RGB Networks

Comparing Adaptive HTTP Streaming Technologies

A Comparison of Apple's HTTP Live Streaming (HLS), Microsoft's Silverlight Smooth Streaming (MSS) and Adobe's HTTP Dynamic Streaming (HDS)

1. Introduction

Watching video online or on the go on a tablet or mobile phone is no longer a novelty, nor is streaming Internet-delivered content to the TV in your living room. Driven by the boom in video-enabled devices, including PCs, smartphones, and Internet-enabled set-top boxes and televisions, consumers have rapidly moved through the early-adopter phase of TV Everywhere to the stage where a growing mass of consumers expect that any media should be available on any device over any network connection, delivered at the same high quality they've come to expect from traditional television services. This explosion of multiscreen IP video – whether regarded as disruption for traditional pay-TV providers or an opportunity to expand their services – is definitely here to stay.

While tremendous advancements in core and last mile bandwidth have been achieved in the last decade – primarily driven by Web-based data consumption – video traffic represents a leap in bandwidth requirements. This, coupled with the fact that the Internet at large is not a managed quality-of-service (QoS) environment, requires that new methods of video transport be considered to provide the same quality of video experience across any device and network that consumers have come to expect from managed TV delivery networks.

The evolution of video delivery transport has led to a new set of de facto standard adaptive delivery protocols from Apple, Microsoft and Adobe that are now positioned for broad adoption. Consequently, networks must now be equipped with servers that can take high-quality video content from live streams or file sources and 'package' it for transport to devices ready to accept these new delivery protocols.

This paper gives a technical comparison of the three main HTTP adaptive streaming technologies available today: Apple's HTTP Live Streaming (HLS), Microsoft Silverlight Smooth Streaming (MSS) and Adobe's HTTP Dynamic Streaming (HDS), also previously referred to as 'Zeri.' The paper is divided into three main parts: (1) it begins by giving an overview of adaptive HTTP streaming, discussing delivery architectures, highlighting its strengths and weaknesses, and discussing live and video-on-demand (VoD) delivery; (2) it then delves into each technology, explaining how they work and highlighting how each technology is different from the others; (3) finally, it looks at specific features and describes how they are implemented or deployed. This last section focuses on:

- Delivery of multiple audio channels
- Encryption and DRM
- Closed captions / subtitling
- Ability to insert ads
- Custom VOD playlists
- Trick modes (fast-forward/rewind, pause)
- Fast channel change

- Failover due to upstream issues
- Stream latency
- Ability to send other data to the client, including manifest compression

2. Adaptive HTTP Video Delivery

Background

In ‘traditional’ IP streaming, a video server sends video to a client at a fixed bandwidth. The client and server must have synchronized states: e.g. ‘video is stopped,’ ‘video is playing,’ ‘video is paused,’ etc. Traditional video streaming is typically delivered over UDP, a connectionless protocol in which packet losses result in poor quality-of-experience (QoE) and which has difficulty passing through firewalls in home routers. Traditional streaming can adapt to changes in network bandwidth, but only with complex synchronization between the server and client, and as a result, such adaptive protocols were never widely adopted.

In comparison, in adaptive HTTP streaming the source video, whether a file or a live stream, is encoded into file segments – sometimes referred to as ‘chunks’ or ‘segments’ – using a desired format, which includes a container, video codec, audio codec, encryption protocol, etc. Segments typically represent two to ten seconds of video. The stream is broken into segments at video Group of Pictures (GOP) boundaries that begin with an IDR frame (a frame that can be independently decoded without dependencies on other frames), giving each segment independence from previous and successive segments. The segments are subsequently hosted on a regular HTTP server. Each sequence of segments is called a profile. Profiles may differ in bitrate, resolution, codec or codec profile/level.

Clients play the stream by requesting segments in a profile from a Web server, downloading them via HTTP. As the segments are downloaded, the client plays back the segments in the order requested. Since the segments are sliced along GOP boundaries with no gaps between, video playback is seamless – even though it is actually just a collection of independent file downloads via a sequence of HTTP GET requests.

Adaptive delivery enables a client to ‘adapt’ to fluctuating network conditions by selecting video segments from different profiles. The client can easily compute the available network bandwidth by comparing the download time of a segment with its size. If the client has a list of available profile bitrates (or resolutions or codecs), it can determine if it must change to a lower bitrate/resolution profile or whether the available bandwidth allows it to download segments from a higher bitrate/resolution profile. This list of available profiles is called a manifest or playlist. The client’s bandwidth calculation is repeated at every chunk download, and so the client can adapt to changing network bandwidth or other conditions every few seconds. Aside from network bandwidth, conditions that may affect the client’s choice of profile may include local CPU load or the client’s ability to play back a specific codec or resolution.

This delivery model works for both live- and file-based sources. In either case, a manifest file is provided to the client. The manifest lists the bitrates (and potentially other data) associated with the available stream profiles and the client uses it to determine how to download the chunks; that is, what URL to use to fetch chunks from specific profiles. In the case of an on-demand file request, the manifest contains information on every chunk in the content. In the case of live streaming, this isn't possible. HLS and HDS deliver a 'rolling window' manifest data that contains references to the last few available chunks, as shown in Figure 1. The client must update its manifest repeatedly in order to know about the most recently available chunks. MSS delivers information in each chunk that lets the client access subsequent chunks, so no rolling window-type manifest is needed.

