A Review of HTTP Live Streaming

Andrew Fecheyr-Lippens
(andrew@fecheyr.be)

Abstract

TODO: write abstract

January 2010

Table of Contents

1.Introduction to streaming media

In the early 1990s consumer-grade personal computers became powerful enough to display video and playback audio. These early forms of computer media were usually delivered over non-streaming channels, by playing it back from CD-ROMs or by downloading a digital file from a remote web server and saving it to a local hard drive on the end user's computer. The latter can be referred to as "download-and-then-play" (Kurose & Ross [1]). Back then the main challenge for network delivery of media was the conflict between media size and bandwidth limit of the network.

During the early 2000s the internet saw a booming increase in network bandwidth. Together with better media compression algorithms and more powerful personal computer systems streaming delivery of media had become possible. The term *streaming* is used to describe a method of relaying data over a computer network as a steady continuous stream, allowing playback to proceed while subsequent data is being received. This is in contrast with *download-and-then-play,* where playback starts after the data has completely been downloaded. Instant playback is the main advantage of streaming delivery as the user no longer has to wait until the download is completed, which can take hours over slow connections.

On-demand vs live streaming

There are two classes of streaming media: on-demand streaming and live streaming. In the former case the media has been previously recorded and compressed. The media files are stored at the server and delivered to one or multiple receivers when requested (on-demand). Thousands of sites provide streaming of stored audio and video today, including Microsoft Video, YouTube, Vimeo and CNN. With live streaming on the other hand the media is captured, compressed and transmitted on the fly. Live streaming requires a significant amount of computing resources and often specific hardware support.

Streaming media offers the client the same functionality as a traditional VCR: pause, rewind, fast forwarding and indexing through the content. Of course fast-forwarding is only possible when streaming stored media.

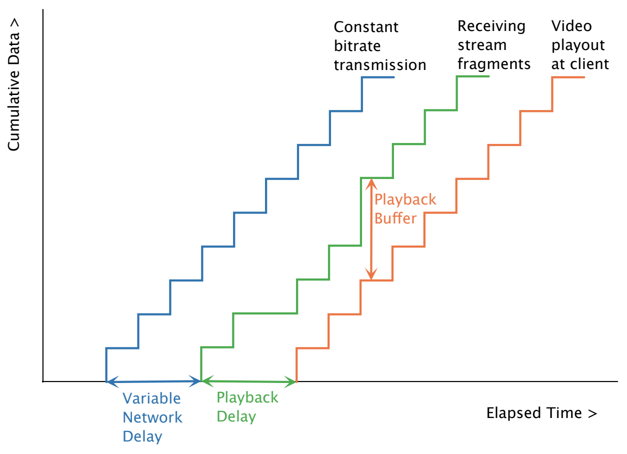


Fig 1. On-demand streaming (stored video)

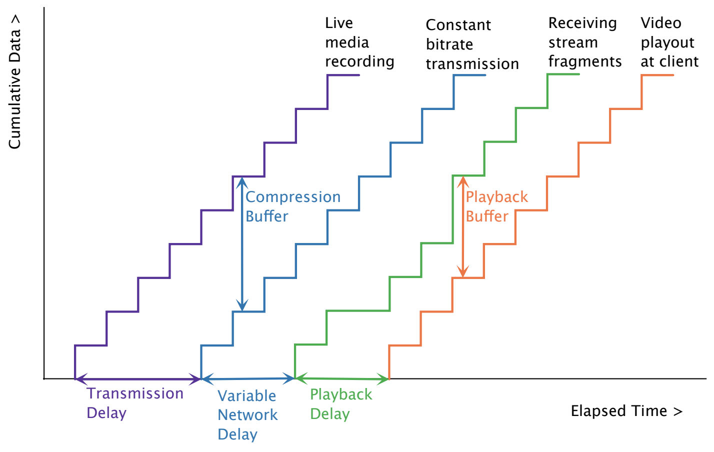


Fig 2. Live streaming

Figure 1 depicts the streaming of stored video. The vertical axis is the cumulative data transferred while the horizontal axis represents elapsed time. The leftmost stepped line represents the transmission in a constant bitrate. Due to network jitter (variations in network delay) the transmitted packets (or stream fragments) arrive at the client unevenly. To diminish these effects of network-induced jitter, a buffer is used and playback is delayed.

The diagram for live streaming (figure 2) adds an extra stepped line, which represents the capturing and recording of the live event. This audio/video input needs to be compressed on the fly and transmitted to listening clients. A compression buffer is used in order to achieve this, causing a transmission delay. The following steps and delay types are equal to those for streaming stored video.

2.Popular solutions

Two solutions that are heavily deployed in the marketplace today are RTP/RTSP and the Adobe Flash Media Streaming Server. We introduce both only briefly as a more thorough discussion of the underlying technologies is out of the scope of this paper.

RTP/RTSP

The Real Time Streaming Protocol (RTSP), originally developed by Netscape and Real in the late '90s, is a network control protocol used to control streaming media servers. The protocol is used to establish and control media sessions between end points. Clients of media servers issue VCR-like commands, such as play and pause, to facilitate real-time control of playback of media files from the server. RTSP was published as RFC 2326 in 1998[5].

The transmission of streaming data itself is not a task of the RTSP protocol. Most RTSP servers use the Real-time Transport Protocol (RTP) for media stream delivery, however some vendors implement proprietary transport protocols. The RTSP server from RealNetworks, for example, also features RealNetworks' proprietary RDT stream transport.

One of the drawbacks of RTSP is that by default it runs on a port (554) which is often blocked by firewalls. This problem is especially common with home Internet gateways.

Adobe Flash Media Streaming Server

Flash Media Server (FMS) is a proprietary data and media server from Adobe Systems (originally a Macromedia product). It can be use to stream video-on-demand and live video. The world's largest video website, YouTube, uses it to stream videos on demand.

The Flash Media Server works as a central hub with Flash based applications connecting to it using the Real Time Messaging Protocol (RTMP). The server can send and receive data to and from the connected users who have the Flash Player installed.

There are a few downsides to using this solution:

- The Flash Media Streaming Server is proprietary and costs $995[7].
- Users need to have the Flash Player installed. The impact of this problem has been reduced as the majority of internet users have downloaded the player over time.
- The Flash Player suffers from bad implementations on some platforms. Especially Linux and Mac OS have buggy implementations that causes high CPU load and thus drain batteries faster. This was the reason for Apple to exclude the Flash Player in it's mobile iPhone and iPod Touch devices.

3.Apple's HTTP Live Streaming

Apple originally developed HTTP Live Streaming to allow content providers to send live or prerecorded audio and video to Apple's iPhone and/or iPod Touch using an ordinary Web server. They later included the same playback functionality in their QuickTime desktop media player. Playback requires iPhone OS 3.0 or later on iPhone or iPod touch and QuickTime X on the desktop. The HTTP Live Streaming specification is an IETF Internet-Draft. The third version of the IETF Internet-Draft can be found online at tools.ietf.org[1].

