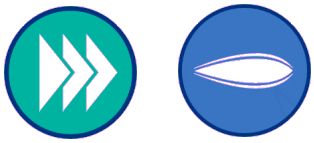


Packet Ship *Streamline* Media Server

ICG-SL

Installation & Configuration Guide
for 3.1 "Antigua" release

Document version 3.1.0

Documents software versions:
ps-streamd 3.1.0

Contents

Introduction....4
Overall architecture....4
Installation....7
Prerequisites....7
Installing on Debian....8
Installing on CentOS / Red Hat....8
Manual installation....9
Trying it out....11
Quick start....13
Obtaining a file....13
Uploading to the server....13
Indexing....13
Streaming the video....14
If it works....14
If it doesn't work....16
Systems Administration....17
Binaries....17
init.d script....17
Configuration....17
Logging....18
Processes & Threads....18
General server configuration....19
Logging....19
Background daemon....20
Watchdog restart....20
User/group identity....20
Licence file....20
General stream control....21
Keepalive timeouts....21
Empty packets....21
End of stream notification....22
Controllers....23
RTSP....23
HTTP....29
HLS....30
SOAP....32
Permanent streams....34
Directory services....36
Direct mapping....36
Configuration files....37
Gridline cache lookup....39
Timeline capture lookup....40
Admission control....41
Access control....41
External verification....43

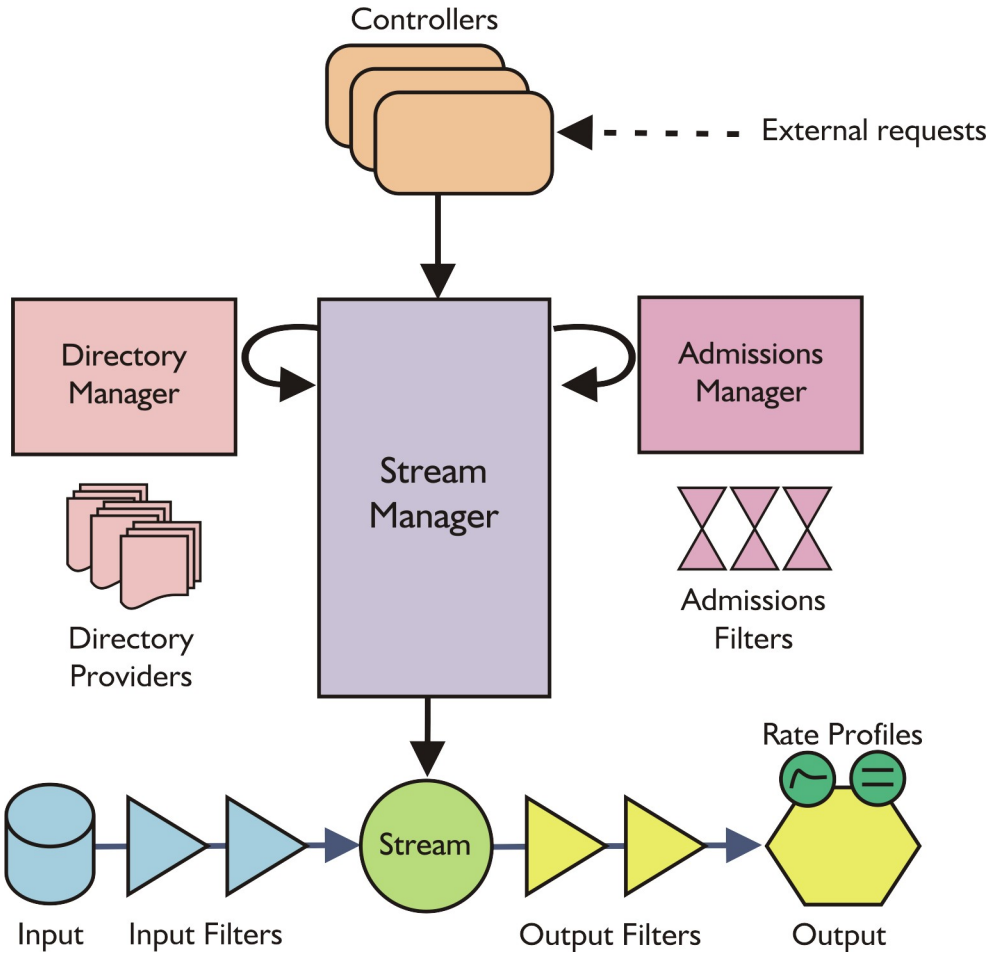
Per client limits....43
Capacity Limits....44
Inputs....46
File input....46
Outputs....48
Common output configuration....48
UDP/IP....50
TCP/IP....51
Output Filters....52
RTP....52
RTSP interleave....52
HTTP chunked....52
Transport Stream timing adjustment....52
Transport Stream continuity counter Adjustment....53
Rate Profiles....55
Fixed rate....55
Rate multiple....55
Rate limits....56
Stream groups....57
Threads....57
Buffer sizes....58
Burst length....59
Lag control....59
The realtime scheduler....61
Installing /dev/rtc....61
Index....63

Introduction

Packet Ship Streamline is a Linux server component which provides a high-performance streaming media server for video- and music-on-demand applications. This document describes how to install and configure Packet Ship Streamline on a Linux server. You should be familiar with Linux systems administration, video-on-demand concepts and XML in order to use this guide.

Overall architecture

The Packet Ship Streamline software is a single Linux daemon, '`ps-streamd`'. This daemon itself comprises a number of communicating modules. The abstract architecture of the daemon is shown in the following diagram:



So in words, a **Controller** – possibly as a result of an external request - asks the **Stream Manager** to start a **Stream**, quoting an asset ID. The **Stream Manager** looks up the requested asset ID in the **Directory Manager** which asks one or more **Directory Providers** to find it. If found, the **Stream Manager** requests admission of the stream in the **Admissions Manager**, which checks against zero or more **Admission Filters** before allowing the **Stream** to be created. If successful, an **Input** is created with zero or more **Input Filters**, and an **Output** with zero or more **Output Filters**, and the

Stream pumps data at a rate determined by the stream and any **Rate Profiles** associated with the **Output** through this pipeline until it is stopped by the **Controller** or comes to an end.

To make this rather abstract view more concrete, here is a list of the various types of modules supported in the current release:

Controllers

Controllers initiate and control streams either from internal configuration or responding to external requests. The currently supported Controllers are:

- **rtsp**: allowing streams to be started, stopped and managed by standard RTSP clients
- **http**: providing basic progressive download through HTTP
- **hls**: supporting Apple's HTTP Live Streaming for Mac, iPhone and iPad.
- **soap**: allowing streams to be started, stopped and managed through SOAP/XML messaging, either through a built-in HTTP server or over XMLMesh[1].
- **permanent**: allowing the creation of permanently running streams such as broadcasts and video loops

Directory Providers

Directory Providers look up an abstract "asset ID" quoted by a controller and turn it into a filesystem path (or playlist of paths). The currently supported Directory Providers are:

- **direct**: providing direct mapping of controller asset IDs to filesystem paths
- **config-files**: providing manually configured assets
- **cache**: providing lookup of assets in a Gridline cache
- **capture**: providing lookup of timed assets in a Timeline capture server

Gridline and Timeline are separate products – please enquire with Packet Ship sales for more details.

Admission Filters

Admission Filters act as a gateway to whether the stream is allowed to start or not, based on the client's address and the rate of stream determined from the asset file. The currently supported Admission Controllers are:

- **access**: Access Control Lists (ACLs) based on controller, client IP address and pattern match on asset ID.
- **capacity**: Capacity-based limits on number of streams and/or total bandwidth, optionally matched by client IP address and asset ID.
- **per-ip**: Per client IP address limits, and an option to kill orphan streams.
- **external**: Access verification with external integration through XMLMesh messages

[1]XMLMesh is a peer-to-peer messaging bus which carries SOAP messages but over a more flexible transport than HTTP

Inputs

An Input represents a source of data. The only Input currently supported is:

- **file**: Read from disk file

Input Filters

An Input Filter filters or observes data on the input side of the stream. There are no InputFilters in the current release; this is left as a future option.

Outputs

Outputs represent a destination of the streamed data. The Outputs currently supported are:

- **udp**: Output of stream data as UDP/IP datagrams
- **tcp**: Output of stream data as TCP/IP data, interleaved with the control socket

Output Filters

Output Filters provide protocol encapsulation or packet modification on the output side of the stream. The currently available Output Filters are:

- **rtp**: Encapsulation of raw stream data in RTP format
- **rtsp-interleave**: Interleaving stream data with RTSP commands
- **http-chunked**: HTTP/1.1 chunking
- **adjust-ts-timing**: Regeneration of Transport Stream timing to create a continuous timebase – for example for discontinuous input data such as playlists or loops.
- **adjust-ts-cc**: Regeneration of Transport Stream continuity counters to avoid discontinuities, for the same reasons.

Rate Profiles

Rate Profiles provide adjustment or limits of the raw streaming rate demanded by the assets (files) being streamed. The currently available Rate Profiles are:

- **fixed**: Overrides asset-derived rate pacing with a fixed maximum – used for TCP streaming where (up to a limit) the client decides the rate of consumption.
- **limit**: Imposes minimum and maximum rate limits on the asset-derived rate.
- **multiple**: Multiplies output rate, optionally for a fixed time at stream startup, to allow pre-fill of client buffers.

Installation

Packet Ship Streamline uses Debian and CentOS Linux as its reference platforms, and by default the software is supplied as Debian packages (.deb) and Red Hat packages (.rpm), for both 32-bit and 64-bit Intel/AMD architectures. One of these will work on most Linux distributions, but it is also possible to install it by hand – see later section.

Unix-friendly

Packet Ship Streamline is designed for integration with existing Linux platforms, so `ps-streamd` runs as a standard Unix daemon, just as you would expect from standard services like 'bind' and 'apache'. Like most Unix software it is configured in text files kept in `/etc`: specifically `/etc/packetship/streamd.cfg.xml`. Most of this guide concerns the contents of this file, but you'll find the comments fairly self-explanatory if you don't like reading manuals!

Also like conventional Unix daemons, `ps-streamd` has an '`init.d`' script and sends its logging to `/var/log/` – so, managing a Packet Ship Streamline installation is pretty much like managing any other Linux server.

Prerequisites

Packet Ship Streamline requires Linux with a 2.6 kernel. The standard release is built for Debian 6.0 ("squeeze') and CentOS 5.

The performance of the server is largely dependent on the performance of the disks. In particular, it is crucial that the disk driver is operating in DMA mode, rather than PIO (where the processor has to do the data fetching itself).

Licensing

The Packet Ship Streamline server defaults to a "Demonstration Edition" mode with limited bandwidth (25Mbit/sec) and number of streams (5). **This is licensed for evaluation, development and demonstration purposes only, not for use in a production system.**

For use in production and for higher capacity the server requires a cryptographically signed licence file to operate, which sets the licensed capabilities and capacities of the server. Licence files are locked to a particular server's MAC address, and need to be generated by Packet Ship specifically for each hardware installation. Once you start production there is an online service where you can order and download licences yourselves.

If you have purchased a full licence you will receive a licence XML file, and this should be copied to `/etc/packetship/licence.xml` (although the location can be changed in configuration if required). If you copy the licence file after installation, the daemon will need to be restarted to obtain the extra capacity.

Installing on Debian

The Debian packages (.debs) provided will install on either Debian 6.0 ('squeeze') or 5.0 (lenny'), or other distributions based on Debian, such as Ubuntu 10.04 LTS.

The packages provided are:

- **ps-streamd**: The main video server daemon
- **ps-index-mpeg2ts**: A tool for creating index files from MPEG-2 Transport Streams for VBR streaming and trick mode fast-forward / rewind
- **ps-analyse-mpeg2ts**: A tool for analysing the rate profile and other aspects of an MPEG-2 Transport Stream

Using an APT source

If you have your own APT package source set up, you can just place the package files in it, then use '`apt-get`' to install them on target machines:

```
# apt-get install ps-streamd ps-index-mpeg2ts ps-analyse-mpeg2ts
```

This is the best way to maintain a number of servers, since you can add upgrades to your distribution and have them automatically upgraded on the servers which use it.

Installing the packages manually

If you want to install the packages manually, use '`dpkg`' to install them: (your version numbers may vary: tab completion is your friend!)

```
# dpkg –i ps-streamd_3.0.1-1_i386.deb
# dpkg -i ps-index-mpeg2ts_3.0.1-1_i386.deb
# dpkg -i ps-analyse-mpeg2ts_3.0.1-1_i386.deb
```

If you are installing on a 64-bit system use the 'amd64' packages instead of 'i386'.

What happens underneath

Like most Debian server packages, the daemon package installs the server binary, the configuration file and the '`init.d`' files in the usual places, then starts them up. For more details of what goes where, see the section on manual installation below.

Installing on CentOS / Red Hat

The Red Hat packages (.rpms) provided will install on CentOS 5, RHEL 5 and most versions of Fedora.

Using a repository

If you have your own 'yum' repository set up, you can just place the package files in it, then use '`yum`' to install it on target machines.

```
# yum install ps-streamd ps-index-mpeg2ts ps-analyse-mpeg2ts
```


This is the best way to maintain a number of servers, since you can add upgrades to your distribution and have them automatically upgraded on the servers which use it.

Installing the packages manually

If you want to install the packages manually, and use '`rpm`' to install them: (your version numbers may vary: tab completion is your friend!)

```
# rpm –i ps-streamd_3.0.1-1.i386.rpm
# rpm –i ps-index-mpeg2ts_3.0.1-1.i386.rpm
# rpm –i ps-analyse-mpeg2ts_3.0.1-1.i386.rpm
```

Note on CentOS/Red Hat firewall

By default, the standard RTSP port (tcp/554) is blocked by the CentOS / Red Hat firewall. To enable external access to the RTSP server, enable port 554 input using the "`system-config-securitylevel`" command:

```
# system-config-securitylevel
```

Select "Customise" and add "`rtsp:tcp`" to "Other ports". If you then list the iptables rules:

```
# iptables -L
```

You should see a line "`ACCEPT ... state NEW tcp dpt:rtsp`".

