **reply channel** (data flow from the Remote Adapter to Lightstreamer Server)
- **notification channel** (data flow from the Remote Adapter to Lightstreamer Server)

Such channels can be implemented in two alternative ways: over TCP sockets or over local pipes.

## 1.1 ARI over TCP Sockets

The figure below shows the general architecture of the ARI when based on TCP sockets:




***ARI Protocol***

The Proxy Data Adapter and the Proxy Metadata Adapter are ready-made Adapters, written in Java and provided as part of the ARI SDK. They run in-process with Lightstreamer Server and expose the Data Provider and Metadata Provider interfaces over TCP/IP sockets.

The **Proxy Data Adapter** implements two server sockets. The first socket is used to transport the data flow of two unidirectional channels: **requests** (from the Proxy Data Adapter to the Remote Data Adapter) and **replies** (from the Remote Data Adapter to the Proxy Data Adapter). The second is used to transport the data flow of one unidirectional channel: **notifications** (from the Remote Data Adapter to the Proxy Data Adapter.

There exist two flavors of this Proxy Data Adapter: **NetworkedDataProvider** and **RobustNetworkedDataProvider** (see below).

The **Proxy Metadata Adapter** implements one server socket, used to transport the data flow of two unidirectional channels: **requests** (from the Proxy Metadata Adapter to the Remote Metadata Adapter) and **replies** (from the Remote Metadata Adapter to the Proxy Metadata Adapter).
This Proxy Metadata Adapter is named **NetworkedMetadataProvider**.
There exist two flavors of this Proxy Metadata Adapter: **NetworkedMetadataProvider** and **RobustNetworkedMetadataProvider** (see below).

There is always a one-to-one relationship between a Proxy Adapter instance and a Remote Adapter instance.

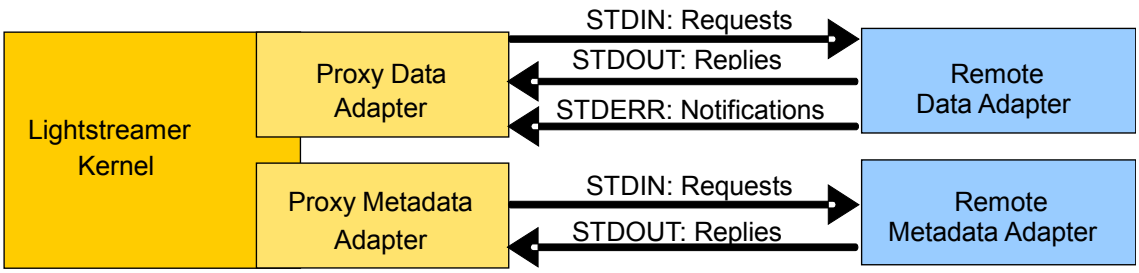
**NOTE:** From a TCP/IP perspective, the Remote Adapters are always clients, that is, they open TCP connections to Lightstreamer Server.

When Lightstreamer Server is launched, together with a Proxy Adapter, it listens on its server sockets(s) waiting for Remote Adapters to connect. After a Remote Adapter has connected, it will keep the connection open forever. Closing the connection could provoke the automatic shutdown of Lightstreamer Server, depending on the Proxy Adapter involved. The **NetworkedDataProvider** and **NetworkedMetadataProvider** shut the Server down, if they disconnect for the remote counterparts, in order to keep client and server state consistency (they cannot take any recovery action, so the best action is to have the client reconnect to another server instance in the cluster). On the other hand, the **RobustNetworkedDataProvider** and **RobustNetworkedMetadataProvider** contain some recovery capabilities and avoid to terminate the Lightstreamer Server process. Full details on the recovery behavior are available as inline comments within the “Lightstreamer/DOCS-SDKs/sdk_adapter_remoting_infrastructure\conf\sockets(robust)/adapters.xml” file.

**NOTE**: Lightstreamer Server performs Adapters initialization in parallel. Hence, if your custom executable manages several Remote .NET Adapter Servers, the connections to the related Proxy Adapters can be performed in sequence and no specific order is required.
Just consider that, after the connection to a Proxy Adapter has succeeded, there is no formal guarantee that the connection to the next Proxy Adapter will succeed immediately; a short time gap may occur on Lightstreamer Server between the opening of the server sockets for different Proxy Adapters.

