Adaptive HTTP Streaming Framework

Release 1.0 alpha

www.longtailvideo.com

February 17, 2011

CONTENTS

CHAPTER

ONE

ABOUT ADAPTIVE STREAMING

This guide offers an introduction to adaptive streaming in general, as well as to currently existing solutions.

1.1 Current streaming methods

At present, online video is mostly delivered using one of the following two methods:

- *Progressive download*; video is delivered to the player as one big download. Used by e.g. Vimeo.
- *Stateful streaming*; video is delivered to the player over a persistent connection. Used by e.g. Hulu.

Both methods have a number of cons that prevent them from overtaking the other for online video streaming:

- Progressive download inherently wastes a lot of bandwidth. For example, a user might start to watch a 10-minute video, but stop after 1 minute. Usually, the entire video is downloaded at that point.
- Progressive download cannot adapt to changing network (e.g. a drop in bandwidth) or client conditions (e.g. a jump to fullscreen).
- Progressive download does not support live streaming.
- Stateful streaming requires specialized streaming servers (like Flash Media Server) and fast, dedicated hardware.
- Stateful streaming needs dedicated protocols (i.e. not TCP/HTTP/80) and a dedicated CDN / caching infrastructure.
- Stateful streaming is very fragile. One hickup in the connection and playback is interrupted.

1.2 Enter adaptive streaming

Adaptive streaming is a third video delivery method, addressing these shortcomings. An adaptive stream consists of hundreds of small video *fragments*, each a few seconds long, seamlessly glued together to one video in the streaming client.

- Adaptive streaming leverages existing HTTP webservers, networking and caching infrastructure.
- At the same time, it can still adapt to network and client changes, simply by loading successive fragments in a higher or lower quality.
- It can be leveraged for live streaming, with the next fragment made available *just in time*.
- Since the client is in full control of all networking operations, it can quickly anticipate and recover from errors

In order to keep track of all available quality levels and fragments, adaptive streams always include a small text or XML file that contains basic information of all available quality levels and fragments. This so-called *manifest* is the URL and single entry point for an adaptive stream.

Adaptive streaming clients load the manifest first. Next, they start loading video fragments, generally in the best possible quality that is available at that point in time.

1.3 Adaptive streaming solutions

Currently, three adaptive streaming solutions are available to publishers. They follow the same premises, but there's small differences in implementation:

- Apple's HTTP Live Streaming is probably the most widely used solution today, simply because it is the only way to stream video to the iPad and iPhone. It uses an M3U8 (like Winamp) manifest format and MPEG TS fragments with H264 video and AAC audio. Each fragment is a separate file on the webserver, which makes this solution both easy to use and difficult to scale.
- Microsoft's Smooth Streaming is the most mature adaptive streaming solution, with support for separate audio and text tracks. It uses an XML manifest format and fragmented MP4 files with H264 video and AAC audio. Video streams are stored as a single file on the server, but require specific server support (e.g. in IIS) for playback.
- Adobe's Dynamic HTTP Streaming is very similar to Smooth Streaming. It supports XML manifests and fragmented MP4 using H264 video and AAC audio. Video streams are stored as a single file on the server; regular Apache with Adobe's HTTP streaming module installed.

Since these three adaptive streaming solutions are so similar, third-party software exists to simultaneously stream to all three formats. Examples are the Wowza Media Server and the Unified Streaming Platform.

Even more promising are the standardization efforts around MPEG's DASH (Dynamic Adaptive Streaming for HTTP) and W3C's HTML5 <video>. DASH aims to become the single umbrella solution for HTTP adaptive streaming.

Sadly, none of these solutions can work with regular MP4 files. At the very least, your existing MP4 files need to be transmuxed from regular MP4 (1 metadata box, 1 sample box) into either fragmented MP4 (lots of metadata/sample pairs) or TS (transport stream) fragments. Apple, Microsoft and Adobe each supply a tool for this, but support for these formats in regular video editors and transcoding tools hasn't landed yet.