Manual installation

If you need to install Packet Ship Streamline on a distribution where the standard package doesn't work, and you can't get '`alien`' to do it for you, you may need to manually install the software. In any case, it's useful and interesting to know what goes where…

If you have specially requested it, we will have supplied you with a zipped tarball of the software called something like `packet-ship-streamline-3.0.1-i386-deb.tgz`. This is a flat group of files which you need to copy manually – we haven't attempted to structure it to unpack at the root directory because that would be dangerous without understanding your system. As usual, to unpack the tarball, move it to an empty directory and type:

```
# tar xfvz packet-ship-streamline-3.0.1-i386-deb.tgz
```

You'll then have a directory full of the following files:

- `ps-streamd` (server binary)
- `streamd.cfg.xml` (configuration file)
- `ps-streamd.init.d` (init.d start/stop script)
- `ps-streamd.logrotate.d` (logrotate script)
- `ps-index-mpeg2ts` (indexing tool)
- `ps-analyse-mpeg2ts` (analysis tool)

Binaries

The server binary, `ps-streamd`, needs to be copied to `/usr/sbin/`, or your usual place for system binaries. It should be owned `root.root` and (for security) writeable only by `root`. There's no particular reason to stop ordinary users reading or executing it since they cannot listen to the system RTSP port, and may wish to create their own non-privileged service – beware however that video serving is a heavy load for a multi-user machine, and you may wish to prevent them running it for this reason.

```
# cp ps-streamd /usr/sbin/
# chown root.root /usr/sbin/ps-streamd
# chmod 755 /usr/sbin/ps-streamd
```

The tool binaries can be used by ordinary users and hence can live in `/usr/bin/`. They should also be owned `root.root` and (for security) writeable only by `root`.

```
# cp ps-index-mpeg2ts ps-analyse-mpeg2ts /usr/bin/
# chown root.root /usr/bin/ps-index-mpeg2ts /usr/bin/ps-analyse-mpeg2ts
# chmod 755 /usr/bin/ps-index-mpeg2ts /usr/bin/ps-analyse-mpeg2ts
```

Configuration files

The configuration file `streamd.cfg.xml` needs to be copied to a new directory `/etc/packetship`. Note that this location for the configuration file is 'hardwired' into the server binary, but can be overridden by passing an alternate configuration file as a command-line argument (this is the only command-line option it has).

```
# mkdir /etc/packetship
# cp streamd.cfg.xml /etc/packetship/
```

Any licence file purchased separately also needs to be copied into `/etc/packetship` as `licence.xml` (it will have a customer and machine-specific filename by default).

```
# cp licence-MyCo-345B452A0C29.xml /etc/packetship/licence.xml
```

Start/stop scripts

If your system uses System V init.d scripts, you can copy the script provided into `/etc/init.d/`, and create the usual links from the `/etc/rc2.d` directory, plus `/etc/rc3.d`, `rc4.d` and so on as well if your system ever gets to runlevels above 2.

```
# cp ps-streamd.init.d /etc/init.d/ps-streamd
# ln –sf /etc/init.d/ps-streamd /etc/rc2.d/S20ps-streamd
```

Log directories

By default – although this is configurable – the daemon puts its log files in `/var/log/packetship` – so this needs to be created:

```
# mkdir /var/log/packetship
```


Content directory

The conventional place to store media content is `/usr/local/share/media/`:

```
# mkdir –p /usr/local/share/media
```

Log rotation scripts

If you use '`logrotate`' to maintain your logs, you can copy the `ps-streamd.logrotate.d` script into the `/etc/logrotate.d` directory. The defaults should be sensible, but check your `logrotate` manual if you want to change them.

```
# cp ps-streamd.logrotate.d /etc/logrotate.d/ps-streamd
```

Trying it out

If you installed from the packages your daemon should already be running, but just for the experience, stop it now. If you installed manually, skip this bit.

```
# /etc/init.d/ps-streamd stop
Stopping Packet Ship streamer daemon: ps-streamd.
```

Now restart the daemon:

```
# /etc/init.d/ps-streamd start
Starting Packet Ship streamer daemon: ps-streamd.
```

The daemon should start cleanly with no errors here.

Watching the processes

If you use '`ps axf`' you should see two ps-streamd processes:

```
# ps axf
…
27415 ?        Ss     0:00 /usr/sbin/ps-streamd
27416 ?        Sl     0:00  \_ /usr/sbin/ps-streamd
…
```

There are two processes because the daemon has an in-built "watchdog" parent process which will restart the server in the unlikely event it should fail. This can be disabled in configuration if another watchdog system (e.g. init) is being used.

You can also see what's happening in the log:

```
# less /var/log/packetship/streamd.log
```


Trying a connection

To test if the RTSP server is listening, you can Telnet to it:

```
# telnet localhost 554
Trying 127.0.0.1
Connected to localhost.
Escape character is '^]'
```

To stop the telnet, press CTRL-] then ENTER, then type 'close'.

If you look in the streamer log, you should see your connecting starting and stopping:

```
# tail /var/log/packetship/streamd.log
…
Mon 21 Feb 12:43:43.056 [2]: RTSP 'rtsp' received connection from 127.0.0.1:36101 ()
Mon 21 Feb 12:44:25.975 [2]: RTSP: 127.0.0.1:36101 disconnected
```

If you're feeling really brave, you can try faking a simple RTSP request!

```
# telnet localhost 554
Trying 127.0.0.1
Connected to localhost.
Escape character is '^]'
OPTIONS * RTSP/1.0                              < you type this
                                                < press ENTER again
RTSP/1.0 200 OK
Content-Length: 0
Server: Packet Ship RTSP Server v3.0.1
Date: Mon, Feb 21 2011 12:46:22 GMT
Public: GET_PARAMETER, SET_PARAMETER, PLAY, PAUSE, DESCRIBE, OPTIONS,
 SETUP, TEARDOWN
```

Or even try to start a (bogus) stream:

```
# telnet localhost 554
Trying 127.0.0.1
Connected to localhost.
Escape character is '^]'
SETUP rtsp://localhost/foo RTSP/1.0               < you type this
Transport: RAW/RAW/UDP;unicast;client_port=11111  < and this
                                                   < ENTER again
RTSP/1.0 404 Not found
Content-Length: 0
Server: Packet Ship RTSP Server v3.0.0
Date: Mon, Feb 21 2011 12:47:26 GMT
```

In this case, you can see in `/var/log/packetship/streamd.log` that the server has been asked to play 'foo', but can't find it. If you see that, then everything is connected up and working – now you need to configure it and add the content!

Quick start

To get your first stream running, you will need to

a) Obtain a file (asset)

b) Upload it to the server

c) Create an index file for it

d) Stream it from a client

The following is a quick guide to get you started...

Obtaining a file

The Packet Ship Streamline video server requires files to be encoded in either MPEG-2 or H.264 (also known as MPEG-4 Part 10, or MPEG-4 AVC), packaged into an MPEG-2 Single Program Transport Stream (SPTS).

We assume you have a source of videos in this format; alternatively you can download our favourite test stream "Big Buck Bunny" (Creative Commons licensed) from:

http://downloads.packetship.com/test/bunny.ts

Although the video is short it's very high quality (H.264 1080p) and hence is over 700MB long.

We'll assume this is the asset you want to use for the rest of this quick start guide. If not, just substitute your filename where "`bunny.ts`" appears.

Uploading to the server

The file (bunny.ts) needs to be uploaded to the video server. The place to put content in the standard configuration is `/usr/local/share/media`:

```
$ scp bunny.ts root@vodserver:/usr/local/share/media/
```

Because of the standard mappings set up, this will then be accessible as "`media/bunny.ts`", as we'll see.

We'll assume in the following that this server is called "vodserver" - substitute with its real name or IP address in the RTSP URLs below.

Indexing

Our encoding of Big Buck Bunny is highly variable bitrate, averaging around 10Mbit/sec but peaking at over 25Mbit/sec (during the initial zoom into the rabbit hole). In order to let the server know what rates to send at through the video, and to enable trick mode (visual fast-forward/rewind), the server needs an "index file". This is created with the `ps-index-mpeg2ts` tool that you installed:

(on the video server, as root)

```
# cd /usr/local/share/media/
# ps-index-mpeg2ts bunny.ts
Packet Ship MPEG2 Transport Stream indexer version 3.0.1
Indexing bunny.ts into bunny.ts.psi
Output index format: PSI2
PMT on PID: 66
Detected H.264 video stream on PID: 69
Creating index of 596 blocks
Duration: 596.198   seconds
Done
```

This process takes a minute or so, because although the video is short (10 minutes), it is very high bandwidth and hence a large file.

That's all you have to do to "ingest" a file assuming you're using simple file mapping. You don't need to restart the server or anything.

Streaming the video

The URL to use on your video client or set-top box for streaming the video over RTSP will be `rtsp://vodserver/media/bunny.ts`

If you have a set-top box, this will have it's own way of starting an RTSP stream. In some cases you can enter this URL directly into the browser; in others, you need to write some Javascript to do it, which unfortunately is different in every case!

The simpler way to test streaming is to use VLC, an excellent streaming client available for all platforms. You can download it for free from www.videolan.org.

If you're using Linux you can run it directly from the command line:

```
$ vlc rtsp://vodserver/media/bunny.ts
```

(or it might be localhost if you've installed the video server on your desktop machine).

If you're using Windows or Mac, or just prefer the GUI interface on Linux, start VLC and then select "Media / Open Network Stream" (or CTRL-N). Enter the network URL "`rtsp://vodserver/media/bunny.ts`", then click "Play".

If it works

If it works, congratulations! You should also be able to use the "faster" and "slower" buttons and reverse button at either end of the progress bar to fast-forward and rewind.

You might also like to look at the logs in /var/log/packetship/streamd.log to see the action from the server's point of view. You should see something like this.

```
Fri 18 Feb 16:06:08.515 [2]: Stream rtsp#1 (3bd2927266e00001) starting, playing media/bunny.ts
Fri 18 Feb 16:06:41.365 [2]: Stream rtsp#1 (3bd2927266e00001) stopped
```


If you would like to see more of what's going on behind the scenes, change the **<log level>** in `streamd.cfg.xml` to 3 and restart the server (see below). This is a good idea when testing in any case, but the level should be reduced to 2 in production otherwise you will get swamped!

Here's a segment of a level 3 log showing the startup of the bunny.ts stream:

```
Fri 18 Feb 16:17:18.345 [3]: RTSP 'rtsp' received connection from 192.168.0.52:1193
(00:02:E3:34:F8:3B)
Fri 18 Feb 16:17:18.356 [3]: RTSP/1.0 request: OPTIONS from 192.168.0.52:1193 for
rtsp://barque/media/bunny.ts
Fri 18 Feb 16:17:18.356 [3]: Response: 200 OK
Fri 18 Feb 16:17:18.357 [3]: RTSP/1.0 request: DESCRIBE from 192.168.0.52:1193 for
rtsp://barque/media/bunny.ts
Fri 18 Feb 16:17:18.357 [3]: DESCRIBE from 192.168.0.52:1193 for rtsp://barque/media/bunny.ts
(application/sdp)
Fri 18 Feb 16:17:18.358 [3]: Mapped asset ID media/bunny.ts to file /usr/local/share/media/bunny.ts
Fri 18 Feb 16:17:18.358 [3]: Content-Type application/sdp selected
Fri 18 Feb 16:17:18.358 [3]: Asset pattern * selected
Fri 18 Feb 16:17:18.358 [3]: Response: 200 OK
Fri 18 Feb 16:17:18.373 [3]: RTSP/1.0 request: SETUP from 192.168.0.52:1193 for
rtsp://barque/media/bunny.ts/
Fri 18 Feb 16:17:18.373 [3]: SETUP from 192.168.0.52:1193 (00:02:E3:34:F8:3B) for
rtsp://barque/media/bunny.ts/
Fri 18 Feb 16:17:18.373 [3]: Selected output 'udp-rtp' from transport
RTP/AVP;unicast;client_port=1194-1195
Fri 18 Feb 16:17:18.373 [3]: Mapped asset ID media/bunny.ts to file /usr/local/share/media/bunny.ts
Fri 18 Feb 16:17:18.374 [3]: Attempting admission control for stream 2
Fri 18 Feb 16:17:18.374 [3]: Checking stream 2 for 'access'
Fri 18 Feb 16:17:18.374 [3]: Stream 2 passed check for 'access'
Fri 18 Feb 16:17:18.374 [3]: Stream 2 claimed 9.76683 Mbit/sec from 'total-capacity'
Fri 18 Feb 16:17:18.374 [3]: Stream 2 claimed 9.76683 Mbit/sec from 'localnet-192-bandwidth'
Fri 18 Feb 16:17:18.374 [3]: Stream 2 claimed 9.76683 Mbit/sec from 'Legacy licence'
Fri 18 Feb 16:17:18.374 [3]: Creating TS timing adjustment filter
Fri 18 Feb 16:17:18.374 [3]: Creating RTP output filter with payload type 33
Fri 18 Feb 16:17:18.374 [3]: Creating rate profile 'limit' with limits (500000,0.1)-(25000000,5)
Fri 18 Feb 16:17:18.374 [3]: Creating UDP output with 1316 byte packets on 0.0.0.0:0
Fri 18 Feb 16:17:18.374 [3]: Bound to local address 0.0.0.0:48171
Fri 18 Feb 16:17:18.375 [3]: Assigning stream #2 to group realtime
Fri 18 Feb 16:17:18.375 [3]: Stream rtsp#2 (2a018d175c1d0002) destinations:
Fri 18 Feb 16:17:18.375 [3]:  - 192.168.0.52:1194
Fri 18 Feb 16:17:18.375 [2]: Stream rtsp#2 (2a018d175c1d0002) starting, playing media/bunny.ts
Fri 18 Feb 16:17:18.376 [3]: Starting realtime stream thread
Fri 18 Feb 16:17:18.377 [3]: Stream rtsp#2 (2a018d175c1d0002) switching to asset media/bunny.ts
Fri 18 Feb 16:17:18.377 [3]: Creating file input
Fri 18 Feb 16:17:18.377 [3]: Direct disk read (O_DIRECT) enabled
Fri 18 Feb 16:17:18.377 [3]: Index video PID filtering enabled
Fri 18 Feb 16:17:18.377 [3]: Index timing adjustment enabled
Fri 18 Feb 16:17:18.377 [3]: File input reading /usr/local/share/media/bunny.ts
Fri 18 Feb 16:17:18.378 [3]: File input checking for index /usr/local/share/media/bunny.ts.psi
Fri 18 Feb 16:17:18.378 [3]: Got index file /usr/local/share/media/bunny.ts.psi with 256B blocks,
65536B buffers
Fri 18 Feb 16:17:18.378 [3]: Index gives rate of 9766828 bits/sec
Fri 18 Feb 16:17:18.405 [3]: Stream rtsp#2 (2a018d175c1d0002) started
Fri 18 Feb 16:17:18.405 [3]: Response: 200 OK
Fri 18 Feb 16:17:18.407 [3]: RTSP/1.0 request: PLAY from 192.168.0.52:1193 for rtsp://barque/media/
bunny.ts
Fri 18 Feb 16:17:18.408 [3]: Got range 0--1
Fri 18 Feb 16:17:18.408 [3]: Stream rtsp#2 (2a018d175c1d0002) play from 0
Fri 18 Feb 16:17:18.408 [3]: Response: 200 OK
Fri 18 Feb 16:17:18.408 [3]: Stream rtsp#2 (2a018d175c1d0002) alignment event
Fri 18 Feb 16:17:18.408 [3]: Stream rtsp#2 (2a018d175c1d0002) seek to 0 event
Fri 18 Feb 16:17:18.408 [3]: Seek to 0 requested in /usr/local/share/media/bunny.ts
Fri 18 Feb 16:17:18.408 [3]: Index file looking for NPT 0
Fri 18 Feb 16:17:18.408 [3]: Snapped to front of file
Fri 18 Feb 16:17:18.408 [3]: Index says to seek to 0
Fri 18 Feb 16:17:18.411 [3]: Stream rtsp#2 (2a018d175c1d0002) change speed to 1 event
Fri 18 Feb 16:17:18.701 [3]: RTSP/1.0 request: GET_PARAMETER from 192.168.0.52:1193 for
rtsp://barque/media/bunny.ts
Fri 18 Feb 16:17:18.702 [3]: Response: 200 OK
```