## 1.2 ARI over Local Pipes

The figure below shows the general architecture of the ARI when based on local pipes (i.e. standard input, standard output and standard error of the Remote Adapter):




***ARI Protocol***

The Proxy Data Adapter and the Proxy Metadata Adapter are the same as in the TCP socket case.

The **Proxy Data Adapter** implements the three channels over pipes: **requests** (sent to the standard input of the Remote Data Adapter), **replies** (received from the standard output of the Remote Data Adapter), **notifications** (received from the standard output of the Remote Data Adapter).
This Proxy Data Adapter is named **PipedDataProvider**.

The **Proxy Metadata Adapter** implements the two channels over pipes: **requests** (sent to the standard input of the Remote Metadata Adapter), **replies** (received from the standard output of the Remote Metadata Adapter), **notifications** (received from the standard output of the Remote Metadata Adapter).
This Proxy Metadata Adapter is named **PipedMetadataProvider**.

There is always a one-to-one relationship between a Proxy Adapter instance and a Remote Adapter instance

**NOTE:** Every Remote Adapter process will be directly spawned by the corresponding Proxy Adapter.

When Lightstreamer Server is launched, together with a Proxy Adapter, a configured Remote Adapter is spawned and all the pipes are connected. After that, the termination of the Remote Adapter process provokes the automatic shutdown of Lightstreamer Server.

When choosing the pipe-based approach, consider that the Remote Adapter process should take care of not using standard input, output and error for other purposes (such as logging, etc.).

## 1.3 Deployment of the Proxy Adapters

All the Proxy Adapter of the Adapter Remoting Infrastructure (i.e. Data and Metadata, socket or pipe based) are contained in the **ls-proxy-adapters.jar** file, available under “Lightstreamer/DOCS-SDKs/sdk_adapter_remoting_infrastructure/lib”.

In order to deploy an Adapter Set which includes both a Proxy Metadata Adapter and a Proxy Data Adapter, the following steps should be followed:

1) Create a directory within “Lightstreamer/adapters”. It can be whatever name (for example, “proxy”).
2) Create a “lib” directory within the “proxy” directory and copy the “ls-proxy-adapters.jar” file in it.
3) Depending on the chosen Proxy Adapter type:
    - If deploying socket-based Proxy Adapters with robust Proxy Data and Metadata Adapters, copy the “adapters.xml” file located under “Lightstreamer/DOCS-SDKs/sdk_adapter_remoting_infrastructure/conf/sockets(robust)” to the “proxy” directory.
    - If deploying socket-based Proxy Adapters, copy the “adapters.xml” file located under “Lightstreamer/DOCS-SDKs/sdk_adapter_remoting_infrastructure/conf/sockets” to the “proxy” directory.
    - If deploying pipe-based Proxy Adapters, copy the “adapters.xml” file located under “Lightstreamer/DOCS-SDKs/sdk_adapter_remoting_infrastructure/conf/pipes” to the “proxy” directory.
4) Edit the copied “adapters.xml” file to configure the Proxy Adapters. See the file for inline comments.

The provided “adapters.xml” files are templates showing a typical configuration, where both the Adapters (Data and Metadata) are socket-based or pipe-based Proxy Adapters. Heterogeneous configurations are possible too. For example:

- A Proxy Data Adapter together with a provided Java Metadata Adapter (such as the LiteralBasedProvider).
- A Proxy Data Adapter together with a custom Java Metadata Adapter.
- A custom Java Data Adapter together with a Proxy Metadata Adapter.
- A Proxy Data Adapter based on sockets together with a Proxy Metadata Adapter based on pipes.
- Etc... Any other combination is possible.

If the defined Adapter Set includes multiple Data Adapters, each of them can be either a Proxy Data Adapter or a custom Java Data Adapter.