CHAPTER

TWO

ABOUT THE ADAPTIVE FRAMEWORK

This guide offers an introduction to the Adaptive framework; what it is, what it does and how it can be used.

2.1 Introduction

The adaptive framework is an ActionScript 3 framework for parsing, muxing and playing adaptive streams in Flash. Two implementations of the framework currently exist:

- A provider that can be loaded into the JW Player 5. This implementation is for publishers who want to do actual adaptive streaming in production.
- A chromeless player that can be controlled from javascript. This implementation is for developers who want to experiment with adaptive streaming, or those who what to build a fully customized interface.

Both the JW Player provider and the chromeless player need only one option to get started: the URL to an adaptive stream. Under the premise of *It Just Works*, no other options for tweaking the framework are available (yet).

Note: The adaptive framework requires a client to have at least Flash Player version 10.1 installed. This is the version that introduced the ability to load binary data into the video decoding pipeline. With an install base of about 70%, this version is not yet ubiquitous.

2.2 Solutions support

In a nutshell, support for the various adaptive streaming solutions is as follows:

`Apple HTTP Live Streaming`
The framework supports Apple HTTP Live streaming for both Live and on-demand. Streams created by Apple's *mediafilesegmenter* as well as those from the USP and Wowza servers work.

`Microsoft Smooth Streaming`
The framework currently does not support Microsoft Smooth Streaming. However, this is coming. Actionscript developers will find both the manifest parsing and MP4 fragment muxing to be largely implemented. At first, only on-demand streams featuring a single video and audio track (but multiple video qualities) will be supported.

`Adobe Zeri (HTTP Dynamic Streaming)`
The framework currently does not support Adobe Zeri (HTTP Dynamic Streaming). Given the incompatibility of this format (can only be played in Flash), supporting it is not a priority at this point. Code commits or valid reasons for doing otherwise are of course welcome.

`MPEG DASH`
The framework currently does not support MPEG DASH. Given the chances this might become the umbrella format for HTTP adaptive streaming, support will probably come.

2.3 Buffering Heuristics

When a file is loaded, the framework first loads and parses the manifest file(s). Next, a number of fragments is loaded sequentially, until a 12 second buffer is filled. The player will at all times try to maintain this buffer, loading an additional fragment if the buffer drops below 100%.

If the buffer drops below 25%, playback will pause and the framework will enter a *BUFFERING* state. If the buffer fills beyond 25%, playback will resume and the framework will enter a *PLAYING* state.

If the framework receives a seek command, the existing buffer is purged. Next, the framework acts exactly the same as on first load: the buffer is filled to 100% seconds, and playback starts when 25% is hit. This implies cached fragments will never be re-used. The framework will always return to the webserver for fragments it might have previously loaded. After all, the environment might have been improved, and the user might now be able to see an HD version of a fragment he previously saw in 240p.

The first fragment that's loaded upon the first start is always of the lowest quality level, so the stream will start fast. Switching heuristics define which subsequent fragments are loaded.

2.4 Switching heuristics

With the first fragment loaded, the framework's Loader component calculates bandwidth for the first time. It does so by dividing the fragment filesize by the duration of the download. This calculation is repeated after every fragment load. Deciding which fragment to load next is done according to the following decisions:

1. The player loads the highest quality fragment whose bitrate does not exceed 67% of the available bandwidth and whose width does not exceed 150% of the current display size.
2. The player will only switch 1 level up at a give time. So if, at a certain point, rule 1 implies the player should switch from level 3 to level 7, it will switch to level 4. Downward switching is not constrained to one level at a time.
3. The player will not take into account levels that are disabled.

Levels can be disabled because they cannot be played back at all (e.g. AAC-only), or because they led to too high a frame *droprate*. This droprate (frames dropped per second) is calculated after each fragment playout. The Loader disables levels according to the following decisions:

1. If the droprate for a fragment exceeded 25% of the framerate, all higher quality levels are disabled for 60 seconds.
2. If the droprate for a fragment exceeded 50% of the framerate, the current quality level is disabled for 60 seconds.
3. The lowest available quality level is never disabled.