This shows the standard OPTIONS – DESCRIBE – SETUP – PLAY sequence of an RTSP session, and also hints at many of the topics covered in the rest of this guide.

If it doesn't work

That's a pity! Here is some basic troubleshooting you can do to sort out any problems. If you get stuck, just e-mail support@packetship.com and we'll be glad to help.

VLC gives an error

If VLC gives an immediate error, check the URL was entered correctly, and you have the correct server address. Then take a look at the log in `/var/log/packetship/streamd.log` and see if you can see any activity like that above.

If you can't see any activity, it may be that the RTSP port (port 554) is blocked on your server's firewall – it is by default on CentOS / RedHat. See the instructions in the installation section above on how to allow RTSP traffic in.

No video

If the stream appears to start, but there is no video, it may be the opposite problem – your client computer's firewall may not be allowing the UDP traffic in. You may want to turn your firewall off briefly to check this, and if that fixes it, enter a rule allowing it for the future.

Stalling video

If you get a picture but it stalls or has errors, either your network or your computer may not be able to handle the extremely high bitrate of this file. Check the server log for errors about "lag" which might indicate a server-side problem (although this is highly unlikely with only one stream).

The solution is to use a higher powered computer or a set-top box which does it in hardware, or failing that to try a lower bitrate asset – ask us for some samples.

No trick play

VLC has only recently offered proper fast-forward / rewind trick play controls in the standard user interface. Check you have the latest version installed, at least 1.1.7.

Still not resolved?

If you still can't resolve it, please contact support@packetship.com. It would be useful if you could set the **`<log level>`** in `streamd.cfg.xml` to "3", restart the server and try the stream again, then send us the log file `/var/log/packetship/streamd.log`.

Systems Administration

The Packet Ship Streamline server, `ps-streamd`, is a standard Unix daemon and is managed in the same way as any other. The following is a quick guide for systems administrators and anyone else who needs to start, stop, restart and observe the daemon as it is running.

Binaries

The daemon is installed by default as `/usr/sbin/ps-streamd`. The `ps-index-mpeg2ts` and `ps-analyse-mpeg2ts` utilities are installed in /usr/bin.

init.d script

Like most daemons `ps-streamd` comes with a System V init script `/etc/init.d/ps-streamd`. This takes the following command parameters:

- **start**: Starts the daemon
- **stop**: Stops the daemon
- **restart**: Stops the daemon and then starts it again (hard restart)
- **reload**: Reloads configuration but keeps any existing streams running (soft restart)

For example:

```
# /etc/init.d/ps-streamd start
# /etc/init.d/ps-streamd reload
# /etc/init.d/ps-streamd stop
```

The installation package installs links to the init.d script into the default `rc<n>.d` directories.

The "reload" command actually sends a SIGHUP to the daemon – this can be done directly if an automated way to reload after configuration is required. Note that in the present version it is only the asset catalogue and permanent streams list which is reloaded; other configuration changes require a hard restart.

Configuration

The `ps-streamd` daemon's core configuration file is `/etc/packetship/streamd.cfg.xml`. It may also search for extra configuration files in `/etc/packetship/streams/` and `/usr/local/share/media/`.

Fully licenced versions are enabled by a licence file in `/etc/packetship/licence.xml`. The same licence may also cover other Packet Ship products.

Logging

By default `ps-streamd` logs everything to `/var/log/packetship/streamd.log`. It also installs a logrotate script in `/etc/logrotate.d/ps-streamd` which rotates this on a weekly basis with a 2 week history.

Log lines consist of a date stamp and millisecond clock, a log level (in square brackets) and the log text. Filtering for "[1]" is a good way to detect errors for automatic monitoring.

Processes & Threads

The daemon runs as two processes. The parent is a 'watchdog' process which will restart the child (which is the main operation) in the unlikely event it should exit unexpectedly. It will start doing this quite aggressively, but will then back off exponentially up to a minute between retries.

The daemon makes heavy use of threads. There is a baseline of around 10-20 threads (depending on controller options), plus one thread for each client connection (RTSP or HTTP). RTSP-TCP and HTTP streams each generate an addition thread each.

The realtime UDP output is performed by a smaller number of shared threads, ideally one per CPU. These depend on access to the kernel's RTC device, specifically periodic interrupts. See the realtime scheduler section at the very end of this document for more details.

General server configuration

There are a number of general configuration options which are standard across all Packet Ship server daemons:

Logging

The logging output for the server is configured with the **<log>** element at the top level of the configuration file.

The **<log>** element has the following attributes:

- The **level** attribute gives the level of logging produced, as follows:

Level	Output
0	Nothing
1	Errors only
2	Summary of major activities
3	Details of activities
4	*Debugging information*
5	*Full dumps of everything*

  Production release versions (which you will receive by default) only have code to log up to level 3 – this makes them more efficient. You won't need a debug version unless we specifically ask you to give us information on an issue. The default level is 2, which is the level at which it is sensible to run most production systems.

- The **file** attribute gives the log file to log to, for production versions. The default for `ps-streamd` is `/var/log/packetship/streamd.log`.

- The **timestamp** attribute gives the format of the timestamp prefixed to each line in the log file. The format is that used by '`strftime`' (see '`man strftime`'), and can include both variables prefixed with % and fixed text. In addition, the following special fields can be used:

Field	Default log file
`%*S`	Seconds including milliseconds
`%*L`	Log level (1-5, as above)

  The default is "`%a %d %b %H:%M:%*S [%*L]: `", which produces lines like this:

```
Tue 29 Jan 14:30:04.582 [2]: ...
```

The following is the default **<log>** configuration for `ps-streamd` as shipped, which is probably fine for the vast majority of installations.

```
…
  <log level="2" file="/var/log/packetship/streamd.log"
   timestamp="%a %d %b %H:%M:%*S [%*L]:"/>
…
```

Background daemon

By default the server becomes a background daemon and detaches itself from the terminal that started it, and logs to the file specified above. However for testing it's sometimes convenient to have the server run in foreground on the terminal and output logging to it. This is controlled by the **daemon** attribute of a top-level <**background**> element. It defaults to "yes", set it to "no" to run in foreground.

```
…
  <background daemon="yes"/>
…
```

Watchdog restart

In the unlikely event the server should fail, there is a built-in 'watchdog' process that can restart it automatically. This is controlled by the **restart** attribute of a top-level **<watchdog>** element. It defaults to "yes", set it to "no" to disable the watchdog restart.

```
…
  <watchdog restart="yes"/>
…
```

User/group identity

The server daemon is usually started as user 'root' in order to allow it to obtain the necessary restricted network ports, enable real-time priority and other root-only functions. However, once it has done this, as an additional security precaution in case of attack it is able to drop its root permissions and become an ordinary user – typically user "nobody" and group "nogroup" (Debian) or "nobody" (Red Hat), that have no permissions at all.

The user to switch to is configured with the **user** and **group** attributes of a **<security>** element at the top level of the configuration file:

```
…
  <security user="nobody" group="nogroup"/>
…
```

If absent, the server remains as the user it is run as (usually 'root').

Licence file

The licence file which governs the licensed features of the server is usually kept in `/etc/packetship/licence.xml`. Should there be any reason to change this, the location can be set with the **file** attribute of a **<licence>** element at the top level:

```
…
  <licence file="/etc/packetship/licence.xml"/>
…
```


General stream control

Overall stream control parameters of the server are configured in a **<streams>** element at the top level of `streamd.cfg.xml`.

Keepalive timeouts

Since streaming clients – particularly set-top boxes – are prone to disappear without warning, some method of repeated "keepalive" is required to maintain the stream, otherwise orphan streams could continue indefinitely (or until the asset runs out), clogging up resources. This is particularly true of RTSP clients, where the protocol specifies that a client disconnection is *not* sufficient reason to stop the stream. In this case the convention is to send a PLAY, SET_PARAMETER or GET_PARAMETER request occasionally (at a rate defined by a Session header parameter) to tell the server the stream is still required.

The **timeout** attribute of a **<keepalive>** element in the **<streams>** element allows configuration of the maximum time between keepalive signals before the server will kill the stream. The timeout can be expressed either in seconds or in a simple textual format: "5 mins", "1 hour", etc. The default of 60 seconds is slightly more than the default advertised timeout, allowing some slack for clients that keep it close to the edge.

```
<streams>
  …
  <keepalive timeout="60"/>
  …
</streams>
```

f the keepalive timeout is set to zero, there will be no timeout applied and streams will potentially continue forever. This is not good practice, and should only be used when clients do not provide the usual keepalive requests. In this case, it is a good idea to also force teardown on disconnect in the RTSP controller (see below) so that if clients do fail there is something which will close the stream.

Empty packets

Some clients (notably VLC) do not recognise the (semi-)standard RTSP method of announcing the end of a stream, but do accept an empty data packet as marking the end and triggering an immediate clean closedown. This is safe to do for all clients in the vast majority of cases, and hence is the default, but can be disabled if required.

The **packet** attribute of an **<empty>** element in the **<streams>** element governs whether the empty packet is sent:

```
<streams>
  …
  <empty packet="yes"/>
  …
</streams>
```


Controllers

Controllers initiate and manage streams, either on behalf of an external client or through configuration. They can also provide services for other parts of the system to use – for example the XMLMesh connection provided by the **soap** controller.

The currently supported Controllers are:

- **rtsp**: allowing streams to be started, stopped and managed by standard RTSP clients
- **http**: providing basic progressive download through HTTP
- **hls**: supporting Apple's HTTP Live Streaming for Mac, iPhone and iPad.
- **soap**: allowing streams to be started, stopped and managed through SOAP/XML messaging, either through a built-in HTTP server or over XMLMesh.
- **permanent**: allowing the creation of permanently running streams such as broadcasts and video loops

Controllers are configured in the **`<controllers>`** element of the top level of `streamd.cfg.xml`. Each controller has an element within this with the type of the controller. If multiple controllers of the same type are required, they should be given unique **`id`** attributes, otherwise the controller is just named after the type.

RTSP

The RTSP controller (element **`<rtsp>`**) provides a standards-compliant RTSP server for on-demand streaming from IPTV set-top boxes and RTSP-compliant PC clients such as VLC. It is possible to have multiple RTSP controllers with different configuration running on different ports.

RTSP server

By default, the RTSP controller listens on the standard RTSP port, 554/tcp. In order to do so, the process must be started as root (as it is when installed), although it drops root privileges after starting. You may want to change the port either to allow non-root use, or to provide multiple independent RTSP controllers, perhaps configured differently.

- The **`port`** attribute of the **`<server>`** element in **`<rtsp>`** configures the port number to listen on. The default is 554.

  Note: In CentOS/RHEL this port will need to be enabled for input in the firewall, as described in the installation instructions.

- The **`address`** attribute of the same element gives an optional bind address for multi-homed hosts. This can improve security allowing only access from a controlled interface. The default is "0.0.0.0", or all interfaces.
- The **`backlog`** attribute configures the listen backlog – that is, the number of new connections which can be queued while the controller is handling a previous one. Because the listener is multi-threaded it can accept new connections reasonably quickly, so it is rarely necessary to change it from its default of 5.

- The **timeout** attribute gives the length of timeout (in seconds, default 90) on the socket. After this timeout, if no data is received, the RTSP server closes the connection, saving a thread. Note that this is independent of the overall stream keepalive timeout described above.