# 2 The ARI Protocol

## 2.1 Protocol Basics

The following are the basic principles of the ARI protocol:

•there are three unidirectional channels: requests, replies and notifications;
•every protocol packet represents a single request, reply or notification;
•requests are sent only from Proxy Adapters, replies and notifications are sent only from the counterpart;
•every request is identified by a unique ID that must be repeated as the ID of its corresponding reply; for some kinds of notification, the ID of the request that originates the notification is required;
•each notification is time stamped with a Java compatible millisecond resolution date-time (for statistical purposes);
•request, reply and notification packets are simple pipe-separated text lines, terminated by a line-feed or carriage-return & line-feed pair;
•every request, reply and notification must carry the information on which interface method it refers;
•requests and replies are completely asynchronous, i.e.: there is no need to answer a request before reading the next one (notifications are one-way only, so they are obviously asynchronous);
•keepalive messages originated by the Remote Data Adapter are allowed, in order to prevent network intermediate nodes from dropping the connections because of inactivity; the keepalive messages are simpler than request, reply and notify messages.

This implies that:

•in order to ensure that data can be transported by text lines, they must be encoded accordingly;
•in order to ensure that correct decoding can be applied for each data segment, corresponding data types must be specified;
•in order to ensure asynchronous request/reply, a pair of queue & de-queuing-thread must be created for each channel (request, replies and notifications);
•in order to ensure correct statistical treatment of notification timestamps, in case the counterpart runs on a separated machine the two machines must be synchronized through NTP or equivalent network time protocol (this is actually optional, statistics can be ignored and timestamp set to 0).

## 2.2 Packet Structure

The general structure of a protocol packet is as follows:

•request and reply packets:

```
<ID>|<method>|<data type 1>|<data segment 1>|...|<data type N>|<data segment N>\r\n
```

•notification packets:

```
<timestamp>|<method>|<data type 1>|<data segment 1>|...|<data type N>|<data segment
N>\r\n
```

•keepalive packets:

```
KEEPALIVE\r\n
```

Each data segment typically represents the content of a field or argument of a method, and the corresponding data type represents the native type in which it is expressed. There are a few cases where the structure above is not respected, particularly in the case of handling of counterpart-generated exceptions. See on for more details.

## 2.3 Data Types and Encodings

- **String**
  Data type: `S`
  Encoding:
  - `#` if null
  - `$` if empty
  - standard WWW UTF-8 url-encoding in any other case

- **Byte array** (signed or unsigned)
  Data type: `Y`
  Encoding:
  - `#` if null
  - `$` if empty
  - standard in-line BASE-64 encoding in any other case

- **Boolean**
  Data type: `B`
  Encoding:
  - `0` if false
  - `1` (or any other value) if true

- **Integer** (32 bit, signed)
  Data type: `I`
  Encoding: simple string representation of the integer

- **Double** (64 bit, signed, IEEE 754 formatted)
  Data type: `D`
  Encoding: simple string representation of the double, with decimal point (not comma)

In addition to these simple native types there are structured types, like exceptions and mode arrays, and the “no data” type:

- **Void** (i.e.: no data)
  Data type: `V`
  Encoding: not encoded, data segment is simply not present

- **Mode array**
  Data type: `M`
  Encoding:
  - `#` if null
  - `$` if empty
  - a character sequence in any other case, where each character represents a mode as per the following table:

| Mode | Character |
|---|---|
| R | Raw |
| M | Merge |
| D | Distinct |
| C | Command |

•**Exception**

Exceptions can be of generic type or of different subtypes, as per the following table:

| Type of exception | Data type | Encoding |
|---|---|---|
| Generic | E | string-encoding of detail message |
| Failure | EF | string-encoding of detail message |
| Subscription | EU | string-encoding of detail message |
| Access | EA | string-encoding of detail message |
| Items | EI | string-encoding of detail message |
| Schema | ES | string-encoding of detail message |
| Notification | EN | string-encoding of detail message |
| Credits | EC | 3 subsequent data segments with:<br>•string-encoding of detail message<br>•integer-encoding of client error code<br>•string-encoding of user message |
| Conflicting session | EX | four subsequent data segments with:<br>•string-encoding of detail message<br>•integer-encoding of client error code<br>•string-encoding of user message<br>•string-encoding of conflicting session ID |

## 2.4 Metadata Provider Protocol Methods

Metadata Provider protocol aims at exposing the MetadataProvider Java interface methods through the protocol. To reduce the protocol overhead, much of the original Java interface small methods have been aggregated in bigger protocol methods, carrying more information that are then cached on the Proxy Adapter side.