Note: This 60 seconds timeout ensures the framework will not rule out a quality level entirely, only because the movie contained a fast motion shot - or the viewer was at that time checking his email.

2.5 Errors

The framework is currently quite punitive towards playback errors. These errors can occur on either the network level or the parsing / muxing level.

If an error is encountered, the framework resets and throws an error. The error is printed in the JW Player display, or broadcasted through the *onError* event in the chromeless player.

CHAPTER

THREE

ENCODING AN ADAPTIVE STREAM

This guide explains how to build an Apple HTTP Live Stream (HLS) using the *HandBrake* and *mediafilesegmenter* tools.

- HandBrake is a free desktop tool for transcoding video files to MP4. It uses the excellent *x264* encoder for H264 video encoding.
- mediafilesegmenter is a commandline tool (available only to official iOS developers) for segmenting (chopping up) an MP4 file into small TS fragments.

Streams encoded according to this guide will be compatible with iOS 3.0 (iPhone/iPad) and the framework.

3.1 Transcoding

In theory, any transcoding tool with support for H264/AAC in MP4 can be used. This guide uses Handbrake because it is free and easy to use, while at the same time capable of setting advanced encoding parameters.

3.1.1 General

When transcoding to HLS (or another adaptive streaming format), there's a couple of deviations from regular transcoding *best practices*:

`Keyframe intervals`
Because every fragment of a stream should be playeable by itself, it needs to start with a keyframe. Therefore, a fixed keyframe interval is needed. A keyframe interval of 2 seconds is recommended.

`Profiles and levels`
Since HLS streams should be playeable on mobile phones, not all bells and whizzles from the H264 format can be used. The iPhone 3GS supports H264 Baseline level 3.1.

`Variable bitrates`
Variable bitrates are possible, but the variation should be small (i.e. within a 20% range). Apple's *streamvalidator* tool will flag fragments that deviate more than 10% from the bitrate set in the manifest.

These constraints seem suboptimal. Do realize the advantages of adaptive streaming over other streaming systems are so vast these constrains are of no concern in the big picture.

3.1.2 Levels

Additionally, adaptive streaming implies you transcode your videos multiple times, into multiple quality levels. Generally, 4 to 8 quality levels, from 100 kbps to ~2mbps, are used. Here's a minimal example with four quality levels:

- H264 Baseline video @ 96 kbps, 10 fps, 160x90px. AAC HE audio @ 32 kbps, 22.05 kHz, stereo.
- H264 Baseline video @ 256 kbps, 30 fps, 320x180px. AAC HE audio @ 64 kbps, 44.1 kHz, stereo.
- H264 Baseline video @ 768 kbps, 30 fps, 640x360px. AAC HE audio @ 64 kbps, 44.1 kHz, stereo.
- H264 Baseline video @ 1536 kbps, 30 fps, 960x540px. AAC HE audio @ 64 kbps, 44.1 kHz, stereo.

Note: The first quality level will look bad, but it is intended for cellular (2G/3G) playback.

When adding quality levels, think about inserting a 512 kbps in between the 256 and 768 ones, and a 1024 kbps one in between the 768 and 1536 ones. An additional 720p quality level (1280x720px, ~2mbps) could be amended to the list.

There's no real need to go beyond 64kbps audio; 5.1 surround isn't supported in HLS - yet.