Each concurrent RTSP connection is handled by its own thread, pulled from a pool of active threads. The **max-threads** attribute of the **<server>** element sets the maximum number of connection threads that can be available at any one time. The **min-spare** attribute sets the number of spare threads to keep lying around in case a new connection arrives. Its default is 1, which ensures there is always an available thread for any single new connection.

```
<rtsp>
  …
  <server port="554" address="0.0.0.0" backlog="5" timeout="90"
          max-threads="100" min-spare="1"/>
  …
</rtsp>
```

It is worth noting that RTSP is designed so that clients need not keep a connection open for the whole duration of the stream – they can open and close the connection as they see fit, as long as they keep sending 'keepalives' (PLAY or GET-PARAMETER commands). Hence there is no inevitable connection between the number of threads available and the number of sessions that can be started.

However, most clients do indeed keep the RTSP connection open for the entire time, in which case the number of threads does represent a limit on the number of concurrent streams, on top of any admission control limits described below.

Session management

The **<sessions>** element at the top level of the <rtsp> element provides a number of configuration options for session management.

Autoplay control

Many clients wrongly assume that a SETUP request also starts the stream playing, when in fact they should issue a separate PLAY request. To work round this, you can turn on an auto-play mode where the stream is immediately started on SETUP.

The **immediate** attribute of a **<play>** element in the **<sessions>** element turns auto-play on.

```
<rtsp>
  …
  <sessions>
    …
    <play immediate="yes"/>
    …
  </sessions>
  …
</rtsp>
```


Enabling this is safe for most clients, since if the client does send a PLAY it will simply continue the stream which has already started. There is however one case where you would want to disable this, which is when the client sends an explicit seek to zero (PLAY with Range: 0) after SETUP. VLC is one common client that does this. In this case the stream will be started automatically by the SETUP and will then be rewound and started again by the PLAY. This causes a stutter at the beginning of the stream and also increases the load on the server because of the additional seek.

End-of-stream announcements

In an extension to RTSP, some clients accept ANNOUNCE commands from the server to indicate a clean end of the stream, rather than watching for cessation of data which would take longer, and may indicate a failure. As previously noted, VLC (apparently alone) accepts an empty packet as signifying the end of stream.

An **<announce>** element in **<sessions>** enables end-of-stream announcements. Different clients may require different headers in the ANNOUNCE command to recognise it as an end-of-stream marker:

- The **header** attribute sets the header name to be added. This can be in lower-case and will be converted to the conventional title-case (first letter upper) by the server.
- The **value** attribute sets the value of the added header.

The server will include a 'Session:' header with the correct session ID in any case, along with the standard 'Server:' and 'CSeq:' headers.

The standard configuration is as follows:

```
<rtsp>
  …
  <sessions>
    …
   <announce header="notice" value="2101 End of Stream"/>
    …
 </sessions>
  …
</rtsp>
```

This results in the server sending something like the following announcement:

```
ANNOUNCE rtsp://10.0.0.1/foo RTSP/1.0
CSeq: 1
Server: Packet Ship RTSP Server v2.3.1
Session: 123456789
Notice: 2101 End of Stream
```

The "Notice: `2101 End of Stream`" format comes from a proposal in the RTSP1.1 working group.

If your client does not understand ANNOUNCE commands or – more commonly – does not handle server-initiated commands at all, you can disable this feature by removing the **<announce>** element altogether.

End of stream XMLMesh notification

It is often useful for an external process to be able to see when a stream has ended through an XMLMesh message. This requires that the **<soap>** controller be configured – see there for more information.

This message is enabled by the **end** attribute of a **<notify>** element inside the **<sessions>** element:

```
<rtsp>
  …
  <sessions>
    …
   <notify end = "yes"/>
    …
 </sessions>
  …
</rtsp>
```

Session timeout parameter

The RTSP server can append a 'timeout' parameter to the Session: header, to inform clients which require it of the keepalive timeout in use. This is configured with the **parameter** attribute of a **<timeout>** element in **<sessions>**, as follows:

```
<rtsp>
  …
  <sessions>
    …
    <timeout parameter="50"/>
    …
  </sessions>
  …
</rtsp>
```

If the **<timeout>** element is absent, no timeout parameter is added (some clients can't handle it)

Note that this timeout is only the one advertised to the client. The actual timeout in use is the one set in the **<keepalive>** element. It is wise to use a somewhat larger timeout in practice than is advertised to the client in case the client takes it literally as the interval at which to take keepalives. A sensible client will apply it's own margin on the timeout, though, so making it too small may produce a large number of unnecessary keepalives.

Teardown on disconnect

If you have mis-implemented clients which do not issue explicit TEARDOWNs but assume that closing the RTSP socket is enough, then you can force a teardown of the session on disconnection of the client with a teardown attribute of a **<disconnect>** element in the **<sessions>** element:

```
<rtsp>
  …
  <sessions>
    …
    <disconnect teardown="yes"/>
    …
  </sessions>
  …
</rtsp>
```


This option is also useful to protect the server from orphan sessions if you have to disable the normal keepalive timeouts, as above.

Transport mapping

A 'transport' is a stack of streaming protocols encapsulating the media data as it is sent to a client. As part of the SETUP request, the client gives a list of possible transports it can accept, in the form {*transport*}/{*profile*}/{*lower-transport*}. Unfortunately, the RTSP standard doesn't give the suggested value for the transport specifier for raw UDP transmission, and different video server manufacturers have used their own – e.g. "RAW/RAW/UDP" and "MP2T/H2221/UDP".

Hence to provide compatibility for clients which implement RTSP for other servers, you can configure a mapping table from RTSP transport specifiers to outputs which define the streaming protocol – see "Outputs" below.

The transport mapping table

The mapping table is contained in **<map>** elements within a top-level **<transports>** element in the **<rtsp>** element. Each **<map>** element has the following attributes:

- The **transport** attribute gives the transport type that will be quoted in RTSP
- The **output** attribute gives the ID of the output to be used for this transport
- The **score** attribute gives an optional score; if clients offer to accept multiple transports, the one with the highest score will be used. The default is 0.

The following is the standard **<transports>** table in the default configuration. There is no particular reason to modify this unless you find that your client is quoting a different transport specifier, in which case you can add it to the map.

```
<rtsp>
  ...
  <transports>
    <map transport="RAW/RAW/UDP"     output="udp"      score="2"/>
    <map transport="MP2T/H2221/UDP"  output="udp"      score="2"/>
    <map transport="RTP/AVP/UDP"     output="udp-rtp"  score="1"/>
    <map transport="RTP/AVP"         output="udp-rtp"  score="1"/>
    <map transport="RAW/RAW/UDP/TCP" output="rtsp-tcp-interleaved"/>
    <map transport="RTP/AVP/TCP"     output="rtsp-tcp-interleaved-rtp"/>
  </transports>
  ...
</rtsp>
```

The effect of the scoring in the above is to choose the most bandwidth efficient protocol if multiple options are given.

DESCRIBE responses

Although many RTSP clients don't bother to use DESCRIBE at all, some do so and expect some valid information about the asset requested. One or more **<describe>** elements inside the **<rtsp>** element allow you to set the data returned from a DESCRIBE from its textual content. The **type** attribute of **<describe>** gives the content-type to be returned. For RTSP compliance this should usually be "application/sdp", which is the default, but some clients written for other servers may require something different.

Fixed response for raw UDP

Here is a simple fixed DESCRIBE configuration with works with most clients which only need generic data, for MPEG-2 Transport Streams in raw UDP:

```
<rtsp>
  …
  <describe type="application/sdp" crlf="yes">
    v=0
    o=- 0 0 IN IP4 0.0.0.0
    s=Packet Ship RTSP session
    c=IN IP4 0.0.0.0
    t=0 0
    m=video 0 udp 33
  </describe>
  …
</rtsp>
```

As you can see, there is very little information present in this, just enough to make it syntactically correct. The only real information is the '33', which indicates MPEG2 Transport Stream.

Also, note that the data is indented in the file to make it look 'pretty' with the rest of the XML. To avoid sending lots of indented lines which would be invalid SDP, the RTSP daemon strips out all common indent from the text (in this case, 6 spaces), so that all the lines begin in the first column as expected.

The **`crlf`** attribute makes the RTSP server convert the line-endings to CR-LF, and should always be used for `application/sdp` format.

Fixed response for RTP

Some clients, such as VLC, use the DESCRIBE response to decide which transport protocol to request in the SETUP phrase. To return a DESCRIBE response indicating that the asset is available as RTP, the 'm' parameter of the SDP result has to change:

```
<rtsp>
  …
  <describe type="application/sdp" crlf="yes">
    v=0
    o=- 0 0 IN IP4 0.0.0.0
    s=Packet Ship RTSP session
    c=IN IP4 0.0.0.0
    t=0 0
    m=video 0 RTP/AVP 33
    a=rtpmap:33 MP2T/90000
  </describe>
  …
</rtsp>
```

Asset-specific data

The above examples return the same result for all assets and all servers, but some clients require more specific information. To enable this, there are a number of asset- and server-specific pieces of information available in a number of variables that can be used in the DESCRIBE response, prefixed by '$'.

The variables defined as as follows:

Variable	Contents
ntp_time	Current time in NTP (use for a unique value)
server_ip	IP address of RTSP interface
asset	Asset ID requested in DESCRIBE URL
length	Length of asset in **bytes**
duration	Length of asset in **seconds** (calculated from average rate)
rate_bps	Rate of asset in bits per second
rate_kpbs	Rate of asset in kbits (1000 bits) per second

Hence the following DESCRIBE configuration produces a more standard SDP result for MPEG-2 Transport Streams in raw UDP:

```
<rtsp>
  …
  <describe type="application/sdp" crlf="yes">
    v=0
    o=- $ntp_time $ntp_time IN IP4 $server_ip
    s=Packet Ship RTSP session: $asset
    c=IN IP4 $server_ip
    t=0 0
    a=range:npt=0-$duration
    b=AS:$rate_kbps
    m=video 0 udp 33
  </describe>
  …
</rtsp>
```

Variant responses by asset ID

It may sometimes be necessary to vary the DESCRIBE response for different types of asset – for example, specifying RTP transport for one set of assets and raw UDP for another. The **`asset`** attribute of the **`<describe>`** element gives a 'glob' pattern match on the asset ID, and the first **`<describe>`** element which matches is used. The default is "`*`", so a **`<describe>`** element with no **`asset`** attribute (as above) always matches.

The following example shows a specific response which matches any assets in a 'mobile' directory:

```
    ...
    <describe asset="mobile/*" type="application/sdp" crlf="yes">
       ... special response ...
    </describe>
    ...
```

Variant media types

The RTSP daemon can also respond with different DESCRIBE responses depending on the media type requested in the Accept: header. For example, to emulate the non-standard SGI 'sgi-mb' format, you can add the following extra **`<describe>`** element to the **`<rtsp>`** section:

```
<rtsp>
  …
  <describe type="application/x-rtsp-mh">
    StreamID = 0
  </describe>
  …
</rtsp>
```


The first **<describe>** element which matches both **type** and **class** will be used.

HTTP

The HTTP Controller (element **<http>**) provides simple progressive download of streams over TCP using standard Web browser HTTP protocol.

HTTP server

The HTTP server is configured in a **<server>** element inside **<http>**, in a very similar way to the RTSP server:

- The **port** attribute of the **<server>** element configures the port number to listen on. The default is 8080 (http-alt).
- The **address** attribute of the same element gives an optional bind address for multi-homed hosts. This can improve security allowing only access from a controlled interface. The default is "0.0.0.0", or all interfaces.
- The **backlog** attribute configures the listen backlog, default 5.
- The **timeout** attribute gives the length of timeout (in seconds, default 90) on the socket. After this timeout, if no data is sent or received, the HTTP server closes the connection, saving a thread.

Each concurrent HTTP connection is handled by its own thread, pulled from a pool of active threads. The **max-threads** attribute of the **<server>** element sets the maximum number of connection threads that can be available at any one time. The **min-spare** attribute sets the number of spare threads to keep lying around in case a new connection arrives. Its default is 1, which ensures there is always an available thread for any single new connection.

```
<http>
  …
  <server port="8080" address="0.0.0.0" backlog="5" timeout="90"
          max-threads="100" min-spare="1"/>
  …
</http>
```

Output type

The Output to use for HTTP streams is configured in the **output** attribute of a **<stream>** element in **<http>**. This needs to be a 'tcp' output, and should provide an **<http-chunked>** OutputFilter. It is also usual to apply a **<fixed>** RateProfile so that the rate of output is (up to a limit) client-controlled. The standard configuration has a "tcp-http-chunked" Output that does this:

```
<http>
  ...
      <stream output="tcp-http-chunked"/>
  ...
</http>
```


HLS

The HLS controller (element <hls>) provides support for Apple's HTTP Live Streaming protocol, which uses standard HTTP to fetch a playlist file which describes the asset as a large number of small chunks, optionally with multiple variants at different bandwidths for adaptive streaming.

The current version of the server implements version 3 of the protocol as described in Internet Draft "draft-pantos-http-live-streaming-05".

A different approach

Because the Packet Ship Streamline server is a streaming server, not a web server, it takes a different approach to implementing HTTP Live Streaming than the conventional one of exploding a file into thousands of chunks and fixed playlists.

Firstly, the server simply uses the standard asset file, and serves "virtual chunks" from on request by the client using URL parameters to mark the start and end of the chunk (in time values). This avoids the overhead of thousands of tiny files for each asset, and also makes it more flexible, since the chunk size can simply be changed by changing configuration, not re-processing all the files. The same asset file can also be used for other forms of streaming, such as RTSP.

Secondly, the playlists are automatically generated on demand, not stored; this makes it easier to handle streams that are changing as they are output, such as chase-play for broadcast TV.

Thirdly, although the protocol is in theory stateless, the server retains session state as an optimisation for the most common case where the client simply requests successive chunks from the same variant in order. This has a number of advantages:

1. Admission control is applied once at the beginning of the stream, providing true quality of service guarantees.
2. Any authentication or authorisation checks required are done once at the beginning of the stream, improving performance.
3. Buffering is maintained between chunks, reducing disk seek load.