For more explanations regarding the semantics of each information, please consult the interface API documentation, either on Java API docs or (for an implementation example) on .Net API docs included in the distributions.

•**Notify User**

Direction: from Proxy Adapter to counterpart
Expects reply: yes
Method tag: NUS

Data segments:

- 1 *user name*, as string (can be null for the anonymous user)
- 1 *user password*, as string (can be null if no password is specified)
- N *http header name-value pairs*, both names and values as string
  the last pair is added by the Server; the name is “REQUEST_ID” and the value is a unique id assigned to the client request.

```
<ID>|NUS|S|<user name>|S|<user password>|S|<header 1>|S|<header value 1>|...|S|
<header N>|S|<header value N>|S|REQUEST_ID|S|<id>
```

Expected reply data segments:
- 1 *allowed max bandwidth*, as double
- 1 *wants table notifications flag*, as boolean

```
<ID>|NUS|D|<allowed max bandwidth>|B|<wants table notifications flag>
```

Reply can be also an exception of *Generic*, *Access* or *Credits* type

Examples:

```
→   10000010c3e4d0462|NUS|S|user1|S|password|S|host|S|www.mycompany.com|...
←   10000010c3e4d0462|NUS|D|40|B|0

→   20000010c3e4d0462|NUS|S|#|S|#|S|connection|S|Keep-Alive|...
←   20000010c3e4d0462|NUS|EC|Anonymous+user+not+allowed|1099|#
```

- **Notify User (extended version to carry identification data included in the client SSL certificate)**

Direction: from Proxy Adapter to counterpart
Expects reply: yes
Method tag: `NUA`

Data segments:
- 1 *user name*, as string (can be null for the anonymous user)
- 1 *user password*, as string (can be null if no password is specified)
- 1 client principal, as string (can be null if client not authenticated)
- N *http header name-value pairs*, both names and values as string
  the last pair is added by the Server; the name is “REQUEST_ID” and the value is a unique id assigned to the client request.

```
<ID>|NUA|S|<user name>|S|<user password>|S|<client principal>|S|<header 1>|S|
<header value 1>|...|S|<header N>|S|<header value N>|S|REQUEST_ID|S|<id>
```

Expected reply data segments:
- 1 *allowed max bandwidth*, as double
- 1 *wants table notifications flag*, as boolean

```
<ID>|NUA|D|<allowed max bandwidth>|B|<wants table notifications flag>
```

Reply can be also an exception of *Generic*, *Access* or *Credits* type

Examples:

```
   →   10000010c3e4d0462|NUA|S|user1|S|password|S|cn=john,cn=users,dc=acme,dc=com
|S|host|S|www.mycompany.com|...
   ←   10000010c3e4d0462|NUA|D|40|B|0

   →   20000010c3e4d0462|NUA|S|user1|S|password|S|#|S|connection|S|Keep-Alive|...
   ←   20000010c3e4d0462|NUA|EC|Unauthenticated+user+not+allowed|1098|#
```

- **Notify New Session**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `NNS`

  Data segments:
  - 1 *user name*, as string (can be null for the anonymous user)
  - 1 *session ID*, as string
  - N *client context name-value pairs*, both names and values as string
    the possible pairs differ from the java interface notifyNewSession case, because the "HTTP_HEADERS" property is not provided; a "REQUEST_ID" property is provided instead (see notes below).

```
<ID>|NNS|S|<user name>|S|<session ID>|S|<context prop name 1>|S|<context prop 1
value>|...|S|<context prop name N>|S|<context prop N value>
```

  Expected reply data segments: none, a void is expected

```
<ID>|NNS|V
```

  Reply can also be an exception of *Generic*, *Notification*, *Credits* or *Conflicting Session* type.
  In the last case, a second invocation of the command with the same "REQUEST_ID" and a different session name will be received.