This chapter will introduce the HTTP Streaming Architecture: how the technology works, what formats are supported, what you need on the server, and what clients are available. Afterwards another section will describe how to set up live broadcast or video on demand sessions, how to implement encryption and authentication, and how to set up alternate bandwidth streams. This chapter borrows heavily from Apple's documentation[2].

HTTP Streaming Architecture

Conceptually, HTTP Live Streaming consists of three parts: the server component, the distribution component, and the client software.

- **The server component** is responsible for taking input streams of media and encoding them digitally, encapsulating them in a format suitable for delivery, and preparing the encapsulated media for distribution.
- **The distribution component** consists of **standard web servers.** They are responsible for accepting client requests and delivering prepared media and associated resources to the client. For large-scale distribution, edge networks or other **content delivery networks** can also be used.
- **The client software** is responsible for determining the appropriate media to request, downloading those resources, and then reassembling them so that the media can be presented to the user in a continuous stream.

The iPhone includes built-in client software: the media player, which is automatically launched when Safari encounters an `<OBJECT>` or `<VIDEO>` tag with a URL whose MIME type is one that the

[1] http://tools.ietf.org/html/draft-pantos-http-live-streaming-02

media player supports. The media player can also be launched from custom iPhone applications using the media player framework.

QuickTime X can also play HTTP Live Streams, enabling playback on the desktop. Developers can use the QuickTime framework to create desktop applications that play HTTP Live Streams. The QuickTime plug-in allows you to embedded streams in websites for playback through a browser without writing any application code.

In a typical configuration, a hardware encoder takes audio-video input and turns it into an MPEG-2 Transport Stream, which is then broken into a series of short media files by a software stream segmenter. These files are placed on a web server.

The segmenter also creates and maintains **an index file** containing a list of the media files. The URL of the index file is published on the web server. Client software reads the index, then requests the listed media files in order and displays them without any pauses or gaps between segments.

An example of a simple HTTP streaming configuration is shown in Figure 3.

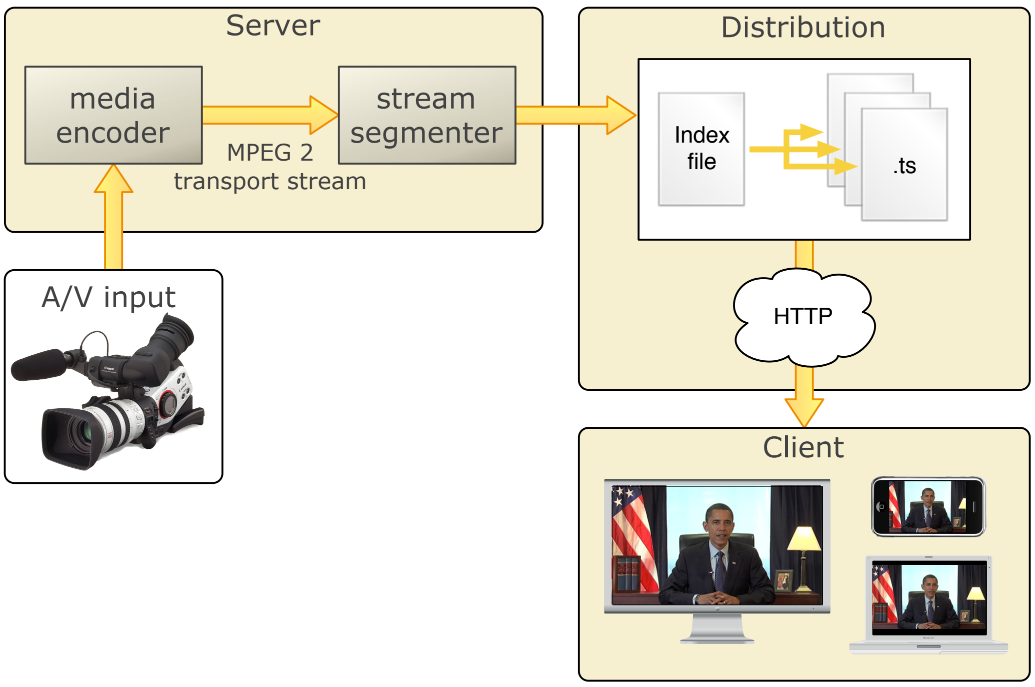


Fig 3. The HTTP Live Streaming Architecture

Input can be live or from a prerecorded source. It is typically encoded into an MPEG-2 Transport Stream by off-the-shelf hardware. The MPEG2 stream is then broken into segments and saved as a series of one or more `.ts` media files. This is typically accomplished using a software tool such as the Apple stream segmenter.

Audio-only streams can be a series of MPEG elementary audio files formatted as either AAC with ADTS headers or MP3.

The segmenter also creates an index file. The index file contains a list of media files and metadata. The index file is in `.M3U8` format. The URL of the index file is accessed by clients, which then request the indexed `.ts` files in sequence.

Server Components

The server requires a media encoder, which can be off-the-shelf hardware, and a way to break the encoded media into segments and save them as files, which can be software such as the media stream segmenter provided by Apple (available in beta for download from the Apple Developer Connection member download site[2]).

Media Encoder

The media encoder takes a real-time signal from an audio-video device, encodes the media, and encapsulates it for delivery. Currently, the supported format is MPEG-2 Transport Streams for audio-video, or MPEG elementary streams for audio. The encoder delivers an MPEG-2 Transport Stream over the local network to the stream segmenter.

The protocol specification is capable of accommodating other formats, but only MPEG-2 video streams (with H.264 video and AAC audio) or MPEG elementary audio streams (in AAC format with HTDS headers or in MP3 format) are supported at this time. It is important to note that the video encoder **should not change stream settings** — such as video dimensions or codec type — in the midst of encoding a stream.

Stream Segmenter

The stream segmenter is a process — typically software — that reads the Transport Stream from the local network and divides it into a series of small media files of equal duration. Even though each segment is in a separate file, video files are made from a continuous stream which can be reconstructed seamlessly.

[2] https://connect.apple.com/cgi-bin/WebObjects/MemberSite.woa/wa/getSoftware?bundleID=20389

The segmenter also creates an index file containing references to the individual media files. Each time the segmenter completes a new media file, the index file is updated. The index is used to track the availability and location of the media files. The segmenter may also encrypt each media segment and create a key file as part of the process.

Media segments are saved as `.ts` files (MPEG-2 streams) and index files are saved as `.M3U8` files, an extension of the `.m3u` format used for MP3 playlists. Because the index file format is an extension of the `.m3u` file format, and because the system also supports `.mp3` audio media files, the client software may also be compatible with typical MP3 playlists used for streaming Internet radio.