HTTP server

The HLS HTTP server is configured in an identical way to the standard HTTP one (see above), except the default port is 8081 (tproxy):

```
<hls>
  …
  <server port="8081" address="0.0.0.0" backlog="5" timeout="90"
          max-threads="100" min-spare="1"/>
  …
</http>
```

Output type

The Output to use for HLS streams is configured in the same way as in HTTP (see above), and will usually use the same output:

```
<hls>
  ...
  <stream output="tcp-http-chunked"/>
  ...
</hls>
```

Target duration

HTTP Live Streaming defines a "target duration" which is the maximum length of a file chunk. The smaller the value, the larger the number of (virtual) chunks, and the finer grained the stream switching can be (if required).

Because the chunks are virtual, there is not the same cost of thousands of files that would happen if a file was split up in advance. However it does make the playlists very large, and there is a protocol cost in requesting each chunk, and hence a sensible minimum (and the default) is 5 seconds.

The target duration (maximum chunk length) is set by the **`duration`** attribute of a **`<target>`** element inside **`<hls>`**:

```
<hls>
  ...
  <target duration="5"/>
  ...
</hls>
```

Idle session reap time

Because an HTTP Live Streaming client can switch streams (variants) or just stop streaming at any time, there needs to be a mechanism to 'reap' sessions which are no longer being used, freeing up resources for other streams. A sensible value would be twice the maximum chunk length.

The time to wait after a session goes idle before reaping it is set by the **`time`** attribute of a **`<reap>`** element inside **`<hls>`**. The default is 10 seconds:

```
<hls>
  ...
  <reap time="10"/>
  ...
</hls>
```

SOAP

The SOAP controller (element **`<soap>`**) provides a mechanism to start and manage streams from a server-side process, using SOAP messaging. The controller supports both conventional (document-based) SOAP over HTTP and also XMLMesh messaging, which is a peer-to-peer messaging bus architecture which also uses SOAP messages.

The SOAP controller provides a set of message handlers which perform particular functions such as starting, stopping streams and fetching status information. These can be mapped to particular HTTP URLs and/or XMLMesh subjects.

HTTP Server

The standard SOAP HTTP server can be configured with an **<http>** element inside the **<soap>** element. The port on which the server will listen for HTTP requests is set by a **port** attribute of a **<server>** element inside this. If required the local interface address to bind to can be specified in an **address** attribute, which defaults to "localhost".

```
<soap>
  ...
  <http>
    <server port="55480" address=”0.0.0.0”/>
  </http>
  ...
</soap>
```

Note that the HTTP server is **not authenticated** and hence, if enabled on an external interface, should be protected by firewall rules from external access. For security it is disabled in the standard configuration.

XMLMesh Connection

The SOAP controller can accept messages through an XMLMesh message broker, and also output notifications and access verification queries through it if required. The XMLMesh connection is configured in an **<xmlmesh>** element inside the **<soap>** element.

Server connection

The XMLMesh message broker server to connect to is configured with the **host** and **port** attributes of a **<server>** element inside **<xmlmesh>**:

```
<soap>
  ...
  <xmlmesh>
    <server host="localhost" port="29167"/>
  </xmlmesh>
  ...
</soap>
```

Stream end notification

It is often useful for an external process which created a stream to be able to see when a stream has ended through an XMLMesh message. The server has an option to send out an XMLMesh 'packetship.stream.notify' message notifying the end of a stream for both SOAP- and RTSP-controlled streams. For the format of this message, see Application Note "AN-SL-601 XML Messaging". This message is enabled by an **end** attribute of a **<notify>** element inside the **<xmlmesh>** element:

```
<soap>
  ...
  <xmlmesh>
    ...
    <notify end = “yes”/>
    ...
  </xmlmesh>
  ...
</soap>
```


Message bindings

The SOAP controller provides a number of message handlers which perform various functions. 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. The currently available message handlers are just to provide backwards compatibility with the 2.x messaging interface. In future versions a new set of message handlers will run in parallel with this:

Handler name	Message type
<legacy-stream-start>	**<ps:start-stream>**
<legacy-stream-stop>	**<ps:stop-stream>**
<legacy-stream-play>	**<ps:play-stream>**
<legacy-stream-get-position>	**<ps:get-stream-position>**
<legacy-master-status>	**<ps:stream-master-status-request>**

For more information on each of these messages, please see the Application Note "AN-SL-601 XML Messaging". However, if you are considering implementing them in new projects we would recommend checking with Packet Ship support as to the status of the new message set.

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

Note that in the standard configuration none of the message handlers are bound to HTTP URLs, because the 2.x server did not support HTTP 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. For example:

```
<soap>
  ...
  <messages>
  ...
    <legacy-master-status>
      <xmlmesh subject="packetship.stream.master.status.request"/>
    </legacy-master-status>
  ...
  </messages>
  ...
</soap>
```

The standard configuration has XMLMesh bindings for the handlers that match those used in the 2.x server.

Permanent streams

The permanent stream controller (element **<permanent>**) creates and manages permanently configured streams such as broadcasts, background loops and so on. The controller obtains information about the permanent streams from XML configuration files which it can scan specified directories for.

Stream configuration files

Permanent streams are configured in stream configuration files, which are XML files with a **<streams>** root element. **<stream>** elements within this element define individual streams. There can be any number of configuration files and their contents are merged together.

Each **<stream>** element has a **content** attribute giving the content ID to stream – this can be an individual asset or a playlist. Looping playlists are commonly used – see the Directory section for more details. Each **<stream>** element also has an **output** attribute which gives the ID of the output to use – see the Outputs section for more details.

Streams can be given a unique ID with an **id** attribute – if not specified, the ID of the content is used. Two streams cannot exist with the same ID, so if the same content is being streamed in two different ways, an explicit ID on one or both of them will be required.

Within the **<stream>** element are one or more **<destination>** elements with **address** and **port** attributes giving the address and port to stream to. Commonly the address will be a multicast address such as "225.1.1.1", but they can also be unicast – for example for streaming to an encoder device.

The following example shows a simple stream configuration file with a single multicast loop transmitting playlist "test-loop" over raw UDP to 225.1.1.1:11111:

```
<streams>

  <stream content="test-loop" output="udp">
    <destination address="225.1.1.1" port="11111"/>
  </stream>

</streams>
```


Stream configuration search

To make it easy to configure streams, the permanent stream controller can search one or more specified directories for files matching a certain pattern, and try to load them as stream configuration files. Each search directory and pattern is specified in a **<search>** element inside the **<permanent>** element, with **directory** and **pattern** attributes:

```
<permanent>
  <search directory="/etc/packetship/streams/" pattern="*.cfg.xml"/>
</permanent>
```

With the standard configuration shown above, it is possible to drop any file ending with "`.cfg.xml`" into /etc/packetship/streams/ and reload the server to start the stream.

Reloading streams

When the server is reloaded (see Server Management above), permanent streams which have not changed are left running. The server detects changes by comparing the entire **<stream>** element XML for the existing stream with the new version – if anything substantive has changed (e.g. content and destinations) the stream is restarted. Comments and whitespace changes are ignored.

If a given stream ID is no longer configured at all, it is stopped.

Directory services

The directory service provides a way of converting the abstract "asset ID" quoted by the Controller (for example, from the path of an RTSP or HTTP URL) into a real filename – or a playlist of filenames – on disk, and obtaining associated rate information.

The directory service is structured as a set of Directory Provider modules, which are checked in turn until one of them is able to resolve the asset ID requested. These are configured as elements inside a **<directory>** element at the top level of `streamd.cfg.xml`.

Index suffix

Most advanced features of the server such as variable bit-rate (VBR) rate pacing, trick mode fast-forward and rewind and HTTP Live Streaming require an 'index' file. This is an overview of the main file, giving information about the location of access points (I-frames) and timing data. The index files are created with the `ps-index-mpeg2ts` utility.

By default the indexes are named after the asset they refer to, with a ".psi" suffix. If it should be necessary to change this, it can be set in the **suffix** attribute of an **<index>** element inside the **<directory>** element:

```
<directory>
  <index suffix=".psi"/>
</directory>
```

Direct mapping

The simplest method of converting asset IDs into filenames is to actually treat them as filenames within a given directory. The 'direct' Directory Provider (element **<direct>**) allows this.

The direct Directory Provider contains a number of mappings from an asset ID prefix to a directory. The remainder of the asset ID is then combined with the directory to form a full path name for the file.

Each mapping is created with a **<map>** element inside **<direct>** with a **prefix** attribute giving the asset ID prefix (with no leading slash) and a **to** attribute giving the directory (including a trailing slash). If the **prefix** is absent or empty, all assets are matched. This is convenient for simple systems but also precludes using any other Directory Providers. Hence the standard configuration maps `media/` to `/usr/local/share/media/`, so that asset IDs such as "media/mymovie.ts" can be used:

```
  <direct>
    <map prefix="media/" to="/usr/local/share/media/"/>
  </direct>
```

Multiple **<map>** elements can be included, and the first one to match will be used. This is equivalent to (but more efficient than) having multiple **<direct>** Directory Providers each with one **<map>**.

Note that index files (.psi) must be used with direct mapping because there is no other way to determine the file rate for admission control and rate pacing.

Configuration files

The traditional way to configure assets in previous versions of the Packet Ship Streamline video server was to include them in XML configuration files (directory files). This method still works and the files used in previous versions will still be recognised. However with direct mapping and discovery of rate from the index file the need for them is less than before for simple assets. However they are still needed in order to define playlists.

The 'config-files' Directory Provider (element **<config-files>**) looks up the Controller's asset ID in one or more XML configuration files, from which it may find an individual asset, or a playlist. It searches for XML files matching one or more filename patterns, each in a particular directory. Each search pattern is defined in a **<search>** element inside **<config-files>**, with a **directory** attribute giving the directory path to search in.and a **pattern** attribute giving the filename pattern ('glob' form, using '*' wildcards).

Here is the standard configuration:

```
<config-files>
  <search directory="/usr/local/share/media/" pattern="*.xml"/>
</config-files>
```

This finds all files ending with ".xml" in the standard media location, `/usr/local/share/media/`

Directory files

A directory file is an XML file with root element **<directory>**. It can contain lists of assets and playlists in **<assets>** and **<playlists>** top-level elements as above:

```
<directory>
  <assets>
     ... assets ...
  </assets>

  <playlists>
     ... playlists ...
  </playlists>
<directory>
```

Asset definitions

An asset is defined in an **<asset>** element an the **<assets>** group. At minimum it has an **id** attribute giving the asset ID and a **file** attribute giving the file path. The file path can be (and usually is) relative, and if so is resolved relative to the directory that the directory file is in. For example:

```
<directory>
  <assets>
    <asset id="foo" file="foo.ts"/>
    <asset id="bar" file="bar.ts"/>
  </assets>
</directory>
```

Advanced <asset> configuration

The **<asset>** tag can contain further optional attributes as follows:

- The **rate** attribute gives the rate in bits/second, as usually quoted for media streams. This is only required if you want Constant Bit Rate (CBR) streaming and have no index file. If an index file is used (recommended in the majority of cases), this attribute serves no purpose.
- The optional **start** attribute gives a start offset expressed as a time (NPT) – either in units (e.g. "1 hour"), seconds ("3600") or in HH:MM:SS form (e.g. "01:00:00").
- The optional **end** attribute gives an end offset expressed in a time, as with **start**.

If the client requests a sub-range of the asset (e.g. uses an RTSP Range: header) this is additional to the **start** and **end** attributes.

If you provide an index file (.psi) for the asset, the time-based start and end points will be calculated exactly to the nearest 'access point' frame. If not, the average rate will be used to estimate an offset in the file, which may be in the middle of a frame, and the client will have to deal with the partial frame data.

Hence a full **<asset>** element is as follows:

```
    <asset id="foo" file="foo.ts"
         rate="4526000"
         start="01:00:00" end="01:30:00"/>
```

Creating playlists

Sometimes you may want to construct a single 'virtual' asset out of a number of real ones, or to create a repeating loop of content for a permanent stream. To do this, you use 'playlists'.

Playlists are created with **<playlist>** elements in the top-level **<playlists>** element. Each **<playlist>** has the following attributes:

- The **id** attribute gives an ID that can be quoted in RTSP, just like a 'real' asset.
- The optional **loop** attribute sets whether the playlist loops forever, or ends naturally – as with all Boolean attributes, "yes" or "true" to set it, "no" or "false" to clear it. Not looping is the default.

Each item in the playlist is added with a **<content>** element, which has contains the ID of the child item as text content (not as an attribute!). This id can either be a real asset ID, or the ID of another playlist (which must already be defined) – so you can create playlists of playlists.

So, assuming the assets configured as above, you can add the following directory file:

```
<directory>
  <playlists>
    <playlist id="foobar-loop" loop="yes">
      <content>foo</content>
      <content>bar</content>
    </playlist>
  </playlists>
<directory>
```

This creates a playlist containing both 'foo' and 'bar', which loops for as long as the server keeps it running. The client (or permanent Controller) requests it just like any other asset.

Numbered chunks

With some legacy content delivery systems it may be necessary to split large assets into multiple smaller chunks, and there may be operational reasons not to want to concatenate them back together again. Such files usually have a numbered suffix, and it is convenient to be able to specify a numeric range in a playlist in order to make them appear as a single large file again.

To do this, the <content> element in a <playlist> can have a numeric range in the form "`{m-n}`" added anywhere in the asset ID, and it will automatically expand this into the numeric series of asset IDs required. These can either be assets configured in files, or directly mapped filenames.

For example:

```
<directory>
  <playlists>
    <playlist id="foo">
      <content>foo.{1,4}.ts</content>
    </playlist>
  </playlists>
<directory>
```

This creates a 'virtual' asset "foo" which comprises mapped files `foo.1.ts`, `foo.2.ts`, `foo.3.ts` and `foo.4.ts`.

Note that in previous versions of the server, this expansion was done at the filesystem level, because assets in a playlist were not treated as a continuous flow. From 3.x (Antigua) onwards, buffering and rate-pacing are continuous over playlist boundaries, and hence the standard playlist mechanism can be used instead. The old 'virtual file' based system is no longer supported. Another implication of this is that index files should now be created individually for the chunks rather than (as before) from the original un-chunked asset.