  Examples:

```
   →   30000010c3e4d0462|NNS|S|user1|S|S8f3da29cfc463220T5454537|S|REMOTE_IP|
S|192.168.0.1|...
   ←   30000010c3e4d0462|NNS|V

   →   40000010c3e4d0462|NNS|S|user1|S|S9cb4758037a95c01T0439915|S|USER_AGENT|S|#|...
   ←   40000010c3e4d0462|NNS|EX|No+more+than+one+session+allowed|1101|#|
S8f3da29cfc463220T5454537
```

- **Notify Session Close**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `NSC`

  Data segments:
  - 1 *session ID*, as string

```
<ID>|NSC|S|<session ID>
```

  Expected reply data segments: none, a void is expected

```
<ID>|NSC|V
```

  Reply can also be an exception of *Generic* or *Notification* type.

  Examples:

```
   →   20000010c3e4d0462|NSC|S|S8f3da29cfc463220T5454537
   ←   20000010c3e4d0462|NSC|V

   →   30000010c3e4d0462|NSC|S|S9cb4758037a95c01T0439915
```

```
←   30000010c3e4d0462|NSC|EN|Session+not+open
```

- **Get Items**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `GIS`

  Data segments:
  - 1 *user name*, as string (can be null for the anonymous user)
  - 1 *group name*, as string
  - 1 *session ID*, as string

```
<ID>|GIS|S|<user name>|S|<group name>|S|<session ID>
```

  Expected reply data segments:
  - N *item names*, as string

```
<ID>|GIS|S|<item 1>|S|<item 2>|...|S|<item N>
```

  Reply can also be an exception of *Generic* or *Items* type.

  Examples:

```
→   50000010c3e4d0462|GIS|S|#|S|nasdaq100_AA_AL|S|S8f3da29cfc463220T5454537
←   50000010c3e4d0462|GIS|S|aapl|S|atvi|S|adbe|S|akam|S|altr

→   60000010c3e4d0462|GIS|S|#|S|nasdaq100_AA_AL|S|S9cb4758037a95c01T0439915
←   60000010c3e4d0462|GIS|EI|Unknown+group
```

- **Get Schema**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `GSC`

  Data segments:
  - 1 *user name*, as string (can be null for the anonymous user)
  - 1 *group name*, as string
  - 1 *schema name*, as string
  - 1 *session ID*, as string

```
<ID>|GSC|S|<user name>|S|<group name>|S|<schema name>|S|<session ID>
```

  Expected reply data segments:
  - N *field names*, as string

```
<ID>|GSC|S|<field 1>|S|<field 2>|...|S|<field N>
```

  Reply can also be an exception of *Generic*, *Items* or *Schema* type.

  Examples:

```
→   70000010c3e4d0462|GSC|S|#|S|nasdaq100_AA_AL|S|short|S|S8f3da29cfc463220T5454537
←   70000010c3e4d0462|GSC|S|last_price|S|time|S|pct_change

→   80000010c3e4d0462|GSC|S|#|S|nasdaq100_AA_AL|S|short|S|S9cb4758037a95c01T0439915
```

```
←   80000010c3e4d0462|GSC|ES|Unknown+schema
```

- **Get Item Data**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `GIT`

  Data segments:
  - N *item names*, as string

```
<ID>|GIT|S|<item 1>|S|<item 2>|...|S|<item N>
```

  Expected reply data segments:
  - N *item data structures*, composed by:
    - 1 *distinct snapshot length*, as integer
    - 1 *min source frequency*, as double
    - 1 *allowed modes*, as Mode array

```
<ID>|GIT|I|<dist. snapsh. len. 1>|D|<min source freq. 1>|M|<all. modes 1>|...
          ...|I|<dist. snapsh. len. N>|D|<min source freq. N>|M|<all. modes N>
```

  Reply can also be an exception of *Generic* type.