Here is a very simple example of an `.M3U8` file a segmenter might produce if the entire stream were contained in three unencrypted 10-second media files:

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-TARGETDURATION:10
#EXTINF:10,
http://media.example.com/segment1.ts
#EXTINF:10,
http://media.example.com/segment2.ts
#EXTINF:10,
http://media.example.com/segment3.ts
#EXT-X-ENDLIST
```

The index file may also contain URLs for encryption key files or alternate index files for different bandwidths. For details of the index file format, see the IETF Internet-Draft of the HTTP Live Streaming specification[3].

Media Segment Files

The media segment files are normally produced by the stream segmenter, based on input from the encoder, and consist of a series of `.ts` files containing segments of an MPEG-2 Transport Stream. For an audio-only broadcast, the segmenter can produce MPEG elementary audio streams containing either AAC audio with ADTS headers or MP3 audio.

Alternatively, it is possible to create the `.M3U8` file and the media segment files independently, provided they conform the published specification. For audio-only broadcasts, for example, you could create an `.M38U` file using a text editor, listing a series of existing `.MP3` files.

Distribution Components

The distribution system is a **web server** or a **web caching system** that delivers the media files and index files to the client over HTTP. No custom server modules are required to deliver the content, and typically very little configuration is needed on the web server.

Recommended configuration is typically limited to specifying MIME-type associations for `.M3U8` files and `.ts` files.

File extension	MIME type
`.M3U8`	`application/x-mpegURL`
`.ts`	`video/MP2T`

Table 1. Recommended MIME-type configuration

Tuning time-to-live (TTL) values for `.M3U8` files may also be necessary to achieve desired caching behavior for downstream web caches, as these files are frequently overwritten, and the latest version should be downloaded for each request.

Client Component

The client software begins by fetching the index file, based on a URL identifying the stream. The index file in turn specifies the location of the available media files, decryption keys, and any alternate streams available. For the selected stream, the client downloads each available media file in sequence. Each file contains a consecutive segment of the stream. Once it has a sufficient amount of data downloaded, the client begins presenting the reassembled stream to the user.

The client is responsible for fetching any decryption keys, authenticating or presenting a user interface to allow authentication, and decrypting media files as needed.

This process continues until the client encounters the `#EXT-X-ENDLIST` tag in the index file. If no `#EXT-X-ENDLIST` tag is encountered, the index file is part of an ongoing broadcast. The client loads a new version of the index file periodically. The client looks for new media files and encryption keys in the updated index and adds these URLs to its queue.

Using HTTP Live Streaming

Session Types

The HTTP Live Streaming protocol supports live broadcast sessions and video on demand (VOD) sessions.

For live sessions, as new media files are created and made available the index file is updated. The updated index file includes the new media files; older files are typically removed. The updated index file presents a moving window into a continuous stream. Figure 4 shows a graphical representation of what is called a "Sliding Window Playlist" in the IETF Draft[3]. Whenever a new segment is ready, the index file is updated to include the newest segment and to remove the oldest one. This way the index file always contains the *x* latest segments. Apple recommends that *x* be larger than 3 so that the client can pause, resume and rewind for at least the duration of 3 segments (30 seconds by default).

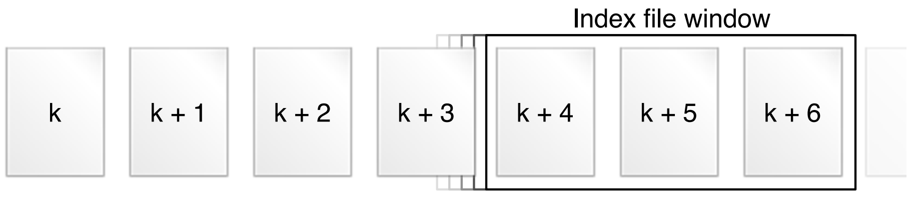


Fig 4. Index file of a live session, updated for every new segment.

For VOD sessions, media files are available representing the entire duration of the presentation. The index file is static and contains a complete list of all files created since the beginning of the presentation. This kind of session allows the client full access to the entire program.

It is possible to create a live broadcast of an event that is instantly available for video on demand. To convert a live broadcast to VOD, do not remove the old media files from the server or delete their URLs from the index file, and add an `#EXT-X-ENDLIST` tag to the index when the broadcast ends. This allows clients to join the broadcast late and still see the entire event. It also allows an event to be archived for rebroadcast with no additional time or effort.

VOD can also be used to deliver "canned" media. It is typically more efficient to deliver such media as a single QuickTime movie or MPEG-4 file, but HTTP Live Streaming offers some advantages, such as support for media encryption and dynamic switching between streams of different data rates in response to changing connection speeds. (QuickTime also supports multiple-data-rate movies using progressive download, but QuickTime movies do not support dynamically switching between data rates in mid-movie.)

Content Protection

Media files containing stream segments may be individually encrypted. When encryption is employed, references to the corresponding key files appear in the index file so that the client can retrieve the keys for decryption.

When a key file is listed in the index file, the key file contains a cipher key that must be used to decrypt subsequent media files listed in the index file. Currently HTTP Live Streaming supports AES-128 encryption using 16-octet keys. The format of the key file is a packed array of these 16 octets in binary format.

The media stream segmenter available from Apple provides encryption and supports three modes for configuring encryption.

1. The first mode allows you to specify a path to an existing key file on disk. In this mode the segmenter inserts the URL of the existing key file in the index file. It encrypts all media files using this key.

2. The second mode instructs the segmenter to generate a random key file, save it in a specified location, and reference it in the index file. All media files are encrypted using this randomly generated key.

3. The third mode instructs the segmenter to generate a random key file, save it in a specified location, reference it in the index file, and then regenerate and reference a new key file every *n* files. This mode is referred to as key rotation. Each group of *n* files is encrypted using a different key.

All media files may be encrypted using the same key, or new keys may be required at intervals. The theoretical limit is one key per media file, but because each media key adds a file request and transfer to the overhead for presenting the following media segments, changing to a new key periodically is less likely to impact system performance than changing keys for each segment.

Key files can be served using either HTTP or HTTPS. You may also choose to protect the delivery of the key files using your own session-based authentication scheme.

Caching and Delivery Protocols

HTTPS is commonly used to deliver key files. It may also be used to deliver the content files and index files, but this is not recommended when scalability is important, since HTTPS requests often bypass web server caches, causing all content requests to be routed through your server and defeating the purpose of edge network distribution systems.

For this very reason, however, it is important to make sure that any **content delivery network** you use understands that the `.M3U8` index files are not to be cached for longer than one media segment duration.

Stream Alternatives

Index files may reference alternate streams of content. References can be used to support delivery of multiple streams of the same content with varying quality levels for different bandwidths or devices. The client software uses heuristics to determine appropriate times to switch between the alternates. Currently, these heuristics are based on recent trends in measured network throughput.

The index file points to alternate streams of media by including a specially tagged list of other index files, as illustrated in Figure 5.

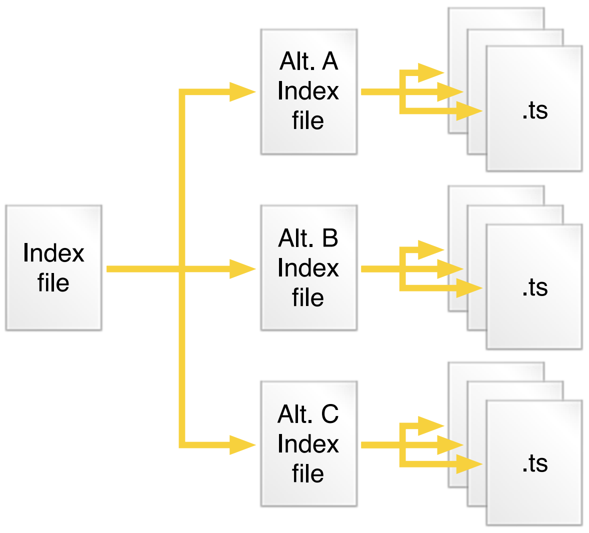


Fig 5. Alternate streams

Note that the client may choose to change to an alternate stream at any time, such as when a mobile device enters or leaves a WiFi hotspot.

Failover Protection

If your playlist contains alternate streams, they can not only operate as bandwidth or device alternates, but as failure fallbacks. Starting with iPhone OS 3.1, if the client is unable to reload the index file for a stream (due to a 404 error, for example), the client attempts to switch to an alternate stream.

In the event of an index load failure on one stream, the client chooses the highest bandwidth alternate stream that the network connection supports. If there are multiple alternates at the same bandwidth, the client chooses among them in the order listed in the playlist.

You can use this feature to provide redundant streams that will allow media to reach clients even in the event of severe local failures, such as a server crashing or a content distributor node going down.

To implement failover protection, create a stream, or multiple alternate bandwidth streams, and generate a playlist file as you normally would. Then create a parallel stream, or set of streams, on a separate server or content distribution service. Add the list of backup streams to the playlist file, so that the backup stream at each bandwidth is listed after the primary stream. For example, if the primary stream comes from server ALPHA, and the backup stream is on server BETA, your playlist file might look something like this:

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=200000
http://ALPHA.mycompany.com/lo/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=200000
http://BETA.mycompany.com/lo/prog_index.m3u8

#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=500000
http://ALPHA.mycompany.com/md/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=500000
http://BETA.mycompany.com/md/prog_index.m3u8
```