Gridline cache lookup

In integrated solutions using the Packet Ship Gridline product for content management and/or distribution, the streaming server will want direct access to the files in the Gridline 'cache' rather than having to copy and configure them in its own directories.

The cache Directory Provider (element **`<cache>`**) looks up an asset ID (actually the bundle path of an asset file) in the cache through an XMLMesh message to `ps-cached` (or indeed any other content management service that implements the same request). This will return the location on disk of the file and its average rate if known. If not known, the cache Directory Provider will calculate it from an associated index file (if any).

There are no configurable properties of the cache Directory Provider – it is created simply with an empty **`<cache/>`** element:

```
<directory>
  ...
  <cache/>
  ...
</directory>
```

The cache Directory Provider requires that an XMLMesh connection is configured in the **`<soap>`** controller.

Timeline capture lookup

The Packet Ship Timeline product provides real-time capture of multicast IPTV channels and EPG data. To access the captured streams from the Streamline streaming server, the capture Directory Provider (element **<capture>**) looks up an asset ID representing a TV channel and optional start time through an XMLMesh message to `ps-captured` (or another capture system with the same interface). This returns the location on disk of the capture file for that channel, and (if requested) the time offset in that file of the given real time.

Asset prefix

To allow the capture Directory Provider to co-exist with other Directory Providers, an optional asset prefix can be specified in a **prefix** attribute of an **<asset>** element in **<capture>**:

```
<capture>
  <asset prefix = "tv/"/>
</capture>
```

Capture asset IDs

The asset IDs used to access captured content come in two forms. Firstly, you can just use the channel name (with optional prefix) to access the current position in a given channel – for example:

```
rtsp://server/tv/BBC1
```

To access a particular point in the captured content, add an ISO timestamp as well:

```
rtsp://server/tv/BBC1/20110115113000
```


Admission control

Admission control imposes limits on access to streams either through authorisation of individual clients or through capacity limits on the entire server, or both. Setting up safe admission control limits is one of the key aspects of creating a secure, reliable streaming service.

Admission of streams is controlled by a series of Admissions Filters, configured in an **<admissions>** element at the top level of `streamd.cfg.xml`. Each Admission Filter has an element within this with the type of the filter. Filters are tried in order, and if *any* fail then the stream is blocked – i.e. a veto system.

Multiple Admission Filters of a given type can be used – if so, it is usual to give them an **id** attribute so they can be distinguished in logging.

Access control

The 'access' Admissions Filter (element **<access>**) provides a generic Access Control List (ACL) mechanism for authorisation of streams based on client IP address.

The ACL system is a standard one across Packet Ship products, and is based on access to *resources* by *users*, optionally modified by IP address. In this case the resource is an asset ID, and the user is the controller name ('rtsp', 'http' etc.)

Each request for a stream is matched against successive resource (asset ID) patterns until one matches the resource name. Within that resource are one or more rules which specify whether particular address or controller name patterns should be allowed or denied. If two rules match the same address or name, denial takes precedence over allowance, whatever the order of the rules. If either no rules match within the first matching resource, or no resources match at all, the request fails safe and is denied.

Resource patterns

Resource (asset ID) patterns are created as **<resource>** elements within an overall **<resources>** element, each with a **name** attribute giving a 'glob' pattern to apply to the asset ID.

For example:

```
<admission>
  ...
  <access>
     <resources>
       <resource name="movies/*">
          ... rules for movies ...
       </resource>

       <resource name="*">
          ... rules for everything else ...
       <resource>
    <resources>
  </access>
  ...
</admission>
```


Access rules

Within each **<resource>** resource pattern, you can put **<allow>** and **<deny>** rules which define the IP addresses and/or controller names to allow or deny. In both cases, an **address** attribute matches an individual network address or address mask (in CIDR format – e.g. "192.168.0.0/24"), and a **user** attribute matches a particular controller name.

Both attributes are optional, and match all addresses and all controllers if left out. If both are specified, both must match. A rule with neither **address** nor **user** attributes therefore matches every client. Hence **<allow/>** is a useful fallback to allow any client except ones that are specifically denied. **<deny/>** denies all clients, but it's redundant since an empty **<resource>** element denies all clients anyway.

For example; to only allow only local network access to RTSP, for all assets:

```
<resource name="*">
   <allow address="192.168.0.0/24" user="rtsp"/>
</resource>
```

Deny rules are more often used as exceptions to a general allowance. For example, if we wish to block a particular network or user from something we otherwise give access to:

```
<resource name="movies/*">
  <deny address="192.168.0.0/24"/>
  <allow/>
</resource>
```

This would block any users from the 192.168.0.0 network but allow anyone else. Note that although we show the deny rules first for clarity here, within a resource element the deny rules *always* act before the allow rules, whatever the order they are given.

Standard configuration

The standard configuration is as follows:

```
<access>
  <resources>

    <!-- This matches all assets -->
    <resource name="*">

      <!-- Allow private network access to all controllers -->
      <allow address="127.0.0.1"/>
      <allow address="10.0.0.0/8"/>
      <allow address="172.16.0.0/12"/>
      <allow address="192.168.0.0/16"/>

      <!-- Allow multicast access to permanent only -->
      <allow user="permanent" address="224.0.0.0/4"/>

    </resource>

  </resources>
</access>
```


This allows access for any Controller (protocol) to localhost and private networks, and allows the permanent Controller to output multicasts. All external requests are blocked by default. To open it to the world for (say) HLS only, you would need to add:

```
...
<allow name="hls"/>
...
```

External verification

Sometimes the admission control that the Streamline server can do itself is not enough, and it needs to be integrated with some external mechanism – for example, a billing system providing authorisation of assets through payment records.

The 'external' Admissions Filter (element 'external') uses a 'packetship.stream.access.verify' XMLMesh message to request verification of access from an external process. This contains the asset ID, client address and port, output type and controller name. See the Application Note AN-SL-601 "XML Messaging" for more details.

The external Admissions Filter has no configuration and is simply enabled by including an empty **<external>** element in **<admission>**:

```
...
<external/>
...
```

This is disabled by default in the standard configuration, otherwise all streams would fail.

Per client limits

In local area networks it is useful to be able to limit any individual device to receiving a maximum number of streams – often only one, because many embedded devices (e.g. set-top boxes) can only handle one in any case, and requesting more than one is usually an error.

Embedded devices also have the property that they are prone to disappear while in the middle of a stream and reboot, often very quickly. In this case the normal keepalive timeout may not have time to fire, and if the device requests a new stream soon after booting it will receive two streams. Hence an option is needed to kill off ('sacrifice') the old stream before a new one is allowed.

The 'per-IP' Admission Filter (element **<per-ip>**) implements a maximum number of streams per client IP address and stream sacrificing as described above. The maximum number of streams is set in a **streams** attribute, and sacrificing is turned on and off with a **sacrifice** attribute:

```
...
<per-ip streams="1" sacrifice="yes"/>
...
```

Public Internet

Note that in the public Internet (or any other routed network), you cannot assume that a single IP address is a single client because of NAT routing. In most cases this is only a single IP per

household, but some ISPs NAT their entire user base! Hence in such networks the **<per-ip>** Admission Filter should not be used.

Performance testing

A per-client limit would also prevent performance testing from a machine emulating multiple clients. Hence it is disabled in the standard configuration.

Capacity Limits

Capacity limits are concerned with the total capacity of the server, or the networks it is connected to. The capacity Admission Filter (element **<capacity>**) keeps a running total of the total number of streams and total bandwidth being used, and blocks streams above a configured limit.

A capacity Admission Filter has a **streams** attribute giving the maximum number of streams and a **bandwidth** attribute giving the maximum bandwidth. If either is unspecified, it is unlimited. The bandwidth is specified in Mbits/sec (decimal millions of bits per second) in terms of data read off disk. Depending on the streaming format the network bandwidth may be more than this, and this should be taken into account if network bandwidth is the limiting factor.

By default the capacity limits apply to all networks and all assets, but these can be restricted by a **route** attribute, taking a CIDR route specification (x.x.x.x/y), and an **asset** attribute, taking a 'glob' pattern on asset ID, respectively. This allows you to enforce limits on particular networks, client groups or types of asset.

Here is a typical server-wide capacity limit:

```
    <capacity id="total-capacity" streams="500" bandwidth="3000"/>
```

Note that the ID has been set to make logging clearer. The following is a specific capacity limit for "test" assets on a local network:

```
    <capacity id="test-streams" asset="test*" route="10.0.0.0/8"
              streams="10"/>
```

The precautionary principle

It is crucial for the quality of any video service that any stream that is admitted continues to run smoothly for its duration, and is not adversely affected by later streams being added. This is the reason for capacity admission control – it is far better to refuse to serve a new stream at all than to risk all the streams being served being delivered badly.

The absolute maximum capacity of any given hardware – disk, processor and network - can only be determined by testing under full load. However good engineering will reduce the configured maximum capacity significantly below this – often to only 50% - to ensure perfect performance in all conditions.

Important Note

The Packet Ship Streamline video server is totally dependent on the performance of the underlying hardware – particularly disks – for its performance, and Packet Ship can make no guarantees or recommendations about what performance is available without knowing the specifics of the hardware and what else is running on it. It is an explicit condition of licensing

of the Packet Ship server that the system integrator test the performance on the target hardware under maximum load and in unusual conditions before release. We recommend that a significant engineering tolerance be applied to the configured limits to ensure satisfactory operation in all cases.

Demonstration version limits

In the demonstration version which can run without a purchased licence, a capacity Admissions Filter is added which limits the server as a whole to 5 streams or 25 Mbit/sec of bandwidth.

Licence limits

Purchased licences for the Streamline server contain an **`<admissions>`** element which is used in addition to any that are specific in the configuration file, and which will contain at least a capacity Admissions Filter limiting the server to the licensed bandwidth.

Inputs

Inputs represent a source of data for streaming, and are configured in an **<inputs>** element at the top level of `streamd.cfg.xml`. In the current version there is only one type of input, file, but others are possible in the future.

File input

The file input (element **<file>**) represents data sourced from a disk file and any associated index file.

O_DIRECT reading

Modern Linux kernels have the facility for real-time applications like `ps-streamd` to bypass the normal Linux disk cache and read directly from disk into memory buffers. This both speeds things up and also stops the huge amount of streaming data 'polluting' the rest of the disk cache. This is known as O_DIRECT after the C flag used to enable it.

O_DIRECT reading is enabled with the **read** attribute of a **<direct>** element inside **<file>**:

```
<file>
  ...
  <direct read="yes"/>
  ...
</file>
```

This is enabled by default, and it should only be necessary to disable it if for some reason your Linux kernel does not support O_DIRECT. This is not recommended and you would be better off upgrading your kernel!

Offset rounding

The Streamline video server is primarily designed around MPEG-2 Transport Streams which have a fixed 188-byte packet structure, and many client devices will not accept streams unless they are aligned to packet boundaries. The use of index files should ensure that any seeks into a file hit the correct boundary in any case, but just in case, the server can be configured to round all file offsets to a given multiple – usually 188 bytes.

Offset rounding is configured in a **rounding** attribute of an **<offset>** element inside **<file>**:

```
<file>
  ...
  <offset rounding="188"/>
  ...
</file>
```

It should only be necessary to disable this if streaming some other format of file (in which case only normal play at constant bit rate will be possible).

Index files

The ability of a video server to deliver a visual stream while fast-forwarding or rewinding is known as 'trick mode'. Obviously you can't just play the stream faster – the load on the server and network

would be too high, and few clients can support an increased frame-rate. The solution is to only send selected frames – I-frames in MPEG-2, SI frames in H.264 – which typically represent one frame in 12 or 15, and can be sent independently of any other frames.

However, it would also be too great a load on the server to have to parse each MPEG stream and locate the I-frames for large numbers of streams. Hence this process is done in advance, to create an 'index file' (.psi file), which is a map of the original stream which allows location of I-frames and fast access to the right place in the index. The same map is also used to provide more accurate seeking based on Normal Play Time (NPT), and alignment of the stream to I-frame boundaries at various transitions. It also gives the server information required to handle VBR streaming, and chunking in HLS. These files are created with the `ps-index-mpeg2ts` indexing tool.

The suffix used to find the index file is configured in the **`<directory>`** section – see above.

An **`<index>`** element inside **`<file>`** provides a number of configuration options for index file reading and use:

Index buffer size

The buffer size for reading index files – and hence the amount of data read ahead at one go during streaming – is configured in a **`size`** attribute of a **`<buffer>`** element inside **`<index>`**. The default is 65536 bytes.

Video filtering

Trick mode is performed by selecting sections of the original file which represent I-Frames (independent frames), using the information in the index file to locate them. Without any other processing this would also include any audio interleaved with the video data, and some clients will attempt to play this, with unpleasant results. This can be filtered out by setting the video attribute of a **`<filter>`** element inside **`<index>`**. This has a small performance impact so if you know that your clients suppress audio during trick play, this can be disabled.

Timing adjustment

Some clients – notably VLC and derivatives – require the trick mode stream to appear as if it were a normal stream in terms of timing. We recommend that clients turn off timing control during trickplay and simply display frames as they arrive, but to support clients that don't do this, the server can adjust the timing values of the stream to make it continuous. This is enabled with an **`adjust`** attribute of a **`<timing>`** element inside **`<index>`**.

Note that this somewhat duplicates the effect of the **`<ts-timing-adjust>`** Output Filter described later in this guide. However **`<ts-timing-adjust>`** applies to all stream output, not just trick mode, and hence is a much more CPU-intensive process. So it is best to only use that for situations that demand it, such as looping playlists – in which case trick mode is probably not going to be used in any case, so there is no harm in leaving the index timing adjustment enabled as well.

The standard file input index configuration is as follows:

```
<file>
  <index>
    <buffer size="65536"/>
    <filter video="yes"/>
    <adjust timing="yes"/>
  </index>
</file>
```


Outputs

Outputs are the 'business end' of the streaming server, where data is actually delivered to the client. Multiple types of Output can be configured, and the stream's Controller specifies which one it requires.

