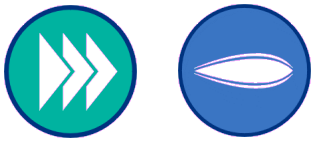

Packet Ship *Streamline* Media Server

Application Note AN-SL-601

v.3.1.2

Controlling & Observing Packet Ship Streamline behaviour with XML Messages

Documents PS-SL-3.1.2 “Antigua” Update 2 release

CONFIDENTIAL

Provided subject to terms of NDA and/or Reseller Agreement

Not for release to end customers

Contents

Introduction

The Packet Ship Streamline video server, ps-streamd, provides two messaging interfaces allowing the server to be integrated with external systems:

1. A standard document-based SOAP-over-HTTP interface
2. A peer-to-peer messaging bus interface using the ObTools XMLMesh bus

The functions of the messaging interface include:

- Stream start (setup) and stop (teardown)
- Stream playback control
- Current stream state and position
- Admission control status
- List of all running streams
- Server shutdown and configuration reload
- External access verification *
- Notification of stream start, stop and end *

The features marked * require messages to be initiated by the server and can only be accessed through XMLMesh.

This Application Note describes how external applications can integrate with Streamline through both of these mechanisms.

SOAP over HTTP

The simplest way to integrate with Streamline from most languages is the standard SOAP over HTTP interface. This is a document-based API (not SOAP RPC) running on a standard HTTP server configured in the **<soap>** controller in `streamd.cfg.xml`:

```
   <soap>
     <http>
       <server port="55480" address="localhost"/>
     </http>
  </soap>
```

The messages are POSTed to the HTTP server with a configurable URL for each type of message. The requests must be in SOAP format, although only the **<env:Body>** part is used. For example:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <ps:stream-start-request xmlns:ps=”http://packetship.com/ns”
                             asset = “test” output = “udp”>
      <ps:destination address=”225.1.1.1” port=”11111”/>
    </ps:stream-start-request>
  </env:Body>
</env:Envelope>
```


The 'ps:' namespace is standard for all Packet Ship messages, and is assumed even if no **`xmlns`** attribute is given.

The response may be a full document:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <ps:stream-start-response xmlns:ps="http://packetship.com/ns"
                             id = "c8e62773b04931730006" />
  </env:Body>
</env:Envelope>
```

or for simple requests that either succeed or fail, a simple **`<ps:ok/>`** element:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <ps:ok/>
  </env:Body>
</env:Envelope>
```

In either case, any errors are returned as a SOAP fault:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <env:Fault>
      <env:Code env:Value="env:Sender"/>
      <env:Reason xmlns:xml="http://www.w3.org/XML/1998/namespace"
        <env:Text xml:lang="en">No such stream</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

XMLMesh architecture

The ObTools XMLMesh message bus is a 'middleware' application which allows independent components, either on the same machine or on different ones, to communicate with XML messages on a publish-subscribe basis. Unlike a traditional Remote Procedure Call (RPC) system such as CORBA or Java RMI, a message is not directed at any particular receiver, but it sent out with a subject which one or more receivers may choose to subscribe to, act upon, and reply to.

Subjects & Subscription

Each XMLMesh message has a textual subject which by convention is split up into words separated by dots, like a reverse domain name, or Java package name. To avoid clashes, it is recommended that the messages begin with the sender's reversed domain name, with the allowance that '.com' can be left out: Hence all Packet Ship messages begin "**`packetship.`**", and all messages to do with streaming are prefixed with "**`packetship.stream.`**" To begin receiving messages, clients attach to the XMLMesh at a well-known TCP port, and request a subscription to a given subject pattern. The pattern can be expressed as a standard 'glob' pattern, such as "**`packetship.*`**". This pattern would receive every single Packet Ship message which passed through the system: Probably overkill, but interesting to try!

Request & Response

Messages can be sent and received in a one-way 'fire and forget' metaphor, or with a more conventional request/response. Each message has an ID, and an 'RSVP' flag which says whether or not a reply is expected. A receiver can simply respond to a message by sending back a reply quoting the request's ID.

Note that it is quite possible for more than one receiver to subscribe to a request, but only the first one to reply will have its reply forwarded back to the sender. However, as we shall see, the others can observe what is going on without having to reply.

Message Bodies

The XMLMesh bus uses the full SOAP <env:Header> format for internal routing, but this is hidden from developers using any of the standard language bindings (C, C++, PHP, Java). The actual messages sent are identical to the SOAP over HTTP ones, except that success for simple messages is indicated by an **`<x:ok/>`** element.

ps-streamd Configuration

In order to provide the messaging services described in this Application Note, the `streamd.cfg.xml` configuration will need to be changed from the default as supplied. The Streamline Installation and Configuration Guide (SL-ICG) gives full details.

Enabling SOAP over HTTP

The SOAP HTTP server is disabled by default and needs to be given a **`port`** number to listen on in order to enable it:

```
  <soap>
    <http>
      <server port="55480" address="localhost"/>
    </http>
    ...
</soap>
```

With **`address`** set to "`localhost`" it will only listen on the local interface ('lo'). To enable connections from outside you can either specify a specific interface IP address or "`0.0.0.0`" to listen on all interfaces.

Note: The SOAP interface is not authenticated so be very careful if this interface is opened up beyond the local machine – or even within the machine, if it has untrusted users.

Enabling XMLMesh

The XMLMesh interface is also configured within the **`<soap>`** controller. The **`<xmlmesh>`** element and contents need to be un-commented:

```
<soap>
   …
  <xmlmesh>
    <server host="localhost" port="29167"/>
    <notify end="no"/>
  </xmlmesh>
  …
</soap>
```

The **`host`** and **`port`** are the address of the XMLMesh broker (`ot-xmlmesh`) to connect to. 29167 is the standard XMLMesh port ('otmp') and by default it is running on the same server.

Enabling stream notifications

One of the features of the XMLMesh interface is that it can send notifications of streams starting, stopping (explicit stop), ending (naturally) and failing (other cause) to the external controller. To enable this, set **`<notifications enabled="yes"/>`** inside **`<xmlmesh>`**.

Enabling external access verification

The Admission Controller can also use XMLMesh to request external verification of a stream request, for example from an external subscriber management system. To enable this, uncomment the **<external>** element in **<admission>**:

```
<admission>
  …
  <external/>
  …
</admission>
```

Configuring message handlers

The server's SOAP controller provides a number of message handlers, each providing a different function. These can be 'bound' to particular HTTP URLs and/or XMLMesh message subjects. If a handler is not bound on one or both of the channels, then it is disabled.

The message bindings are defined in a **<messages>** element within the **<soap>** element. Each child element of this defines a message handler by name.

Handler name	Message type
<stream-start>	**<ps:stream-start-request>**
<stream-control>	**<ps:stream-control-request>**
<stream-status>	**<ps:stream-status-request>**
<stream-stop>	**<ps:stream-stop-request>**
<stream-server-status>	**<ps:stream-server-status-request>**

The function and syntax of each message handler is described later in this document.

HTTP URL bindings

Each message handler can be bound to a particular HTTP URL with a **<soap>** element inside the handler element. The **url** attribute gives the URL (or URL glob pattern) to bind to. For example:

```
<soap>
  ...
  <messages>
  ...
    <stream-status>
      <soap url="/status"/>
    </stream-status>
  ...
  </messages>
  ...
</soap>
```


XMLMesh subject bindings

Each message handler can also be bound to a particular XMLMesh subject pattern with an **`<xmlmesh>`** element inside the handler element. The **`subject`** attribute gives the message subject (or subject glob pattern) to bind to, minus the final "`.request`" which is added automatically. For example:

```
<soap>
  ...
  <messages>
  ...
    <server-status>
      <xmlmesh subject="packetship.stream.server.status"/>
    </server-status>
  ...
  </messages>
  ...
</soap>
```


Message handlers

The following section outlines the format and function of each of the message handlers.

Message handler: stream-start

Standard SOAP URL: **/start**

Standard XMLMesh subject: **packetship.stream.start.request**

The stream start message is the most fundamental, and requests the server to start a stream for a given asset to one or more destinations.

The body of the message must be a **<ps:stream-start-request>** element, with the following attributes:

- The **asset** gives the asset ID to be used, as looked up in the server's asset directory.
- The **output** gives the output ID to be used – e.g. "udp"or "rtp-udp" (note TCP outputs are not usable with externally initiated streams because they require a pre-existing client connection)

Then within the **<ps:stream-start-request>** can be placed one or more <ps:destination> elements with the following attributes:

- The **address** identifies the IP address of the receiving device for the stream, and should be in dotted-quad notation
- The **port** gives the receiver's UDP port number for the stream, in decimal

For example:

```
<ps:stream-start-request xmlns:ps="http://packetship.com/ns"
                         asset = "test" output = "udp">
  <ps:destination address="225.1.1.1" port="11111"/>
</ps:stream-start-request>
```

Response: <ps:stream-start-response>

The response to a **<ps:stream-start-request>** should be a **<ps:stream-start-response>** message or a SOAP fault message.

The **<ps:stream-start-response>** element has a single attribute:

- The **id** is the server's ID for the session, which should be quoted in further requests for this stream.

For example:

```
<ps:stream-start-response xmlns:ps="http://packetship.com/ns"
   id          = "c8e62773b04931730006"
/>
```


Message Handler: stream-control

Standard SOAP URL: **/control**

Standard XMLMesh subject: **packetship.stream.control.request**

All stream controls such as pause, play, fast-forward and rewind are handled by the same message handler. The body of the message must be a **<ps:stream-control-request>** element, with the following attributes:

- The **id** is the ID of the stream returned in the start response
- The **speed** attribute gives the speed of playback, as a decimal

Speed	Action
0	Pause
1.0	Play normally
< 0	Rewind
> 1.0	Fast forward

  The **speed** attribute can be left out, in which case 1.0 (normal play) is assumed.

- The **offset** attribute gives the offset to play at, in decimal seconds of Normal Play Time (NPT). If left out, -1.0 is assumed, which indicates 'continue from here, or from start'.
- The **end** attribute gives the offset to stop the stream at, in NPT seconds. If set to "0", this indicates play to the natural end of the stream. If left out, -1.0 is assumed, which means "leave it as it was before".

Control requests are also used as 'keepalive' requests for running streams with just an ID attribute. Note that external controllers have to provide keepalives just as clients do, unless the keepalive timeout is disabled.

For example (a pause request):

```
<ps:stream-control-request xmlns:ps="http://packetship.com/ns"
   id    = "c8e62773b04931730006"
   speed = "0"
/>
```

The result of a control request is just a basic **<ps:ok/>** (SOAP) or **<x:ok/>** (XMLMesh) message, or a SOAP fault if the stream ID doesn't exist.

Message Handler: stream-status

Standard SOAP URL: **/status**

Standard XMLMesh subject: **packetship.stream.status.request**

The stream request allows an external controller to obtain the current state and position of any stream (not just ones it started itself) – for example to bookmark videos for later resumption. The body of the message must be a **<ps:stream-status-request>** element, with a single **id** attribute giving the stream ID.

For example:

```
<ps:stream-status-request xmlns:ps="http://packetship.com/ns"
   id    = "c8e62773b04931730006"
/>
```

Response: <ps:stream-status-response>

The response to a **<ps:stream-status-request>** will be a **<ps:stream-status-response>** message or a SOAP fault message. The **<ps:stream-status-response>** contains a single **<ps:stream>** element with the following attributes:

- The **id** is the ID of the stream
- The **asset** is the asset ID being streamed
- The **controller** is the name of the controller that 'owns' the stream ("soap" for streams created through this interface)
- The **index** is the server's internal index number for the stream
- The **output** is the output being used – e.g. "udp"
- The **position** is the current position in Normal Play Time (NPT) seconds (floating point)
- The **speed** is the current speed of playback
- The **state** is the current state of the stream, one of:

State name	Meaning
starting	Stream being started
playing	Playing normally (at any speed)
runout	Aligning to an access point before a transition
held	Held at the end or beginning during fast-forward or rewind
finishing	Reached natural end, playing out from buffers
finished	Reached natural end, last packet sent
notified	Controllers notified of end
failed	Internal failure
stopped	Stopped by controller

Within the **<ps:stream>** element will be a **<ps:destination>** element for each destination address, containing **address** (dotted-quad) and **port** attributes.

For example:

```
<ps:stream-status-response xmlns:ps="http://packetship.com/ns">
  <ps:stream asset="test" controller="soap" id="c8e62773b04931730006"
index="1" output="udp" position="18.462" speed="1" state="playing">
    <ps:destination address="225.1.1.1" port="11111"/>
  </ps:stream>
</ps:stream-status-response>
```

Message Handler: stream-stop

Standard SOAP URL: **/stop**

Standard XMLMesh subject: **packetship.stream.stop.request**

The stream stop request stops a stream. The body of the message must be a **<ps:stream-stop-request>** element, with an **id** attribute being the ID of the session returned in the start response.

For example:

```
<ps:stream-stop-request xmlns:ps="http://packetship.com/ns"
   id = "c8e62773b04931730006"
/>
```

The result of a stop request is just a basic **<ps:ok/>** (SOAP) or **<x:ok/>** (XMLMesh) message, or a SOAP fault.

Message Handler: server-status

Standard SOAP URL: **/server-status**

Standard XMLMesh subject: **packetship.stream.server.status.request**

The master status request is used to request a status report from the streaming daemon. The body of the message should be a **<ps:server-status-request>** element, which has a single **streams** attribute. If this is set to 'true', the result will include detailed reports on every stream; otherwise it will only contain summary information.

For example:

```
<ps:server-status-request
        xmlns:ps="http://packetship.com/ns" streams="true"/>
```

Response: <ps:server-status-response>

This message is the result of a server status request, and will contain a **<ps:server-status-response>** element, containin a **<ps:streams>** element, giving details on all the streams in the system (only if **streams** is set in the request) and a **<ps:admission>** element giving the admission control status of the server.

Stream list

The **<ps:streams>** element is only included if the **streams** attribute is set in the request. It contains a **<ps:stream>** element for every stream as is returned for individual stream status requests (see above).

Admission control status

The **<ps:admission>** element is always present and contains information about the admission control status of the server. It contains an element for each stateful admission control filter in the system. In the current version this only means **<capacity>** filters, which create **<ps:capacity>** elements.

Each **<ps:capacity>** element has an **id** attribute as configured in `streamd.cfg.xml`, or for licence limits, as read from the licence file. It also contains **<ps:streams>** and **<ps:bandwidth>** elements giving information about stream numbers and bandwidth capacity and usage. Each of these in turn contain **max** and **used** attributes giving the maximum configured and the current amount in use, respectively. Bandwidth is counted in megabits/second, the same way it is configured in the **<capacity>** admissions filter.

If a **<capacity>** filter is specific to a particular network route or asset ID pattern, the **<ps:capacity>** element will also have **route** and/or **asset** attributes, respectively, as configured in streamd.cfg.xml.

Server status example

The following example shows a typical status result with a single stream:

```
<ps:server-status-response xmlns:ps="http://packetship.com/ns">
  <ps:streams>
    <ps:stream asset="test" controller="soap"
               id="c8e62773b04931730006" index="1" output="udp"
               position="19.79" speed="1" state="playing">
      <ps:destination address="225.1.1.1" port="11111"/>
    </ps:stream>
  </ps:streams>
  <ps:admission>
    <ps:capacity id="multicast-bandwidth" route="224.0.0.0/4">
      <ps:streams max="0" used="1"/>
      <ps:bandwidth max="50" used="4.526"/>
    </ps:capacity>
    <ps:capacity asset="test*" id="test-streams">
      <ps:streams max="10" used="1"/>
      <ps:bandwidth max="0" used="4.526"/>
    </ps:capacity>
    <ps:capacity id="licence">
      <ps:streams max="0" used="1"/>
      <ps:bandwidth max="500" used="4.526"/>
    </ps:capacity>
  </ps:admission>
</ps:server-status-response>
```


Message Handler: server-control

Standard SOAP URL: **/server-control**

Standard XMLMesh subject: **packetship.stream.server.control.request**

The server control request has various effects on the server as a whole, depending on the content of the message. The body of the message must be a **<ps:server-control-request>** element, which can contain one or more of the following sub-elements:

<ps:shutdown>

If present anywhere in the message, the server is cleanly shut down and all other sub-elements are ignored.

```
<ps:server-control-request xmlns:ps="http://packetship.com/ns">
  <ps:shutdown/>
</ps:server-control-request>
```

<ps:reload>

All the dynamic configuration of the server is reloaded, as if it had received a SIGHUP. If present, other reload-type commands are ignored.

```
<ps:server-control-request xmlns:ps="http://packetship.com/ns">
  <ps:reload/>
</ps:server-control-request>
```

<ps:reload-directory>

Only the asset directory is reloaded.