Note that the backup streams are intermixed with the primary streams in the playlist, with the backup at each bandwidth listed after the primary for that bandwidth.

You are not limited to a single backup stream set. In the example above, ALPHA and BETA could be followed by GAMMA, for instance. Similarly, you need not provide a complete parallel set of streams. You could provide a single low-bandwidth stream on a backup server, for example.

4.Critical Comparison

Now that the HTTP Live Streaming protocol by Apple has been explained we can examine how it compares to two currently popular solutions: RTP/RTSP and Flash Media Streaming. For the tests I used a video clip from the 2009 documentary "Objectified" about industrial design. The clip shows an interview of Jonathan Ive and has the following characteristics:

Filename	jonathan_ive_design.mp4
Video Codec	H.264
Audio Codec	AAC
Dimensions	480 × 270
Duration	0:05:50
Filesize	24,6 MB

Table 2. Source video characteristics

I built a test website at http://streaming_test.andrewsblog.org that streams the same video using the three different technologies. Apple's HTTP Live Streaming has two entries: one created using Apple's tools and one created with an experimental open source toolchain. I did not buy a license for Adobe's Flash Media Streaming Server so the flash version of the video is simply embedded from YouTube. The source code of the test website is available online on github[3].

Ease of Setup

I will briefly describe the steps taken to setup the videos for streaming using HTTP Live Streaming with Apple's developer tools and an experimental open source toolchain as well as using RTP/RTSP with the Darwin Streaming Server. Unfortunately the Adobe Flash Media Streaming Server could not be tested as the server software is proprietary and costs $995.

[3] http://github.com/andruby/StreamingMedia-Test

HTTP Live Streaming - Apple tools

After installing Apple's HTTP Live Streaming developer tools[4] we only needed one command to convert the source video to a suitable format and segment it.

```
mediafilesegmenter -t 10 -f t_10/ jonathan_ive_design.mp4
```

The first parameter (-t | -target-duration) is the desired duration of each segment. I chose 10 seconds because that is the default. The second parameter (-f | -file-base) is the directory to store the media and index files. The command took only 3 seconds on a 1.8Ghz Macbook as no transcoding had to be done. Appendix 1 contains the index file generated by the tool.

The above command created the index file at path `t_10/prog_index.m3u8` and all 36 segments at path `t_10/fileSequenceX.ts` with X from 0 to 35. Embedding the stream into an html file is incredibly easy with the HTML5 `<video>` tag:

```
<video src='t_10/prog_index.m3u8' controls='on'>
  Video tag not supported
</video>
```

Because the segments and the index file are standard HTTP objects ready to be served by any web server, there is nothing more to do if you already have one. Apache, nginx, IIS, lighttpd, etc all work out of the box without the need for a special module or specific configuration.

The setup was really easy, the tools provided by Apple are well documented and all in all it took less than 4 minutes to put the stream online.

HTTP Live Streaming - Open Source tools

One downside of the tools Apple offers is that they only work on Apple's platform. It might be possible to port them to Linux/BSD but that hasn't been done to date. One open source developer, Carson McDonald[4], has built an open source toolchain around FFMpeg, ruby and libavformat that does everything and more than Apple's tools. The project is available online on github[5] under the GPLv2 license.

[4] https://connect.apple.com/cgi-bin/WebObjects/MemberSite.woa/wa/getSoftware?bundleID=20389
[5] http://github.com/carsonmcdonald/HTTP-Live-Video-Stream-Segmenter-and-Distributor

I used this toolchain on a FreeBSD system to built a multirate stream with 3 quality levels at bitrates of 128kbps, 386kbps and 512kbps.The configuration file used for the toolchain is included in Appendix 2 and the resulting index files in Appendix 3.

Using this toolchain took a little longer than Apple's tool as it had to transcode the input video in 3 different bitrates. Nevertheless, setting up the video for streaming only took about 10 minutes. For more information visit the developers project page[6].

RTP/RTSP - Darwin Streaming Server

Installing the open source Darwin Streaming Server (DSS) on FreeBSD is staight forward with the `portinstall` or `pkg_add` command.