Outputs are configured in an **<output>** element at the top level of `streamd.cfg.xml`. Within this, individual Outputs are created with elements giving the type of the output, and in most cases an **id** attribute which is used to select it. If no **id** attribute is given, the Output's ID is simply the type itself (e.g. 'udp').

Common output configuration

All Outputs share the following common configuration properties. In the examples below **<udp>** is used as the example output.

Streaming group

Streaming groups configure common streaming parameters such as thread usage, buffer sizes and packet bursting – these are described in the "Stream Groups" section below. However the stream group that a stream is placed in is controlled by the Output it uses (these stream parameters are often tuned differently for different protocols).

The stream group to use is configured in an id attribute of a <group> element inside the Output's element:

```
<udp>
...
  <group id="realtime"/>
...
</udp>
```

Packet size

The size of packets to output is configured with the **size** attribute of a **<packet>** element in the Output's element:

```
<udp>
...
  <packet size="1316"/>
...
</udp>
```

For UDP outputs, this is simply the datagram size (excluding any data added by Output Filters). The default of 1316 is a safe size, since it represents a round number (7) of Transport Stream packets (as required by RTP and many raw UDP implementations), and fits inside a single Ethernet frame, even with RTP headers. This means it does not have to be fragmented, and some devices cannot handle fragmented datagrams.

If you know that your target devices can handle fragmentation, there is both a server-side (fewer system calls) and network-side (fewer UDP/RTP headers) advantage to increasing this. A value of

8084 is a good choice, being the maximum number (43) of Transport Packets that will fit within an 8K datagram.

For TCP outputs, this is the data size sent to the kernel at one go, and if chunked encoding is used, the chunk size. In both cases the default of 65536 is reasonable. There is no requirement to make it a round number of TS packets.

Source address

If the server has multiple interfaces it is sometimes useful to be able to pin the Output to a particular source address and hence a particular interface. The source address can be configured in an address attribute of a <source> element inside the Output's element:

```
<udp>
...
  <source address="10.0.0.1"/>
...
</udp>
```

If absent, or set to "0.0.0.0", the choice of source address (interface) is left to the kernel through normal routing configuration.

Output Filters

Output Filters modify the output in some way, for example implementing wrapper protocols such as RTP, or fixing up stream timing data for looping playlists. The Output Filters available are discussed in a later section, but just to note at this point that they are configured as elements with the Output Filter's name in a <filters> element in the Output's element – for example:

```
<udp id="udp-rtp">
...
  <filters>
    <rtp>
      ... RTP configuration ...
    </rtp>
  </filters>
...
</udp>
```

Note that this instance of the **<udp>** Output has an **id** set to distinguish it from the raw UDP one, with no filters.

Rate Profiles

Similarly, Rate Profiles are discussed in their own section later, but again these are configured in a **<rate-profiles>** element inside the Output's element – for example:

```
<udp>
...
  <rate-profiles>
    <limit>
      ... rate limit configuration ...
    </limit>
  </rate-profiles>
...
</udp>
```


UDP/IP

The UDP/IP Output (element **<udp>**) generates UDP datagrams on a datagram socket which is independent of whatever (if any) connection is used by the Controller. The UDP/IP Output can be used with **<rtsp>**, **<soap>** and **<permanent>** Controllers. In the case of RTSP it is configured as part of the Transport header mapping; with **<soap>** it is specified by the stream start message, and with **<permanent>** it is specified as the **output** attribute of the **<stream>** element in the **<streams>** configuration file.

The specific configuration parameters for the UDP/IP Output are as follows:

IP Type of Service (TOS)

If you are using Differentiated Services (DiffServ) in your network, or have policy routing enabled on the server, you may want to set the Type of Service (TOS) bits on the outgoing IP packets. The Linux kernel also uses the precedence bits of the TOS field to prioritise packets within the kernel itself. The entire IP TOS field can be set with the **tos** attribute of an **<ip>** element inside the **<udp>** element:

```
<udp>
  ...
  <ip tos="136"/>
  ...
</udp>
```

The value given is passed directly to the IP_TOS `setsockopt()` option – see Linux manuals for details.

Note that the value 136 is hex 0x88, binary 10001000, and includes the two bottom ECN bits (which are not under application control and will be masked off anyway). Hence this represents DiffServ code point 100010 (DSCP value 34), which is the highest "Assured Forwarding" class with minimum drop probability, AF41.

To get the highest "Expedited Forwarding" code point, '101110' (DSCP value 46) you would use 184 as the TOS value. However, in order to do this Linux requires that you are running as a privileged user when the socket is created, so you would need to disable the switch to non-privileged user/group option in **<security>** to do so (see the section on common configuration above).

IP Time To Live (TTL)

If you are streaming over a routed network (multiple subnets) you may need to, or wish to, set the Time To Live (TTL) value of the outgoing packets in order to extend, or limit, the normal propagation distance in the network. By default in Linux, unicast packets have a TTL of 64 (and hence can propagate almost anywhere) and multicast have a TTL of just 1 (and hence cannot propagate beyond the local subnet).

The TTL values for each type of traffic can be set with **ttl** (unicast) and **multicast-ttl** attributes of the **<ip>** elements already described – e.g.:

```
...
<ip tos="136" ttl="3" multicast-ttl="2"/>
...
```


This sets the unicast TTL to (an unusually low) 3, and multicast TTL to 2, allowing propagation to one level of routed network.

It is unlikely you would want to change the unicast TTL except to provide some limited security, but it is quite common to need to extend the multicast TTL if you require multicast beyond the local subnet. If you have a routed network and unicast 'broadcasts' are working but multicasts are not, this is usually your first port of call. Beware of setting multicast TTL to unnecessarily large values, though, because you may cause leakage onto non-multicast switches which will then trigger a broadcast storm.

Joining own multicast

Some Ethernet switches will revert to flooding multicast packets to every interface (like a broadcast) unless at least one interface has joined to the multicast. To avoid this, the UDP Output can be configured to join to its own multicast outputs with **<join multicast="yes"/>**:

```
<udp>
  ...
  <join multicast="yes"/>
  ...
</udp>
```

TCP/IP

The TCP/IP Output (element **<tcp>**) generates TCP stream data on the same connection that was used for the client to connect to stream's Controller, interleaved with any command data or responses. The TCP/IP Output can be used with **<rtsp>**, **<http>** and **<hls>** Controllers. In the case of RTSP it is configured as part of the Transport header mapping; with **<http>** and **<hls>** it is specified as the **output** attribute of the **<stream>** element in the Controller configuration.

There are no specific configuration parameters for the TCP/IP Output.

Output Filters

Output Filters add extra protocol layers or make other changes to the output data. They are configured within a <filters> element inside the Output element they attach to, as described above.

RTP

The RTP Output Filter (element **<rtp>**) wraps the output data in an RTP header, according to RFC 3550. The SSRC is randomised for each stream, the sequence number is incremented for each packet and the timestamp comes from the server's clock at the time of sending.

RTP payload type

The payload type can be set with the **type** attribute of a **<payload>** element inside **<rtp>**.

```
<rtp-udp>
  <payload type="33"/>
</rtp-udp>
```

The default payload type of 33 is assigned for MPEG-2 Transport Streams (RFC 1890).

RTSP interleave

The RTSP Interleave Output Filter (element **<rtsp-interleave>**) wraps the output data in an RTSP TCP Interleave header ('$' format) which allows the binary data to be interleaved with command data on the single RTSP connection. This is required for RTSP-TCP connections.

This filter must appear last in the chain of Output Filters. It does not have any configuration properties.

HTTP chunked

The HTTP Chunked Output Filter (element **<http-chunked>**) wraps the output data in an HTTP/1.1 chunk header, data to be progressively downloaded and cleanly marks the end of a stream without needing to drop the connection. This is required for all HTTP connections, both progessive download (**<http>**) and HTTP Live Streaming (**<hls>**).

This filter must appear last in the chain of Output Filters. It does not have any configuration properties.

Transport Stream timing adjustment

The Transport Stream timing adjustment Output Filter (element **<ts-timing-adjust>**) fixes up time discontinuities in the output stream (MPEG-2 Transport Streams only) in order to provide a smooth timebase for the decoder. This is sometimes needed when using playlists – particularly

looping playlists – in order to achieve a seamless transition between assets (or on loop of an asset) in some decoders.

There is a fairly high processor overhead from doing this, and hence we recommend you create a special Output with this filter attached, and use it only for streams (e.g. permanent streams) where it is required.

The filter has the following configuration properties:

Enabling in normal play / trick mode

Depending on the application the adjustment may not be required all the time. In particular, the file Input has the capability to adjust timing of its trick mode output itself, making applying the filter again a waste of processor time. However, the adjustment of the trick mode output alone cannot create a seamless join between trick mode and normal play, which some devices prefer.

Hence it is possible to configure whether the filter operates in trick mode, normal play or both. To enable it in trick mode, set the **`mode`** attribute of a **`<trick>`** element; for normal play the **`play`** attribute of a **`<normal>`** element, both within **`<ts-timing-adjust>`**:

```
<ts-timing-adjust>
  <trick mode="no"/>
  <normal play="yes"/>
</ts-timing-adjust>
```

Maximum discontinuity

In order to create a smooth timebase, the filter spots discontinuities in the natural timebase of the stream, and adds or subtracts an adjustment based on the real time between packets to correct for them. The rest of the time, it simply maintains the (possibly offset) timebase from the original stream.

The maximum discontinuity (in seconds) to allow before triggering a re-basing is set by the **`delta`** attribute of a **`<discontinuity>`** element inside **`<ts-timing-adjust>`**:

```
<ts-timing-adjust>
  <discontinuity delta = "1.0"/>
</ts-timing-adjust>
```

The default is 1 second. Rewinds in the timebase are *always* considered a discontinuity.

Transport Stream continuity counter Adjustment

The Transport Stream continuity counter Adjustment Output Filter (element **`<ts-cc-adjust>`**) fixes up MPEG-2 Transport Stream continuity counters in the output stream in order to prevent the receiving device thinking there has been packet loss. Like timing adjustment, this is sometimes needed when using playlists – particularly looping playlists, or during trick mode output.

Enabling in normal play / trick mode

This filter can be enabled in either trick mode or normal play, or both, in the same way as the Timing Adjustment one:

```
<ts-cc-adjust>
  <trick mode="no"/>
  <normal play="yes"/>
</ts-cc-adjust>
```

Rate Profiles

As a streaming server, the Packet Ship Streamline video server keeps tight control of the rate it sends data out, to avoid either overflowing (sending too much)or underflowing (not sending enough) the network capacity and the buffers on the receiving device.

Usually the server obtains the rate to send the data at from the index file, which marks the position of specific times within the file; the server then aims to send that much data in the given time, giving Variable Bit Rate (VBR) output. Alternatively, in rare cases it may be enough to send the data at a fixed Constant Bit Rate (CBR), but this is not recommended (the quality is worse for the same average bandwidth).

On top of the 'natural' rate for the asset, you can apply Rate Profiles which modify it in some way. These are configured in the **<rate-profiles>** element of an Output element, as described above.

Fixed rate

The fixed Rate Profile (element **<fixed>**) simply ignores the natural rate altogether, and tries to send at a fixed rate determined by its bandwidth attribute (specified in decimal Mbit/sec). This would be an unusual thing to do for UDP traffic, but it makes sense for TCP streams where the client is largely in control of the amount of data sent, and can usually be expected to demand it no faster than it can make use of it.

Having said that, it is a good idea to limit the rate in some way, to avoid clients pulling data at ridiculous rates, either in error or maliciously. By setting a **<fixed>** Rate Profile on the **<tcp>** output, the streaming rate will be limited either by the client or your fixed rate, whichever is the lower:

```
<rate-profiles>
  <fixed bandwidth="10"/>
</rate-profiles>
```

Rate multiple

The multiple Rate Profile (element **<multiple>**) applies a multiplier (which can be less than 1) to the natural rate of the stream, optionally for a limited time at the start of the stream. This is useful to quickly fill the client's buffers and hence get the stream playing more quickly.

Applying a multiplier is also a somewhat safer way to limit TCP stream rates; allowing the client to pull down more than it needs but only up to a certain point, and optionally only for a certain time.

The rate multiplier to apply is specified by a floating point **multiple** attribute of a **<rate>** element inside **<multiple>**. The time to allow the multiple for is specified in seconds as the **time** attribute of a **<start>** element. If the time is left out, the multiplier continues to apply for the whole stream. This is not a good idea for UDP streams, since it will almost certainly cause a buffer overflow in the client eventually.

The following applies a 20% rate increase for 5 seconds at the start of the stream:

```
<multiple>
  <rate multiple = "1.2"/>
  <start time="5"/>
</multiple>
```

Rate limits

The limit Rate Profile (element **<limit>**) applies a minimum and maximum rate to the stream, either as a fixed bandwidth or as a proportion of the natural average rate of the asset. This is particularly useful for Variable Bit Rate (VBR) streams in order to "peak-lop" (or "trough-fill") their rate of output, to avoid hitting network or device limits.

Fixed bandwidth limits

Minimum and maximum rate limits in terms of fixed bandwidths are specified as bandwidth attributes of **<minimum>** and **<maximum>** elements (respectively) inside the **<limit>** element. The bandwidths are specified as decimal Mbit/sec (millions of bits per second). If either are unspecified, the limit does not apply.

The following simply applies a maximum bandwidth of 20Mbit/sec to all streams:

```
<limit>
  <maximum bandwidth="20"/>
</limit>
```

Proportional bandwidth limits

Rather than a fixed number of Mbit/sec, it may be more useful to specify the limits as a proportion of the average rate of the asset. These can be specified as **proportion** attributes of the same **<minimum>** and **<maximum>** elements, and are expressed as a floating point multiplier (which may be less than 1).