```
<ps:server-control-request xmlns:ps="http://packetship.com/ns">
  <ps:reload-directory/>
</ps:server-control-request>
```

<ps:reload-controller>

The configuration of the controller identified by an **id** attribute is reloaded. This is usually used with **id="permanent"** to reload the permanent stream (broadcast) configuration.

```
<ps:server-control-request xmlns:ps="http://packetship.com/ns">
  <ps:reload-controller id="permanent"/>
</ps:server-control-request>
```

The result of any server control request is just a basic **<ps:ok/>** (SOAP) or **<x:ok/>** (XMLMesh) message, or a SOAP fault.

Outgoing messages

The following messages are sent by the `ps-streamd` daemon to external listeners through XMLMesh only.

Notification: <ps:stream-notification>

Whenever a stream is started, stopped, ends or fails, the video server can send a one-way notification of the event which allows an external controller to deal with it in some way. This applies to all streams in the server, not just those started by the SOAP controller itself.

The body of the notification message will be a **`<ps:stream-notification>`** element, with an **`id`** attribute being the session ID, and an **`event`** attribute giving the event type, which will be one of:

- "`started`": The stream has successfully started
- "`stopped`": The stream was explicitly stopped by its controller
- "`ended`": The stream came to a natural end
- "`failed`": The stream failed for some other reason

The message element also has the following other attributes:

- **`controller`**: The name of the controller that created the stream
- **`asset`**: The asset ID being streamed
- **`output`**: The name of the output being used.

Then for every destination there will be a **`<ps:destination>`** element with **`address`** (dotted-quad) and **`port`** attributes.

The XMLMesh subject is constructed from the session ID as **`packetship.stream.notify.<event>.<sessionID>`**

For example:

```
[packetship.stream.notify.started.bdbda1b84d60002 ]
<ps:stream-notification asset="test" controller="rtsp"
        event="started" id="bdbda1b84d60002" output="udp"
        xmlns:ps="http://packetship.com/ns">
  <ps:destination address="127.0.0.1" port="48612"/>
</ps:stream-notification>
```

There is no response to a notification.

Request: <ps:stream-verify-request>

To allow more complex security mechanisms than can be handled by the video server itself – for example, checking billing status, the server offers a mechanism for hooking into its access verification process. This feature is enabled by **`<external/>`** in the **`<admission>`** element of

`streamd.cfg.xml`. Note that external verification can also be combined with any other kind of admission filtering.

The message contains a **`<ps:stream-verify-request>`** element with the following information:

- An **`asset`** attribute giving the asset ID being requested
- An **`output`** attribute giving the internal transport type (output) requested (e.g. “udp”)
- A **`controller`** attribute giving the name of the controller (e.g. “rtsp”)

Then for every destination there will be a **`<ps:destination>`** element with **`address`** (dotted-quad) and **`port`** attributes.

The XMLMesh subject of the request is **`packetship.stream.verify.request`**

The receiver of the message should respond with either a standard XMLMesh 'ok' response, or an XMLMesh error. If an error is returned the client will receive a “403 Forbidden” response.

For example:

```
[packetship.stream.access.verify]
<ps:stream-verify-request xmlns:ps="http://packetship.com/ns"
             controller=”rtsp” asset=“wotw-2” output=”udp”>
  <ps:destination client=“192.168.0.52” client-port=”11111”/>
</ps:stream-verify-request>
```


XMLMesh Message Interfaces

To send and receive XMLMesh messages requires a message interface to properly format the message, connect to the bus and send and receive the message data. The XMLMesh platform currently supports the following message interfaces:

- A native library interface in C++ (from which all the others are derived)
- A native library interface in C (sending only)
- A PHP module (sending only)
- A generic pair of command-line tools for send and receive, which can be called/called by any scripting language such as Perl, shell or Python

Because it is the most general, this document describes the last option; using the others will be similar but in the native syntax of the calling language.

The ot-xmlmesh-cli package

Although it is not part of the standard Packet Ship Streamline release, we can supply on request an additional package, 'ot-xmlmesh-cli'. This is installed just like the standard packages:

```
# dpkg –i ot-xmlmesh-cli_1.2.0-1_i386.deb
```

This installs two utilities in `/usr/bin/`: **`ot-xmlmesh-send`** and **`ot-xmlmesh-receive`**.

Sending messages with ot-xmlmesh-send

Running ot-xmlmesh-send with no arguments provides the following usage message:

```
Usage:
  ot-xmlmesh-send [options] <subject> [<file>]

Reads message from <file> or stdin, and sends it with the given subject
May output response to stdout if requested
Result code 0 for success, 1 for message failure, 2 for fatal error

Options:
  -c --check      Request response and check for OK, or output error to
stderr
  -r --response   Request response and output body to stdout
  -s --soap       Show full SOAP response (only if -r)
  -v --verbose    More logging
  -q --quiet      No logging, even on error
  -h --host       Set XMLMesh host (default 'localhost')
  -p --port       Set XMLMesh port (default 29167)
  -? --help       Output this usage
```


Sending simple one-way messages

In its simplest use, ot-xmlmesh-send reads an XML message from its standard input, or from a supplied file, and sends it with the subject given on the command line. For example

```
$ ot-xmlmesh-send “foo.test”
<foo:test/>
[ctrl-D]
```

This sends the simplest possible fire-and-forget message with subject “foo.test”. However, all the Packet Ship messages that can be sent by the RTSP daemon have a result, so we have to do a little more…

Sending a simple request

The next step up is to request a result and check for a standard ‘OK’ response, or an error. The ‘`-c`’ or ‘`--check`’ option provides this facility, and sets its result code accordingly. It also outputs any error to the standard error.

With this, we can send any of the stream control messages which just have a simple OK/Error result, e.g.:

```
$ ot-xmlmesh-send –c “packetship.stream.stop.request”
<ps:stream-stop-request xmlns:ps=”http://packetship.com/ns”
   id = ”c8e62773b04931730006”/>
[ctrl-D]
```

Sending a request and receiving the response

Finally, we may want to actually receive the full response from a request which generates one. The ‘`-r`’ or ‘`--response`’ option provides this. It outputs any response it receives to standard output, even if it’s an error (in which case it will be in the form of an **`<env:fault>`** element containing a SOAP fault structure)

With this, we can obtain the result of a stream start message:

```
$ ot-xmlmesh-send –r “packetship.stream.start.request”
<ps:stream-start-request xmlns:ps=”http://packetship.com/ns”
                         asset = “test” output = “udp”>
  <ps:destination address=”225.1.1.1” port=”11111”/>
</ps:stream-start-request>
[ctrl-D]
<ps:stream-start-response id = ”c8e62773b04931730006”
   xmlns:ps=”http://packetship.com/ns”/>
```


Receiving messages with ot-xmlmesh-receive

To receive messages (other than responses to a request that you initiated with `ot-xmlmesh-send`), you need to use the `ot-xmlmesh-receive` server. This runs as a permanent daemon, subscribing to message subjects that you specify, and then spawns a copy of your script for each message received. The best analogy is a CGI script being run by Apache.

If you run ot-xmlmesh-receive with no arguments, you get the following usage message:

```
Usage:
  ot-xmlmesh-receive [options] <subject> <receiver>

Runs as a daemon and subscribes for given <subject> and spawns <receiver>
for each message, with argv[1] as subject and message text on stdin.
Options:
  -o --observe     Observe only, don't return response even if requested
  -c --check       Check return code of receiver and send OK or Error
                   If return code is non-zero, any output will go into fault
  -r --response    Return response body from output of receiver
  -R --response-subject <subject>
                   Set subject of response (only when -r)
                   Default is received subject with '.response' appended
  -s --soap        Pass in full SOAP message wrapper
  -v --verbose     More logging
  -q --quiet       No logging, even on error
  -f --foreground  Run in foreground rather than as a daemon
  -1 --oneshot     Receive only one message and exit (default, loops forever)
  -h --host <host> Set XMLMesh host (default 'localhost')
  -p --port <port> Set XMLMesh port (default 29167)
  -? --help       Output this usage
```

The subject given can be a pattern match, such as "`packetship.stream.*`". The receiver script is called with the actual subject of the message as the first argument, and the body of the message as its input.

The '`--check`' (`-c`) option checks the return code of the receiver and sends back a simple OK/Error response, whereas the '`--response`' (`-r`) option accepts a complete response from the output of the script. If you are only observing notifications, and not responding to them, you only need to use the simplest form, '`--observe`' (`-o`).

Need more?

For more support and example scripts in the language of your choice, please contact support@packetship.com and we will do our best to help you.