  Examples:

```
→   90000010c3e4d0462|GIT|S|aapl|S|atvi
←   90000010c3e4d0462|GIT|I|10|D|0|M|RMDC|I|30|D|0.01|M|R

→   a0000010c3e4d0462|GIT|S|aapl|S|atvi
←   a0000010c3e4d0462|GIT|E|Database+connection+error
```

- **Get User Item Data**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `GUI`

  Data segments:
  - 1 *user name*, as string (can be null for the anonymous user)
  - N *item names*, as string

```
<ID>|GUI|S|<user name>|S|<item 1>|S|<item 2>|...|S|<item N>
```

  Expected reply data segments:
  - N *user item data structures*, composed by:
    - 1 *allowed buffer size*, as integer
    - 1 *allowed max item frequency*, as double
    - 1 *allowed modes*, as Mode array

```
<ID>|GUI|I|<all. buf. size 1>|D|<all. max item freq. 1>|M|<all. modes 1>|...
       ...|I|< all. buf. size N>|D|<all. max item freq. N>|M|<all. modes N>
```

  Reply can also be an exception of *Generic* type.

  Examples:

```
→   b0000010c3e4d0462|GUI|S|user1|S|aapl|S|atvi
←   b0000010c3e4d0462|GUI|I|30|D|3|M|RMDC|I|30|D|0.3|M|$

→   c0000010c3e4d0462|GUI|S|#|S|aapl|S|atvi
←   c0000010c3e4d0462|GUI|E|Database+connection+error
```

- **Notify User Message**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `NUM`

  Data segments:
  - 1 *user name*, as string (can be null for the anonymous user)
  - 1 *session ID*, as string
  - 1 *user message*, as string

```
<ID>|NUM|S|<user name>|S|<session ID>|S|<user message>
```

  Expected reply data segments: none, a void is expected

```
<ID>|NUM|V
```

  Reply can also be an exception of *Generic*, *Notification* or *Credits* type.

  Examples:

```
→   d0000010c3e4d0462|NUM|S|user1|S|S8f3da29cfc463220T5454537|S|stop+logging
←   d0000010c3e4d0462|NUM|V

→   e0000010c3e4d0462|NUM|S|#|S|S9cb4758037a95c01T0439915|S|start+logging
←   e0000010c3e4d0462|NUM|EC|Anonymous+user+logging+not+allowed|1095|#
```

- **Notify New Tables**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `NNT`

  Data segments:
  - 1 *user name*, as string (can be null for the anonymous user)
  - 1 *session ID*, as string
  - N *table info structures*, composed by:
    - 1 *window index*, as integer
    - 1 *publishing mode*, as 1-length Mode array
    - 1 *group name*, as string
    - 1 *schema name*, as string
    - 1 *index of the first item*, as integer
    - 1 *index of the last item*, as integer
    - 1 *table selector*, as string (can be null if no selector is associated)

```
<ID>|NNT|S|<user name>|S|<session ID>|
      I|<win. index 1>|M|<pub. mode 1>|S|<group 1>|
      S|<schema 1>|I|<first item idx. 1>|I|<last item idx. 1>|S|<selector 1>|
      ...
      I|<win. index N>|M|<pub. mode N>|S|<group N>|
      S|<schema N>|I|<first item idx. N>|I|<last item idx. N>|S|<selector N>
```

Expected reply data segments: none, a void is expected

```
<ID>|NNT|V
```

Reply can also be an exception of *Generic*, *Notification* or *Credits* type.

Examples:

```
   →    f0000010c3e4d0462|NNT|S|#|S|S8f3da29cfc463220T5454537|I|1|M|M|
S|nasdaq100_AA_AL|S|short|I|1|I|5|S|#
   ←    f0000010c3e4d0462|NNT|V

   →    10000010c3e4d0462|NNT|S|#|S|S9cb4758037a95c01T0439915|I|1|M|M|
S|nasdaq100_AA_AL|S|short|I|1|I|5|S|#
   ←    10000010c3e4d0462|NNT|EN|Session+timed+out
```

- **Notify Tables Close**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: `NTC`

  Data segments:
  - 1 *session ID*, as string
  - N *table info structures*, composed by:
    - 1 *window index*, as integer
    - 1 *publishing mode*, as 1-length Mode array
    - 1 *group name*, as string
    - 1 *schema name*, as string
    - 1 *index of the first item*, as integer
    - 1 *index of the last item*, as integer
    - 1 *table selector*, as string (can be null if no selector is associated)

```
<ID>|NTC|S|<session ID>|
      I|<win. index 1>|M|<pub. mode 1>|S|<group 1>|
      S|<schema 1>|I|<first item idx. 1>|I|<last item idx. 1>|S|<selector 1>|
      ...
      I|<win. index N>|M|<pub. mode N>|S|<group N>|
      S|<schema N>|I|<first item idx. N>|I|<last item idx. N>|S|<selector N>
```

Expected reply data segments: none, a void is expected

```
<ID>|NTC|V
```

Reply can also be an exception of *Generic* or *Notification* type.