The following applies a wide "order-of-magnitude" limit on the rate:

```
<limit>
  <minimum proportion="0.1"/>
  <maximum proportion="10"/>
</limit>
```

Combined limits

The two forms of limit can be combined, with the strictest limit applying. Hence the following applies the rule "not less than 100Kbit/sec, or one tenth of the average rate, whichever is the larger, and not more than 20Mbit/sec, or ten times the average rate, whichever is the smaller":

```
<limit>
  <minimum bandwidth="0.1" proportion="0.1"/>
  <maximum bandwidth="20"  proportion="10"/>
</limit>
```


Stream groups

Each stream in the server is a member of a "stream group", which defines how it is assigned to a processor thread, its buffer sizes and so on. The stream group is chosen by the Output being used, and it is usual to have at least one group for UDP streams and one for TCP, although more can be created if you require different configuration for different types of content or Controller.

Stream groups are configured in a **<groups>** element at the top level of `streamd.cfg.xml`. Each group is created with a **<group>** element within this, with an **id** attribute which defines its ID for association with an Output.

In the standard configuration there are two stream groups defined:

1. 'realtime': Used for UDP streams
2. 'thread-per-stream': Used for TCP streams

```
<groups>
  <group id="realtime">
     ... realtime (UDP) group parameters ...
  </group>

  <group id="thread-per-stream">
     ... thread-per-stream (TCP) group parameters
  </group>
</groups>
```

Threads

The usage of processor threads by the group of streams is configured in a **<threads>** element inside the **<group>** element.

Realtime scheduling

For UDP streams, the timeliness of sending of the packets is crucial, and hence they need to be run as realtime (high priority) threads under the control of the realtime scheduler (see below). For TCP streams, however, the output is naturally burstier (at a millisecond level) and hence realtime scheduling is not required.

Whether the threads are realtime or not is set by the **realtime** attribute of the **<threads>** element.

Thread limits

For UDP streams, there is no advantage in having more threads than processors, because each thread can handle multiple streams. Hence it is usual to fix the number of threads in a realtime group to the number of CPU cores available – this ensures an even split of processor load across cores. For servers which are also performing other functions you may wish to specify the number of threads as less than this, to keep some clear for other processes.

For TCP streams, however, each stream has its own thread, and hence there is no reason to limit them here (limits on the total number of streams is part of admission control).

The maximum number of threads used by the group is set in a **max** attribute of the **<threads>** element. If absent, it is unlimited. Also, the minimum number of threads to keep alive (even if nothing is using them) is configurable in a **min** attribute. For the small number of threads used for a realtime group, there is no reason not to keep them alive all the time, and set **min** = **max**.

Stream limits

The maximum number of streams served by an individual thread can be set in a **streams** attribute. This is usually set to 0 (unlimited) for realtime groups and necessarily set to 1 for TCP groups.

Sleep time

Realtime streams are woken up by the realtime scheduler, but non-realtime streams simply sleep for a while and then come back to check if they need to send any more data. The amount of time (in microseconds) that non-realtime threads sleep for is set in a **time** attribute of a **<sleep>** element inside the **<group>** element.

If the sleep time for a thread is set too low, depending on the kernel version the kernel may not actually sleep the thread at all, and it will just busy-wait, locking up the machine for anything else. The minimum safe sleep time for all kernels is 10 ms (10000 μs).

Standard configuration

The following are the standard thread configurations for the realtime and thread-per-stream groups

```
<group id="realtime">
  <threads realtime="yes" min="4" max="4"/>
  ...
</group>

<group id="thread-per-stream">
  <threads realtime="no" streams="1"/>
  <sleep time=”10000”/>
  ...
</group>
```

Buffer sizes

One of the main limitations on performance of any streaming server is the speed of reading from the disks. For conventional disks the main cost is in seeking between different portions of the disk – Solid State Drives don't have this problem, but there is still some cost in each disk read.

Hence to optimise disk access the server reads the data off disk in large chunks at once into memory buffers. So that data can be sent while disk data is being read, there are always at least two buffers, although there can be more for extra resiliency against other processes using the disk.

The number and size of buffers are set in the **count** and size attributes of a **<buffer>** element in the **<group>** element:

```
  <buffer size=”4194304” count=”2”/>
```


This is the standard configuration for both realtime and thread-per-stream groups, and hence requires 8MB per stream. This is a factor to bear in mind when considering admission control and server memory sizing.

Queue sizes

As the Input reads in the data it is responsible for generating a queue of rate samples which govern the variable bitrate (VBR) output of the stream, and also a queue of access point positions which is used in alignment on transitions. Each of these queues is a fixed maximum length, and needs to be set long enough to cover the maximum number of rate samples or access point positions that could be held in the configured amount of buffering.

A typical calculation would be as follows: If you have 8MB of buffering (the default) of a 4Mbit/sec stream, that represents 16 seconds of streaming. Allowing 50 rate samples and 10 access points per second, that gives queue sizes of 800 and 160 respectively. Allowing a good engineering margin gives the defaults of 1500 and 300.

If you are seeing "Rate queue is full!" or "Access point queue is full!" errors, this is a sign that the length is not set high enough, usually for very low bit rate streams. There is no particular loss other than a small amount of memory for having it too large, so you can be generous! Alternatively, you might be better off reducing the buffer size for such streams.

Queue sizes are configured in a **<queues>** element inside the **<group>** element. Within this are a **<rate>** element and an **<access-point>** element, each with a size attribute giving the respective queue size:

```
<queues>
  <rate size = "1500"/>
  <access-point size = "300"/>
</queues>
```

Burst length

Each time a stream thread is woken up, it asks each stream to output some data according to its rate pacing up to the current time. The stream also indicates whether if it needs to send more than one packet to catch up. If any say they need more output, the server then runs those again, and continues doing this until all say they are up to date.

In the case of a corrupted stream with unreasonably high rate expectations, however, this could go on forever, blocking other threads and processes from running. Hence there is a limit to the number of packets which can be sent in each 'burst' before the stream sleeps again. This is configured in the **length** attribute of a **<burst>** element inside the **<group>** element:

```
    <burst length="16"/>
```

Implication on maximum rates

This parameter imposes a maximum data output rate which depends on the packet size and the scheduling frequency.

UDP Streams

For example, if the realtime scheduler runs 1024 times per second, with a burst of up to 64 packets of 1316 bytes each (the standard configuration for realtime UDP streams), then up to 1024*16*1316*8 = 172Mbit can be sent per second per stream. This is a reasonable limit for current encodings!

TCP Streams

On the other hand, for non-realtime TCP streams, they may only be scheduled every 20ms (50 times a second), but can send up to 65536 bytes each time. Hence a burst length of 16 gives up to 50*16*65536*8 = 419MBit/sec per stream!

The symptom of hitting either limit (which is only likely if some parameters are drastically changed) is lag warnings as described below.

Lag control

'Lag' occurs when for some reason the server is unable to keep up with the rate of flow of data demanded by the stream – for example, because it is overloaded, or the stream's peak rate is just too fast for the server to handle. Careful selection of admission control parameter should prevent this happening, but it is always possible for some external process to 'steal' the resources (CPU, disk bandwidth) that the server needs, and hence it is careful to recover properly if it does happen.

Lag control is configured in a **<lag>** element in the **<group>** element:

```
  ...
   <lag>
     <warning min="8192" interval="1.0"/>
     <recovery max=”131072”/>
   </lag>
…
```

Lag warnings

The first thing the server does if it sees lag in the stream is to warn about it. A small amount of lag (a few KB) is normal and can be ignored, but warnings may be needed above this. It is also important that the warnings themselves don't make the situation any worse!

The lag threshold at which to warn at is specified (in bytes) in the **min** attribute of a **<warning>** element in the **<lag>** element, and the minimum interval for warnings is set in an **interval** attribute, in decimal seconds.

Once the server has warned about a lag it will then suppress further warnings at that threshold, reducing the threshold back to the minimum over a period.

Lag recovery

Normally if the stream lags the server will just send data faster afterwards to catch up. If the lag is very extreme, though, this attempt to catch up may flood the network and client, and make matters worse. Hence above a certain threshold, the server will simply accept the lag and continue at the standard rate. The effect of this will be to cause a stall on the client, but not lose packets through overload, which would be visually worse.

The maximum lag acceptable is configured (in bytes) in a **`max`** attribute of a **`<recovery>`** element in the **`<lag>`** element. The default of 128KB is a reasonable estimate of the amount of buffering that a typical set-top box client may have and hence the limit of lag before a stall will be evidence in any case.

The realtime scheduler

UDP streams which require realtime packet timing are usually placed in a realtime stream group (see above), in which the threads are woken up by a signal from a realtime scheduler. This scheduler runs off the Linux Realtime Clock (RTC) device at a fixed rate of signals per second.

The scheduling frequency to use is set in a **<scheduler>** element at the top level of `streamd.cfg.xml`, containing an **<rtc>** element with a **frequency** attribute giving the frequency in HZ (number of signals per second). The frequency must be a power of two, so typical values are 1024 (just over one per millisecond, the default) or 8192 (8 per millisecond for very tight accuracy).

```
<scheduler>
  <rtc frequency="1024"/>
</scheduler>
```

If /dev/rtc is not readable the scheduler outputs an error at startup, but falls back to non-realtime sleeping at a safe sleep time of 10000.

Installing /dev/rtc

To ensure that the RTC driver is working on your system, you need to make sure the following are in place:

- RTC support available as a kernel module (CONFIG_RTC='m')
- /dev/rtc node in place (major 0, minor 135)
- 'rtc' driver module loaded (e.g. "`modprobe rtc`")
- On some motherboards, you need to turn ACPI off to make the RTC interrupts work properly. The symptom is the RTC appears to work but doesn't generate any interrupts. Append "acpi=off" to your kernel parameters in your GRUB or LILO configuration.

Once you have the driver working, you should have a `/proc/driver/rtc` file available, containing something like the following:

```
# cat /proc/driver/rtc
rtc_time           : 09:52:15
rtc_date           : 2006-08-22
rtc_epoch : 1900
alarm              : 00:00:00
DST_enable         : no
BCD                : yes
24hr               : yes
square_wave        : no
alarm_IRQ : no
update_IRQ         : no
periodic_IRQ       : no
periodic_freq      : 1024
```


Once ps-streamd is running and has enabled periodic interrupts, the 'periodic_IRQ' will change to 'yes', and you should be able to see the interrupts building up in `/proc/interrupts`:

```
# cat /proc/interrupts
           CPU0
  0:  282649576          XT-PIC  timer
  1:     324356          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  8: 3063867103          XT-PIC  rtc
  9:          0          XT-PIC  Intel ICH2
 11:   30225769          XT-PIC  usb-uhci, eth0
 12:     734748          XT-PIC  PS/2 Mouse
 14:   10102374          XT-PIC  ide0
NMI:          0
LOC:  282649985
ERR:          0
MIS:          0
```


Index

Admission control....**41**
Admission Filter....4
  access....5
  Access control....**41**
  capacity....5
  Capacity Limits....**44**
  external....5
  External verification....**43**
  Per client limits....**43**
  per-ip....5
Admissions Manager....4
ANNOUNCE....24
Apple....**30**
APT....8
Autoplay....23
AVC....13
backlog....22
Big Buck Bunny....13
broadcast....34
Buffer sizes....**58**
Burst length....**59**
CentOS....**7**
Chunks, numbered....**39**
continuity counter....53
Controller....4, 22
  HLS....5
  HTTP....5
  permanent streams....5
  RTSP....5, **22**
  SOAP....5
daemon....20
Debian....**7**
Demonstration Edition....7
Demonstration version....45
DESCRIBE....26
Direct mapping....**36**
Directory Manager....4
Directory Provider....4
  Configuration files....5, **37**
  Direct mapping....5, **36**
  Gridline cache....5
  Gridline cache lookup....**39**
  Timeline capture....5
  Timeline capture lookup....**40**
Directory service....**36**
Empty packets....21
End of stream....25
End-of-stream....24
firewall....9
GET-PARAMETER....23
Gridline....5
H.264....13
HLS....**30**
HTTP....**29**

HTTP Live Streaming....5, **30**
immediate play....23
Index files....**46**
Index suffix....**36**
Indexing....13
init.d....17
Input....4
  file....6
  File input....**46**
Input Filter....4
Inputs....**46**
Installation....7
  CentOS....8
  Debian....8
  Red Hat....8
IP Time To Live (TTL)....**50**
IP Type of Service (TOS)....**50**
IPTV....40
keepalive....**21,** 23
Lag....**60**
Licence....20
Licensing....7
Logging....18, **19**
media types....28
MPEG-2....13
MTU....48
multicast....51
notification....25
O_DIRECT....46
Offset rounding....46
Output....4
  TCP....6
  UDP....6
Output Filter....4
  Adjust TS CC....6
  Adjust TS Timing....6
  HTTP Chunked....6
  HTTP chunked **52**
  RTP....6, **52**
  RTSP Interleave....6
  RTSP interleave **52**
  Transport Stream continuity counter Adjustment....**53**
  Transport Stream timing adjustment....**52**
Output Filters....**49, 52**
Outputs....**48**
Packet size....48
Permanent streams....**34**
Playlists....**38**
Precautionary principle....44
processes....18
ps-index-mpeg2ts....13
ps-streamd....**7**
Queue sizes....**59**
Quick start....13
Rate Profile....5
Rate Profile....
  fixed....6
  Fixed rate....**55**

limit....6
multiple....6
Rate limits....**56**
Rate multiple....**55**
Rate Profiles....49, **55**
Realtime scheduler....**62**
Realtime scheduling....**57**
RTC....18, **62**
RTP....27
RTSP....**22**
RTSP server....12, **22**
security....**20**
SOAP....**31**
Source address....49
SPTS....13
Stalling....16
Stream....4
Stream groups....**57**
Stream Manager....4
streamd.cfg.xml....17
streamd.log....18, 19
Streaming groups....48
Systems Administration....17
Target duration (HLS)....**31**
TCP/IP....**51**
Teardown....25
Teardown on disconnect....25
Threads....18, 23, **57**
Timeline....5
timeout....21, 23, 25
timestamp....19
TOS....**50**
transport....26
Transport mapping....26
ts-timing-adjust....52
TTL....**50**
UDP/IP....**50**
virtual chunks....**30**
VLC....14
watchdog....18, **20**
XMLMesh....**32**
yum....8
/dev/rtc....**62**