To configure the server I had to create a user with the `qtpasswd` command and add the user to the admin group in `/usr/local/etc/streaming/qtgroups`. When the server is started a Setup Assistant Wizard is available though a web interface on port 1220. The assistant suggested to setup streaming on port 80 to stream through firewalls. However, this might interfere with any web servers installed. I needed to bind the web server to one IP address and DSS to another.

The biggest issue however was that media files have to be "hinted" in order to be streamable by DSS over RTSP. I had to install the toolkit from MPEG4IP project[7] in order to *hint* the source video. Installing that package on FreeBSD took 20 minutes.

"Hinting" the `.mp4` file required these 2 commands:

```
mp4creator -hint=1 jonathan_ive_design.mp4
mp4creator -hint=2 jonathan_ive_design.mp4
```

Embedding the stream into our test website was harder, and doesn't work with FireFox so a link to open the stream with an external player was added.

Time to setup the video stream: ~ 1 hour

[6] http://www.ioncannon.net/projects/http-live-video-stream-segmenter-and-distributor
[7] http://www.mpeg4ip.net

Compatibility

One of the compatibility issues that originally impeded the growth of RTSP was the fact that it runs on a port number (554) which is often blocked by consumer firewalls. Most RTSP servers offer the functionality to tunnel RTSP communication over the same port as HTTP (port 80). We setup DSS to use this function to minimize firewall issues. For the same reason, the communication protocol of the Flash Media Streaming Server, RTMP, is encapsulated within HTTP requests.

HTTP Live Streaming natively runs on the HTTP port and thus has no firewall issues. This is one of the main motives why Apple developed the protocol for it's mobile internet devices.

We opened the test website (http://streaming_test.andrewsblog.org) with several different browsers on 3 separate platforms and tested if each of the technologies worked. The results are displayed in Table 3.

	HTTP Live Streaming	RTP/RTSP	Flash Media
Safari 4.0.4, MacOS 10.6.2	yes	yes	yes
FireFox 3.5.7, MacOS 10.6.2	external player	external player	yes
Chrome 4.0.249, MacOS 10.6.2	external player	external player	yes
FireFox 3.5, Vista SP1	no	external player	yes
IExplorer 8, Vista SP1	no	external player	yes
iPhone 3.1.2	yes	no	redirected*

* YouTube automatically sends an alternative HTTP Live Stream to iPhone users.

Table 3. Compatibility test

It is clear that the winner of this test is Flash Media, as it could be played in every situation. Important to note here is that YouTube chooses to offer HTTP Live Streams for iPhone visitors. Because of this, the HTTP Live Streaming technology suddenly gained a huge installed base, as all the video's on YouTube are also available through this streaming technology.

However, HTTP Live Streaming still suffers from lacking adoption on the desktop. Hopefully this will change when it becomes an official IETF standard in the near future. Until broad support for Apple's new technology is a fact, the "fall-through" mechanism of HTML5 can be used: If the video tag is not recognized, the browser will attempt to render the next child tag. This is extremely useful, allowing a runtime like Flash to handle playback on browsers that do not yet support HTML 5 video playback. The video tag can also fall-through across multiple sources, in the event that a browser does not support a particular file format or compression. The HTML 5 specification defines the syntax for the video tag, but it is up to the browser which media formats to support. The code for this example is available on page 9 of Akamai's HD for iPhone Encoding Best Practices[10].

Features

HTTP Live Streaming offers alternate streams with different bitrates. The client automatically switches to the optimal bitrate based on network conditions for a smooth quality playback experience. For streaming media to mobile devices with different types of network connections this is a killer feature. It is sometimes referred to as "Adaptive Bitrate". During the testing of the OS toolchain stream the switching of bitrates was clear in the video quality and was recorded in the web server access logs. I encourage you to checkout an excerpt of that logfile in Appendix 4. The Akamai recommended bitrates used for encoding are included in Appendix 5 as a reference.

Some RTSP streaming servers offer a similar "bit rate adaption" for 3GPP files[8]. The newest version of Adobe's Flash Media Streaming Server (3.5) might offer an equivalent solution called "Dynamic Streaming"[9].

Video encryption and authentication over HTTPS are two other features of HTTP Live Streaming of great value to content providers who want to protect their source of income. Authentication can also be built into RTSP and Flash servers. The newest version of Adobe's Flash Media Streaming Server offers encrypted H.264 streaming and encrypted Real Time Messaging Protocol (RTMPE).

All three protocols offer VCR like functionality, such as play/pause and timeshifting seek. While testing we found that the RTSP Stream from DSS was the most responsive to seeking. The HTTP Live Stream suffered from a noticable delay because a full 10 second segment needs to be downloaded before playback can resume. This delay can be significantly reduced by using shorter segments. The specification draft allows segment durations with a minimum of 1 second. The tradeoff here is a higher overhead, as more segments need to be requested. Seeking though YouTube video's causes variable delays, which are sometimes better than the HTTP Live Stream and sometimes worse.

[8] http://www.apple.com/quicktime/streamingserver/
[9] http://www.adobe.com/products/flashmediastreaming/features/

Ease of Distribution

One of the main advantages of HTTP Live Streaming is the fact that all parts that need to be transferred are standard HTTP objects. These objects can be cached by proxy servers without any modification. Because of this Content Distribution Networks (CDNs) can distribute HTTP Live Streams just as they would distribute other HTTP objects such as images or html pages.

Akamai, the world's leading CDN, has built a specific solution for media delivery to the iPhone platform using HTTP Live Streaming[8]. But also the smaller pay-as-you-go CDN's, such as Amazon's Cloudfront service, can be used without a problem. In fact, the open source toolchain used to setup the 2nd stream has the ability to upload segments directly to Amazon's S3 Storage and use Amazon's Cloudfront CDN. Figure 6 shows a diagram of the dataflow from an article of the developer[9].

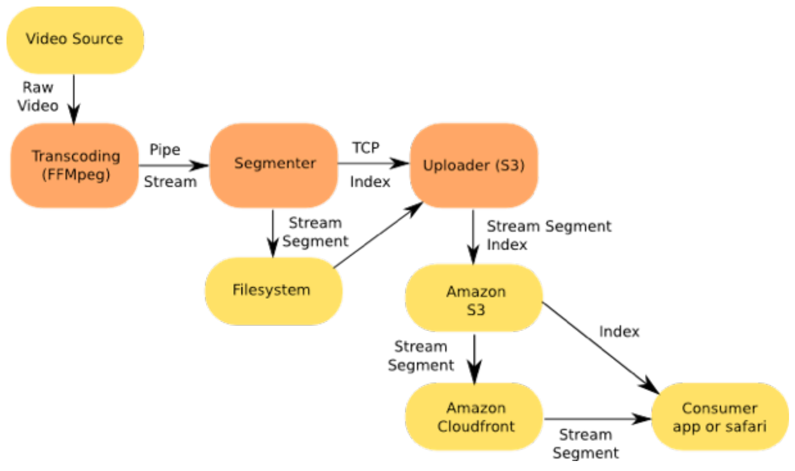


Fig 6. Opensource Toolchain with Amazon's Cloudfront CDN

As a consequence, everyone can deploy a live or on-demand stream on top of a worldwide CDN in a very cost effective way. The author of the article calculated the cost and concluded the following:

"For 100 streams of 5 minutes worth of video you would be looking at something around 20 cents"[9]

In other words: for every $1 you can stream a total of 2500 minutes[10] from any of the 14 edge locations of Amazon's global CDN. Figure 7 shows these 14 edge locations on a world map.