3.1.3 HandBrake

Since handbrake only supports one output video per queue entry, you have to create an entry for each quality level:

- Use the *Picture settings* to scale down your video to the desired resolution.
- In the *Video* panel, use x264 for encoding with the *average bitrate* set to your target bitrate.
- In the *Audio* panel, use CoreAudio AAC with your target mixdown (stereo), bitrate and samplerate.
- In the *Advanced* panel, use the following x264 encoding settings. They will disable settings that are Main-profile only (such as CABAC and B-frames), as well as keyframe insertion on scene changes:

```
ref=2:bframes=0:subq=6:mixed-refs=0:8x8dct=0:cabac=0:scenecut=0:min-keyint=60:keyint=60
```

Note the *keyint* settings in this line is the keyframe interval: 2 seconds for a video with 30 fps. Change the settings if your video has a different framerate.

3.2 Segmenting

Next step is segmentation of the stream. This is the process of chopping up the MP4 video into small (e.g. 2-second) fragments. In the case of Apple HLS, these segments have to be transmuxed from the MP4 format into the TS (transport stream) format as well.

The *mediafilesegmenter* tool from Apple will both do the segmenting, the transmuxing into TS and the creating of an M3U8 playlist. After installing the tool (MAC and only available to iOS developers), run the following command:

```
mediafilesegmenter -t 2 -f 300/ bunny-300.mp4
```

This command will fragment the video *bunny-300.mp4* into two-second chunks, placing those chunks and the resulting M3U8 playlist into the folder *300*. Repeat the process for each stream you have.

Note: Next to Apple's *mediafilesegmenter*, an open-source *Segmenter* tool is available. We haven't tested it yet.

3.3 Creating the manifest

The final step is creating an overall manifest to bind together the different quality streams. Such a manifest is a straightforward text document that can be built using any text editor. Here's an example:

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1600000,RESOLUTION=960x540
960/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=832000,RESOLUTION=640x360
640/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=320000,RESOLUTION=320x180
320/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=128000,RESOLUTION=160x90
160/prog_index.m3u8
```

This file (e.g. *manifest.m3u8*) tells the player which quality levels are available and where the indexes for these levels can be found.

Always start with preferred level (highest quality) first, since players generally go down the list until they find a stream they can play (although the framework does a re-sort on bitrate).

The *RESOLUTION* setting is very useful for the Adaptive provider. When set, the provider will take the resolution of each quality level into account for switching heuristics. If, for example, a client has plenty of bandwidth but only a videoplayer size of 300px wide, the 320x180 stream will be used. When the client resizes the viewport (e.g. when switching to fullscreen), the player automatically detects the change and moves to a higher quality level. In short, this setting will help constrain bandwidth usage without affecting perceived quality.

The HLS format defines another stream info parameter called *CODECS*. It can be used to list the audio/video codecs used, e.g. in case one or more quality levels are audio-only, or H264 Main or High profile.

Note: Both iDevices and the Adaptive framework can play HLS with a single quality level. In such case, the additional manifest file is not needed.

CHAPTER

FOUR

EMBEDDING AN ADAPTIVE STREAM

This guide explains how to embed an adaptive stream into your website, using the JW Player. The resulting player will playback the adaptive stream in browsers that support Flash and on iDevices. For browsers that support neither, a download fallback is displayed.

4.1 Preparations

If you haven't *encoded* an adaptive stream yet, now would be the time. In addition to the adaptive stream, we'll use a plain MP4 encode of the video, for the download fallback. This MP4 version can e.g. be 320x180 pixels (the second quality level in the encoding guide).

A recent version of the JW Player (5.5+) is needed. As of this version, it is possible to set specific video files to load into the specific modes (Flash, HTML5, Download) of the player.

Last, you need the adaptive provider (*adaptive.swf*), since adaptive streaming support is not built into the core JW Player yet.

Copy all these files onto your webserver. A plain webserver will do, since adaptive streaming does not require a specialized streaming server (a big advantage). After uploading, please point your browser directly to one of the *.m3u8* and one of the *.ts* files. When they display or download, all is fine. If they 404, you should add mimetypes for supporting those formats to your webserver configuration. Please refer to your webserver's documentation for more help.

Note: If the player and the stream are hosted at different domains, the domain hosting the adaptive stream **must contain a crossdomain.xml file in its wwwroot**. If not, The Flash plugin will be denied access to the manifest for crossdomain security reasons.

4.2 The embed code

With all the files copied onto your webserver, it's time to craft the embed code. First, make sure you load the JW Player library, e.g. somewhere in the head of your site:

```
<script type="text/javascript" src="/assets/jwplayer.js"></script>
```

Next, insert a <div> on your page at the location you want the player to appear:

```
<div id="container">The player will popup here.</div>
```

Finally, call the player setup script with all the appropriate options. This will ensure the contents of your <div> will be replaced by the JW Player:

```
<script type="text/javascript">
jwplayer("container").setup({
   height: 360,
   modes: [
     { type:'flash',
       src:'/assets/player.swf',
       provider:'/assets/adaptive.swf',
       file:'/videos/bbb.manifest.m3u8' },
     { type:'html5',
       file:'/videos/bbb.manifest.m3u8' },
     { type:'download',
       file:'/videos/bbb/fallback.mp4' }
  ],
  width: 640
});
</script>
```

The *height* and *width* of the player should be straightforward. The main logic resides in the *modes* block:

- For *flash*, the location to both the player and the adaptive provider is set. Additionally, the manifest file of the video to play is set.
- For *html5*, only the manifest file of the video to play is needed.
- For the *download* mode, the location of the fallback is provided.

This code will first check if a browser supports Flash, loading the player, provider and M3U8 manifest. Next, it will check if a browser supports HTML5. Since all desktop browsers support Flash, this will only leave the iOS browsers. Last, for all devices that support neither Flash nor HTML5 (e.g. Android < 2.2 or WinPho7), the download fallback is provided.

4.3 Live streaming

Live adaptive streaming can be done with e.g. the Wowza Media Server 2.0+. The server will take care of generating the TS fragments and manifest files, so you only need to upload the JW Player assets and provider.

The embed code setup is more or less the same, with the exception there's no download fallback. Instead, you could place a more descriptive message in the <div>, since the player will not overwrite the contents of this <div> if it cannot display anything. Example:

```
<div id="container">This live stream can only be watched in Flash or iOS</div>

<script type="text/javascript">
jwplayer("container").setup({
   height: 360,
   modes: [
     { type:'flash',
       src:'/assets/player.swf',
       provider:'/assets/adaptive.swf',
       file:'http://example.com/live/manifest.m3u8' },
     { type:'html5',
       file:'http://example.com/live/manifest.m3u8' }
  ],
  width: 640
});
</script>
```

CHAPTER

FIVE

IMPLEMENTING THE FRAMEWORK

This guide is for developers. It gives an overview of the framework and the chromeless player.

5.1 Structure

The framework consists of three major components:

- The *Manifest*, which (re)loads the adaptive streaming manifest and exposes its information.
- The *Loader*, which loads the video fragments. It also keeps track of QOS metrics to decide which quality level to load.
- The *Buffer*, which manages playback and the playback buffer. If the buffer underruns, it requests new fragments from the *Loader*.

A schematic overview of how these three components work together can be seen in the separate **cheatsheet.pdf** file.

5.2 API

The three core components are wrapped by an *API*, which exposes a number of variables, functions and events to the outside world. The API can be used to e.g. *play*, *pause* and *stop* an adaptive stream, and to retrieve the list of *quality levels* or *QOS metrics*.

Actionscript developers interested in playing adaptive streams in their application need only import the following three classes in their application:

`com.longtailvideo.adaptive.Adaptive`
    The class that implement all API calls. Simply instantiate it and use the *getVideo()* call to get the video container to place on your stage.

`com.longtailvideo.adaptive.AdaptiveEvent`
    This class defines all event types fired by the API, plus their possible parameter.

`com.longtailvideo.adaptive.AdaptiveState`
    This class defines the four playback states of the framework (idle, buffering, paused and playing).

A list of available API calls can be found in the separate **cheatsheet.pdf** file. See the source code of the JW Player provider for an actionscript implementation of the framework.

5.3 Chromeless player

The chromeless player can be used by javascript developers to provide a customized interface. It does three things:

- Displaying the video (With the correct aspect ratio).
- Forwarding all API calls to javascript.
- Pinging an *onAdaptiveReady(player)* function to signal javascript the player is initialized.

A full overview of all available API calls can be found in the separate **cheatsheet.pdf** file. All getters and setters behave exactly the same as in actionscript. Events can be retrieved by adding listeners to the player, **after** the ready call was fired. Please note the string representations of the functions in below example:

```
function onAdaptiveReady(player) {
    player.onComplete("completeHandler");
    player.onSwitch("switchHandler");
    player.play("http://example.com/manifest.m3u8");
};
function completeHandler() {
    alert("Video completed");
};
function switchHandler(level) {
    alert("Video switched to quality level "+level);
};
```

A list of available API calls can be found in the separate **cheatsheet.pdf** file. See the source code of the framework's test page for a javascript implementation of the chromeless player.

5.4 Contributing

As always, contributions and feedback to the framework are very welcome. We are especially interested in:

- Commits that accelerate the implementation of Smooth Streaming, Adobe Zeri and MPEG DASH.
- Feedback on support with/for the various transcoding and streaming tools out there.
- Actionscript or javascript implementations for third-party players.

Please direct feedback through the player development section of our support forum.

CHAPTER

SIX

MANIFEST & FRAGMENT SUPPORT

This guide elaborates on the framework's support for the various adaptive streaming solutions.

6.1 Apple HTTP Live Streaming

- Both single playlists and a manifest + playlists are supported. Both live *sliding window* and on-demand streaming are supported.
- AES decryption is not supported.
- In the case of a live event, viewers are allowed to pause/seek within the time range of the playlist at any given time. So if, for example, a live playlist contains three 10-second segments and the live head is at 04:45, the user can seek back to 04:15, or pause for 30 seconds. When seeking back beyond, or pausing longer than that timeframe, the framework will resume playback from *live head - 30*.
- The *RESOLUTION* parameter is required in order for the framework to constrain level selection to screen dimensions. If omitted, the framework might load a huge video into a small window if bandwidth allows so.
- Audio-only quality levels are not supported, but filtered if available. The framework filters based upon the value of the *CODECS* parameter. Alternatively, the framework filters out the audio levels by looking at fragment extensions; *.aac* is ignored.
- The framework follows the TS conventions as laid out by the HLS IETF draft: one PMT per TS, containing one H264 (Baseline, Main) and one AAC (Main, HE, LC) elementary stream.
- TS fragments should each start with a PES packet header for both the audio and video stream. The stream will die if this isn't the case.
- TS fragments should each start with a PES containing SPS/PPS data and a keyframe slice (NAL unit 5). Fragments starting without SPS/PPS and/or with an interframe (NAL unit 1) will work for continous playback, but will break the player if the fragment is the first fragment after a seek or quality level switch.
- The framework supports the *optimize* option of Apple's *mediafilesegmenter*. This option slices up ADTS frames or NAL units into multiple PES packets in order to keep overhead low (removing adaptation field filler data). As stated above though, this is **not** supported across TS fragments.
- B-frames should be supported (the player respects composition times), but haven't been tested yet. Likewise, H264 High profile should also be supported (not tested yet).

6.2 Microsoft Smooth Streaming

Some early info on Smooth Streaming support:

- AVCC and ADIF data are constructed from the manifest. This means the manifest must contain *CodecPrivate-Data* for video and *Channels* and *SamplingRate* for audio.
- Only one audio track (the first one in the manifest) is currently supported. Text tracks are currently not supported.

6.3 Adobe Zeri

Adobe Zeri (HTTP Dynamic Streaming) is currently not supported.

6.4 MPEG DASH

MPEG DASH is currently not supported.