Examples:

```
   →    f0000010c3e4d0462|NTC|S|S8f3da29cfc463220T5454537|I|1|M|M|
S|nasdaq100_AA_AL|S|short|I|1|I|5|S|#
   ←    f0000010c3e4d0462|NTC|V

   →    10000010c3e4d0462|NTC|S|S9cb4758037a95c01T0439915|I|1|M|M|
S|nasdaq100_AA_AL|S|short|I|1|I|5|S|#
   ←    10000010c3e4d0462|NTC|EN|Table+not+open
```

## 2.5 Data Provider Protocol Methods

Data Provider protocol aims at exposing the DataProvider Java interface methods through the protocol. To find the best tradeoff between pros of interface remoting and cons of protocol overhead, some of the original Java interface methods have not been remotized, and others have been aggregated.

For more explanations regarding the semantics of each information, please consult the interface API documentation, either on Java API docs or (for an implementation example) on .Net API docs included in the distributions.

- **Subscribe**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: SUB

  Data segments:
  - 1 *item name*, as string

```
<ID>|SUB|S|<item name>
```

  Expected reply data segments: none, a void is expected

```
<ID>|SUB|V
```

  Reply can also be an exception of *Generic*, *Subscription* or *Failure* type.

  Examples:

```
→   10000010c3e4d0462|SUB|S|aapl
←   10000010c3e4d0462|SUB|V

→   20000010c3e4d0462|SUB|S|xyzy
←   20000010c3e4d0462|SUB|EU|Unknown+item
```

- **Unsubscribe**

  Direction: from Proxy Adapter to counterpart
  Expects reply: yes
  Method tag: USB

  Data segments:
  - 1 *item name*, as string

```
<ID>|USB|S|<item name>
```

  Expected reply data segments: none, a void is expected

```
<ID>|USB|V
```

  Reply can also be an exception of *Generic*, *Subscription* or *Failure* type.

  Examples:

```
→   30000010c3e4d0462|USB|S|aapl
←   30000010c3e4d0462|USB|V

→   40000010c3e4d0462|USB|S|xyzy
←   40000010c3e4d0462|USB|EU|Item+not+subscribed
```

- **End Of Snapshot**

  Direction: from counterpart to Proxy Adapter (notify)
  Expects reply: no
  Method tag: EOS

  Data segments:
  - 1 *item name*, as string
  - 1 *unique ID of the originating subscription request*, as string

```
<timestamp>|EOS|S|<item name>|S|<ID>
```

  Examples:

```
→   1152096504423|EOS|S|aapl|S|10000010c3e4d0462
```

- **Update By Map**

  Direction: from counterpart to Proxy Adapter (notify)
  Expects reply: no
  Method tag: UD3

  Data segments:
  - 1 *item name*, as string
  - 1 *unique ID of the originating subscription request*, as string
  - 1 *is snapshot flag*, as boolean
  - N *field-value pairs*, composed by:
    - 1 *field name*, as string
    - 1 *field value*, as string or byte array

```
<timestamp>|UD3|S|<item name>|S|<ID>|B|<is snapshot>|S|<field 1>|S|<value
1>|  ...  |S|<field N>|S|<value N>

<timestamp>|UD3|S|<item name>|S|<ID>|B|<is snapshot>|S|<field 1>|Y|<value
1>|  ...  |S|<field N>|Y|<value N>
```

  Note: the special mandatory fields for items to be requested in COMMAND mode, named “key” and “command”, must be encoded as string.