[10] (100 streams x 5 minutes) / 0.20 dollar

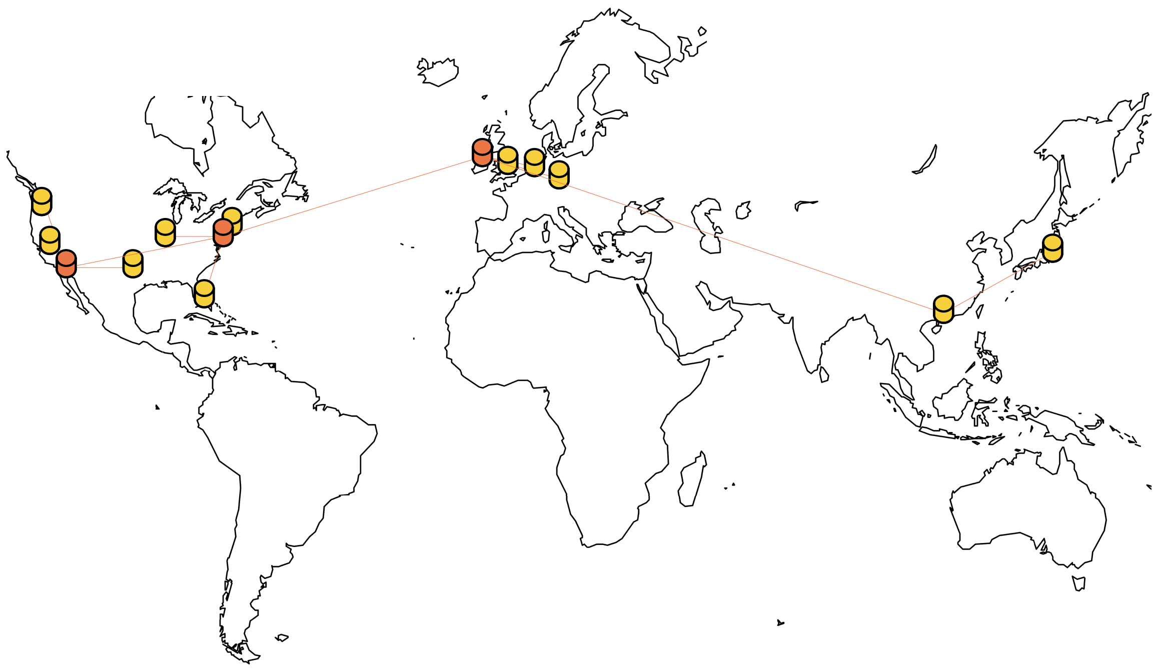


Fig 7. Amazon's Cloudfront CDN Servers

Cost

We previously mentioned that Adobe's Flash Media Streaming Server has a licence fee of $995. There is, however, a reverse engineered open source project called Red5[11] which aims to produce a feature-complete implementation written in Java.

HTTP Live Streaming and RTSP are open protocols and thus can be used without paying license fees. Both have open source tools and/or servers available for free.

[11] http://www.osflash.org/red5

5.Conclusion

Apple identified what it considers a few issues with standard streaming, which generally uses the Real Time Streaming Protocol. The biggest issue with RTSP is that the protocol or its necessary ports may be blocked by routers or firewall settings, preventing a device from accessing the stream. HTTP, being the standard protocol for the Web, is generally accessible. Furthermore, no special server is required other than a standard HTTP server, which is more widely supported in content distribution networks. In addition, more expertise in optimizing HTTP delivery is generally available than for RTSP.

Enter HTTP Live Streaming. The basic mechanics involve using software on the server to break an MPEG-2 transport stream into small chunks saved as separate files, and an extension to the `.m3u` playlist specification to tell the client where to get the files that make up the complete stream. The media player client merely downloads and plays the small chunks in the order specified in the playlist, and in the case of a live stream, periodically refreshes the playlist to see if there have been any new chunks added to the stream.

The new streaming protocol supports adaptive bitrates and automatically switches to the optimal bitrate based on network conditions for a smooth quality playback experience. The protocol also provides for media encryption and user authentication over HTTPS, allowing publishers to protect their work.

Currently the standard is an Internet-Draft, and we have yet to see any evidence that others are ready to jump on Apple's bandwagon. But given the issues Apple has identified with RTSP, the proprietary nature of other common solutions, and the inclusion of support for encrypted streams, it has potential to appeal to a wide variety of content providers.

In this paper we have tested and compared the proposed standard alongside two popular and widely used technologies. We deployed the three streaming technologies on a publicly accessible website[12] and compared the setup difficulty, the features and costs of each alternative. We also mentioned Content Distribution Networks and the role and impact of the simplified distribution of standard HTTP objects.

The findings of our comparison are summarized in the table on the following page.

[12] http://streaming_test.andrewsblog.org

	HTTP Live Streaming	RTP/RTSP	Flash Media
Protocol Type	Standard	Standard	Proprietary
Server Cost	Free	Free	$995
Ease of setup	Very easy	Okay	Unknown
Compatibility	Low	Using plugins	Very high
Client Implementation	Great (especially for embedded devices)	Good	High CPU load on some platforms
Server Implementation	Generic web server	Special streaming server	Proprietary or reversed engineered
Variable bitrates	Built in	Some servers	Only latest version
Encryption	Built in	Unknown	~ Server version
Authentication	Possible by protecting the key files	Through custom module[11]	Unknown
Built in Failover	Yes	No	No
Seek delay	Depends on segments duration and bitrate	< 2 seconds	4-10 seconds
Ease of distribution	Very easy	Possible	Possible
When to use?	**Embedded devices**	**Flexible seeking**	**Highest compatibility**

As a result we can conclude that each technology has its merits and pitfalls. HTTP Live Streaming is a young protocol that proves its value in embedded devices and is noteworthy for its simplicity. Deployment and distribution are remarkably easy and might open up video streaming to a much wider audience. Standardization and further adaption by other vendors and the open source community could boost the importance of this protocol in the upcoming HTML5 specification.

Adobe Flash is a good solution — popularized by YouTube — for embedding videos in a webpage when worried about client compatibility. Because of this broad compatibility it is also a good candidate as a fallback technology for any of the other two protocols. Nevertheless, the proprietary nature and the lack of adoption in the mobile market are two important disadvantages.

RTSP remains a flexible solution for latency sensitive live or on-demand streaming. Tunneling over HTTP, as implemented in several streaming servers,has alleviated its biggest issue. The protocol allows streaming different formats to a wide variety of devices but is more of a hassle to setup and distribute.

6.Appendices

Appendix 1 - Index file generated by Apple tool

The contents of the `prog_index.m3u8` index file generated by `mediafilesegmenter`:

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10,
fileSequence0.ts
#EXTINF:10,
fileSequence1.ts
#EXTINF:10,
fileSequence2.ts
#EXTINF:10,
fileSequence3.ts
#EXTINF:10,
fileSequence4.ts
#EXTINF:10,
fileSequence5.ts
#EXTINF:10,

[...]

#EXTINF:10,
fileSequence32.ts
#EXTINF:10,
fileSequence33.ts
#EXTINF:10,
fileSequence34.ts
#EXTINF:1,
fileSequence35.ts
#EXT-X-ENDLIST
```

Appendix 2 - Configuration file for the O.S. toolchain

Configuration file for the Open Source toolchain (in YAML format), items changed from the default value are highlighted in green.

```
temp_dir: '/tmp/'
segment_prefix: 'os_sample'
index_prefix: 'os_stream'

# type of logging: STDOUT, FILE
log_type: 'STDOUT'
log_level: 'DEBUG'

# where the origin video is going to come from
input_location: 'jonathan_ive_design.mp4'

# segment length in seconds
segment_length: 10

# this is the URL where the stream (ts) files will end up
url_prefix: ""

# this command is used for multirate encodings and is what pushes the encoders
source_command: 'cat %s'

# This is the location of the segmenter
segmenter_binary: './live_segmenter'

# the encoding profile to use
encoding_profile: [ 'ep_128k', 'ep_386k', 'ep_512k' ]

# The upload profile to use
transfer_profile: 'copy_dev'

# Encoding profiles
ep_128k:
  ffmpeg_command: "ffmpeg -er 4 -y -i %s -f mpegts -acodec libmp3lame -ar 48000 -ab 64k -s 480x270
  -vcodec libx264 -b 128k -flags +loop -cmp +chroma -partitions +parti4x4+partp8x8+partb8x8 -subq 5
  -trellis 1 -refs 1 -coder 0 -me_range 16 -keyint_min 25 -sc_threshold 40 -i_qfactor 0.71 -bt 128k
  -maxrate 128k -bufsize 128k -rc_eq 'blurCplx^(1-qComp)' -qcomp 0.6 -qmin 10 -qmax 51 -qdiff 4
  -level 30 -aspect 480:270 -g 30 -async 2 - | %s %s %s %s %s"
  bandwidth: 128000

ep_386k:
  ffmpeg_command: "ffmpeg -er 4 -y -i %s -f mpegts -acodec libmp3lame -ar 48000 -ab 64k -s 480x270
  -vcodec libx264 -b 386k -flags +loop -cmp +chroma -partitions +parti4x4+partp8x8+partb8x8 -subq 5
  -trellis 1 -refs 1 -coder 0 -me_range 16 -keyint_min 25 -sc_threshold 40 -i_qfactor 0.71 -bt 386k
  -maxrate 386k -bufsize 386k -rc_eq 'blurCplx^(1-qComp)' -qcomp 0.6 -qmin 10 -qmax 51 -qdiff 4
  -level 30 -aspect 480:270 -g 30 -async 2 - | %s %s %s %s %s"
  bandwidth: 386000

ep_512k:
  ffmpeg_command: "ffmpeg -er 4 -y -i %s -f mpegts -acodec libmp3lame -ar 48000 -ab 64k -s 480x270
  -vcodec libx264 -b 512k -flags +loop -cmp +chroma -partitions +parti4x4+partp8x8+partb8x8 -subq 5
  -trellis 1 -refs 1 -coder 0 -me_range 16 -keyint_min 25 -sc_threshold 40 -i_qfactor 0.71 -bt 512k
  -maxrate 512k -bufsize 512k -rc_eq 'blurCplx^(1-qComp)' -qcomp 0.6 -qmin 10 -qmax 51 -qdiff 4
  -level 30 -aspect 480:270 -g 30 -async 2 - | %s %s %s %s %s"
  bandwidth: 512000

copy_dev:
  transfer_type: 'copy'
  directory: '/var/www/vps.bedesign.be/http_live/os_10'
```

Appendix 3 - Index files generated by O.S. toolchain

Master index file pointing to 3 alternative streams with different bitrates (128k, 386k and 512k)

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=128000
os_stream_ep_128k.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=386000
os_stream_ep_386k.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=512000
os_stream_ep_512k.m3u8
```

File os_10/os_stream_multi.m3u8

Index file of the 386k alternate stream (similar to the index file in Appendix 1)

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:1
#EXTINF:10
os_sample_ep_386k-00001.ts
#EXTINF:10
os_sample_ep_386k-00002.ts
#EXTINF:10
os_sample_ep_386k-00003.ts
#EXTINF:10
os_sample_ep_386k-00004.ts

[...]

#EXTINF:10
os_sample_ep_386k-00032.ts
#EXTINF:10
os_sample_ep_386k-00033.ts
#EXTINF:10
os_sample_ep_386k-00034.ts
#EXT-X-ENDLIST
```

File os_10/os_stream_ep_386k.m3u8

Appendix 4 - Web server access log

An excerpt of the access log of the web server demonstrating the switch in requested bitrate.