  Examples:

```
   →   1152096504423|UD3|S|aapl|S|10000010c3e4d0462|B|1|S|pct_change|S|0.44|
S|last_price|S|6.82|S|time|S|12%3a48%3a24

   →   1152096504423|UD3|S|aapl|S|10000010c3e4d0462|B|1|S|pct_change|Y|MC40NA==|
S|last_price|Y|Ni44Mg==|S|time|Y|MTI6NDg6MjQ=
```

- **Failure**

  Direction: from counterpart to Proxy Adapter (notify)
  Expects reply: no
  Method tag: FAL

Data segments:

- 1 *reason*, as generic exception

```
<timestamp>|FAL|E|<reason>
```

Examples:

→ `1152096504423|FAL|E|Connection+lost`

## 2.6 Notes on Protocol Method Sequence

- **End Of Snapshot**

  In case the snapshot for an item is not available, an *End Of Snapshot* notify must be sent to the Proxy Adapter during the subscription of the item. This will signal Lightstreamer kernel to avoid waiting for a successive set of snapshot updates (i.e. updates with the “is snapshot” flag set to true) for that item.

- **Notify User**

  The *Notify User* method is the very first request sent from the Java proxy to the counterpart (which version is used depends on the configuration of <use_client_auth>; by default, the base version is used). This is guaranteed by an exception that is thrown on the Proxy Adapter side if the call sequence should be different. This means that the Metadata Adapter has always a chance to authenticate users before any detail about their profile is requested.

- **Notify User**, **Notify New Session**

  All the authorization request management is expected to depend on the user name only. Anyway, some information on the specific client request instance are supplied to the *Notify New Session* method, as the *client context*. The *REQUEST_ID* property is the same id that has just been supplied to *Notify User* for the same client request instance; this allows for using local authentication-related details for the authorization task.
  Note: the Remote Adapter is responsible for disposing any cached information in case Notify New Session is not issued because of any early error during request management.

- **Notify New Tables**, **Notify Tables Close**

  These methods are requested by the Proxy Adapter only when the Remote Adapter asks for them through the *wants table notifications flag* returned with the *Notify User* method, on a user basis. If this flag is always returned as false, then these calls are never received.

- **Notify User, Notify New Tables**, **Notify New Session**, **Notify User Message, Get Items, Get Schema, Get Item Data**, **Get User Item Data**

  These methods are requested by the Proxy Adapter synchronously (i.e.: the requesting thread waits for the reply; however, the Proxy Adapter may still issue multiple requests in parallel, as stated in the general notes). Moreover, the requesting threads are taken from a limited pool. This means that these requests should be processed as fast as possible; but, anyway, any roundtrip delay related to the remote call will keep the Server waiting.

In order to avoid that delays on one session propagate to other sessions, the size of the thread pool devoted to the management of the client requests should be properly set, through the "server_pool_max_size" flag, in the Server configuration file.
Alternatively, a dedicated pool, properly sized, can be defined for the involved Adapter Set in “adapters.xml”. Still more restricted dedicated pools can be defined:

- for the *Notify User* method only,
- for the *Notify User Message* method only,
- for each Data Adapter in the Adapter Set.

The latter pool would also run any Metadata Adapter method related to the items supplied by the specified Data Adapter.

- **Notify New Tables**

  This method is requested by the Server while holding a lock to the session. This means that any operation on the session, including update dispatching, is blocked during the roundtrip time. Hence, this method should be used only if strictly needed. In general, the *wants table notifications flag* returned with the *Notify User* method should be false.

- **Notify Tables Close**, **Notify Session Close**, **Subscribe**, **Unsubscribe**

  These methods are requested by the Proxy Adapter side asynchronously (i.e.: without waiting for the reply). This does not mean that the reply should not be sent: the reply is mandatory (or a timeout exception will be raised on the Proxy Adapter side), but any exception that it should carry would simply be logged. Anyway, this means that these methods don't need to be non-blocking, as long as they don't last more than the configured timeout limit.

- **Notify New Session**, **Notify Session Close**

  These methods are invoked consistently. *Notify New Session* always precedes other methods related with the same session and *Notify Session Close* always follows other methods related with the same session.

- **Subscribe**, **Unsubscribe**

  These methods are invoked consistently. *Subscribe* always precedes *Unsubscribe* for the same item, but multiple *Subscribe-Unsubscribe* request pairs can be invoked for the same item.