```
88.16.217.147 - [19/Jan/2010:00:09:03] "GET /http_live_oss HTTP/1.1" 200 2036
88.16.217.147 - [19/Jan/2010:00:09:05] "GET /os_10/os_stream_multi.m3u8 HTTP/1.1" 200 221
88.16.217.147 - [19/Jan/2010:00:09:05] "GET /os_10/os_stream_ep_128k.m3u8 HTTP/1.1" 200 1325
88.16.217.147 - [19/Jan/2010:00:09:09] "GET /os_10/os_sample_ep_128k-00001.ts HTTP/1.1" 200 446312
88.16.217.147 - [19/Jan/2010:00:09:12] "GET /os_10/os_sample_ep_128k-00002.ts HTTP/1.1" 200 504780
88.16.217.147 - [19/Jan/2010:00:09:15] "GET /os_10/os_sample_ep_128k-00003.ts HTTP/1.1" 200 500080
88.16.217.147 - [19/Jan/2010:00:09:16] "GET /os_10/os_stream_ep_512k.m3u8 HTTP/1.1" 200 1363
88.16.217.147 - [19/Jan/2010:00:09:26] "GET /os_10/os_sample_ep_512k-00003.ts HTTP/1.1" 200 1370896
88.16.217.147 - [19/Jan/2010:00:09:33] "GET /os_10/os_sample_ep_512k-00004.ts HTTP/1.1" 200 1168984
88.16.217.147 - [19/Jan/2010:00:09:41] "GET /os_10/os_sample_ep_512k-00005.ts HTTP/1.1" 200 1193988
```

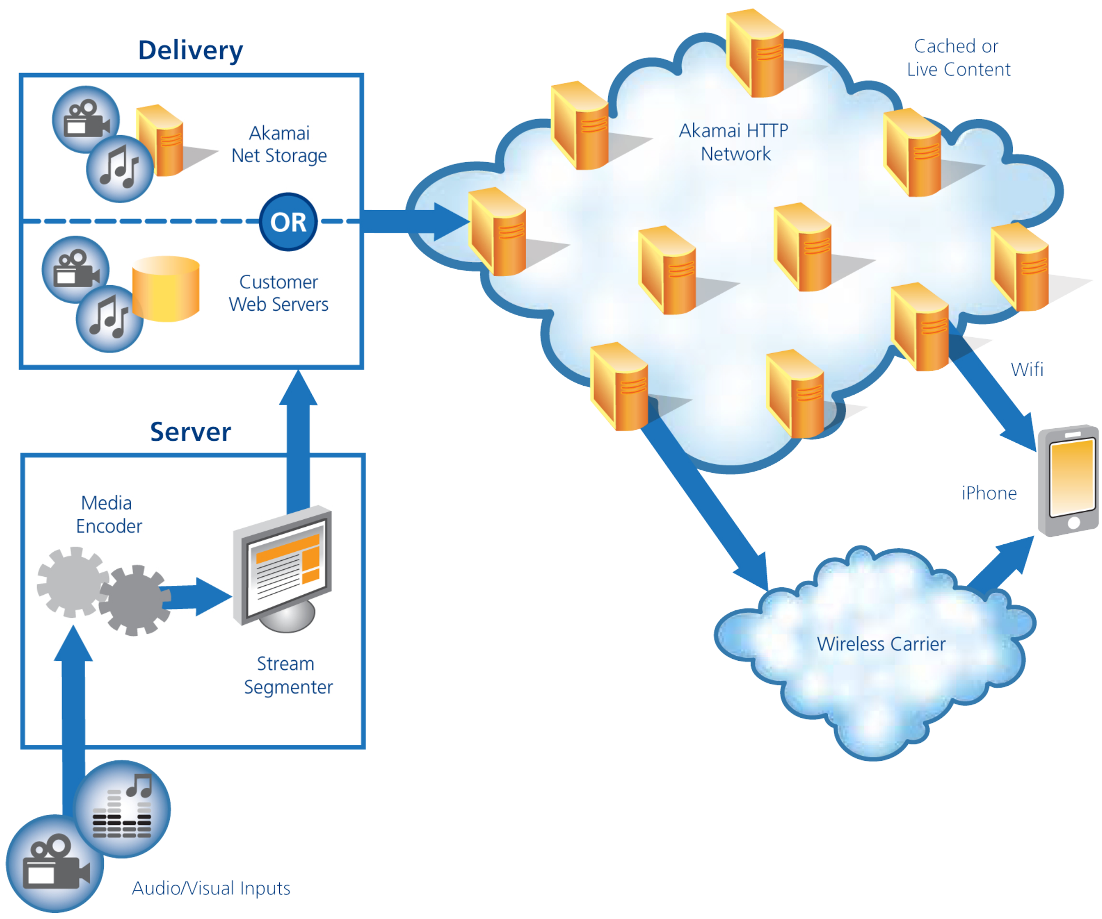
As can be seen from this excerpt, the client media player first requests the master index file (`os_stream_multi.m3u8`), parses it and requests the 1st stream alternative index file (128kbps) (`os_stream_ep_128k.m3u8`). After playing the first 2 segments, the player switches to a higher bitrate alternative (512kbps), requests the index (`os_stream_ep_512k.m3u8`) and the segments starting from segment 3.

An interactive video demonstrating this progressive enhancement is available on YouTube[1].

[1] http://www.youtube.com/watch?v=teKAyN0qZVY

Appendix 5 - Akamai HD for iPhone architecture

The architecture of Akamai's HD for iPhone solution[10].



Recommended encoding bitrates depending on network connection (page 7 of [10])

Delivery	Dimensions	Frame Rate	Total Bit Rate	Video Bit Rate	Audio Bit Rate	Audio Sample Rate
Aspect 4:3						
Edge	360x270	1/2 Current	112	85	32	16
3G (Low)	360x270	Current	314	250	64	32
3G (High)	360x270	Current	514	450	64	32
WiFi	360x270	Current	864	800	64	32
Aspect 16:9						
Edge	400x224	1/2 Current	112	85	32	16
3G (Low)	400x224	Current	314	250	64	32
3G (High)	400x224	Current	514	450	64	32
WiFi	400x224	Current	864	800	64	32

7.References

[1] James F. Kurose, Keith W. Ross, 2009. Computer Networking: A Top Down Approach: 5th edition. Addison Wesley, Chapter 7.

[2] Apple Inc., 2009. HTTP Live Streaming Overview. PDF available online at: http://developer.apple.com/iphone/library/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide

[3] Roger Pantos, 2009, Apple Inc. HTTP Live Streaming: draft-pantos-http-live-streaming-02. Online PDF: http://tools.ietf.org/pdf/draft-pantos-http-live-streaming-02.pdf

[4] Carson McDonald, 2009, HTTP Live Video Stream Segmenter and Distributor, http://www.ioncannon.net/projects/http-live-video-stream-segmenter-and-distributor/

[5] H. Schulzrinne, Real Time Streaming Protocol, RFC 2326, http://tools.ietf.org/html/rfc2326

[6] Real Time Streaming Protocol, Wikipedia http://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol

[7] Adobe, Flash Media Streaming Server 3.5, http://www.adobe.com/products/flashmediastreaming/

[8] Akamai, Akamai HD for iPhone, http://www.akamai.com/html/solutions/iphone.html

[9] Carson McDonald, iPhone Windowed HTTP Live Streaming Using Amazon S3 and Cloudfront Proof of Concept, http://www.ioncannon.net/programming/475/iphone-windowed-http-live-streaming-using-amazon-s3-and-cloudfront-proof-of-concept/

[10] Akamai, Akamai HD for iPhone Encoding Best Practices, http://www.akamai.com/dl/akamai/iphone_wp.pdf

[11] Koushik Biswas, Darwin Streaming Server 6.0.3 - setup, customization, plugin and module development, http://www.codeproject.com/KB/audio-video/DarwinSS_on_Linux.aspx

[12] Ars Technica, Apple proposes HTTP streaming feature as IETF standard, http://arstechnica.com/web/news/2009/07/apple-proposes-http-streaming-feature-as-a-protocol-standard.ars