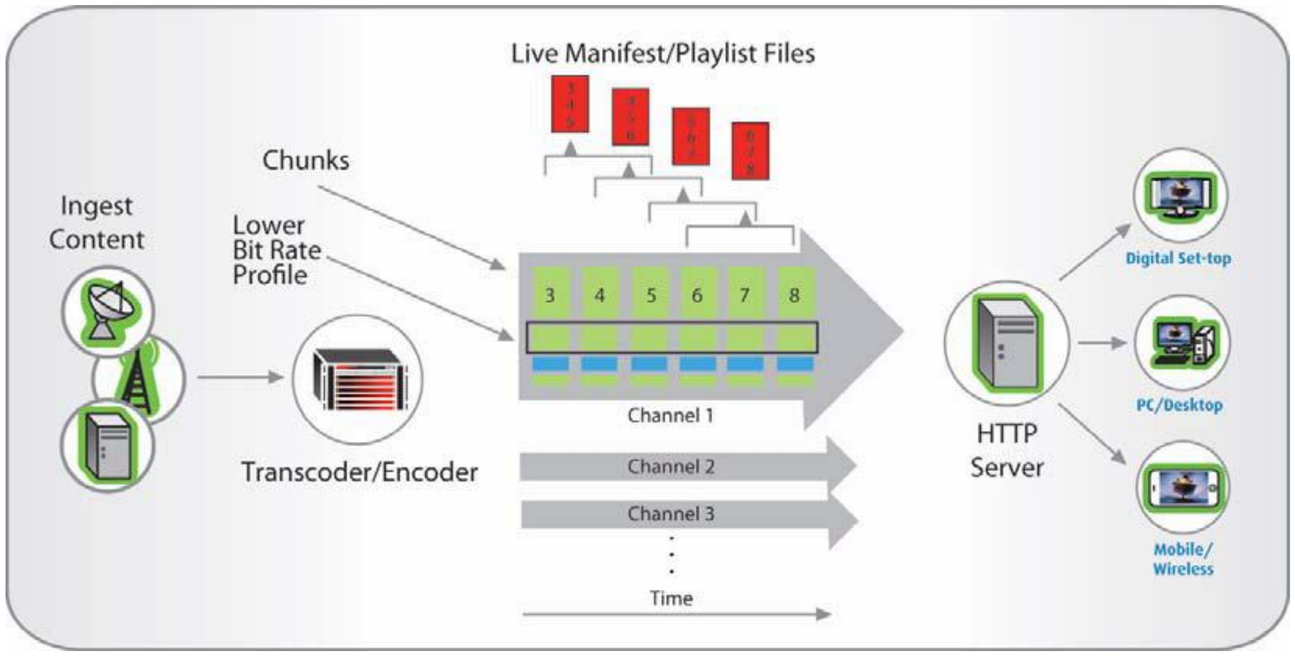

Figure 1. The content delivery chain for a live adaptive HTTP stream. The client downloads the 'rolling window' manifest files that refer to the latest available chunks. The client then uses these references to download chunks and play them back sequentially. In the figure, the first manifest refers to chunks 3, 4, and 5, which are available in multiple bitrates. As new chunks become available the playlist is updated to reference the latest available chunks.

Adaptive HTTP streaming has a number of advantages over traditional streaming:

- Lower infrastructure costs for content providers by eliminating specialty streaming servers in lieu of generic HTTP caches/proxies (that may already be in place for HTTP data delivery);
- Content delivery is dynamically adapted to the weakest link in the end-to-end delivery chain, including highly varying last mile conditions;
- Subscribers no longer need to statically select a bitrate on their own, as the client can now perform that function dynamically and automatically;
- Subscribers enjoy fast start-up and seek times as playback control functions can be initiated via the lowest bitrate chunks and subsequently ratcheted up to a higher bitrates;
- Annoying user experience shortcomings, including long initial buffer time, disconnects and playback start/stop are virtually eliminated;

- The client can control bitrate switching – with no intelligence in the server – taking into account CPU load, available bandwidth, resolution, codec and other local conditions;
- HTTP delivery works through firewalls and NAT.

Adaptive HTTP streaming also has some consequences:

- The clients must buffer a few chunks to make sure they don't starve their input buffers, which increases the end-to-end latency of live streams;
- HTTP is based on TCP; when packet loss is low, TCP recovers well and this means that video playback has no artifacts caused by missing data. However, when packet loss rises, TCP can fail completely. Thus, overall, clients will typically have good quality playback or playback that stops completely, as opposed to quality that degrades proportionally to the amount of packet loss in the delivery network. The internet is generally reliable enough so that the benefit of completely clean video at low packet drop rates (with TCP) supersedes the value of some, but very poor quality, video at high packet drop rates (with UDP).

2.1 Video Delivery Components

At a high level, the components in an adaptive HTTP streaming data flow consist of an encoder or transcoder, a packager (also called a segmenter or a fragmenter), and a content delivery network (CDN). This section discusses the features of these components (see Figure 2) that are related to adaptive streaming.

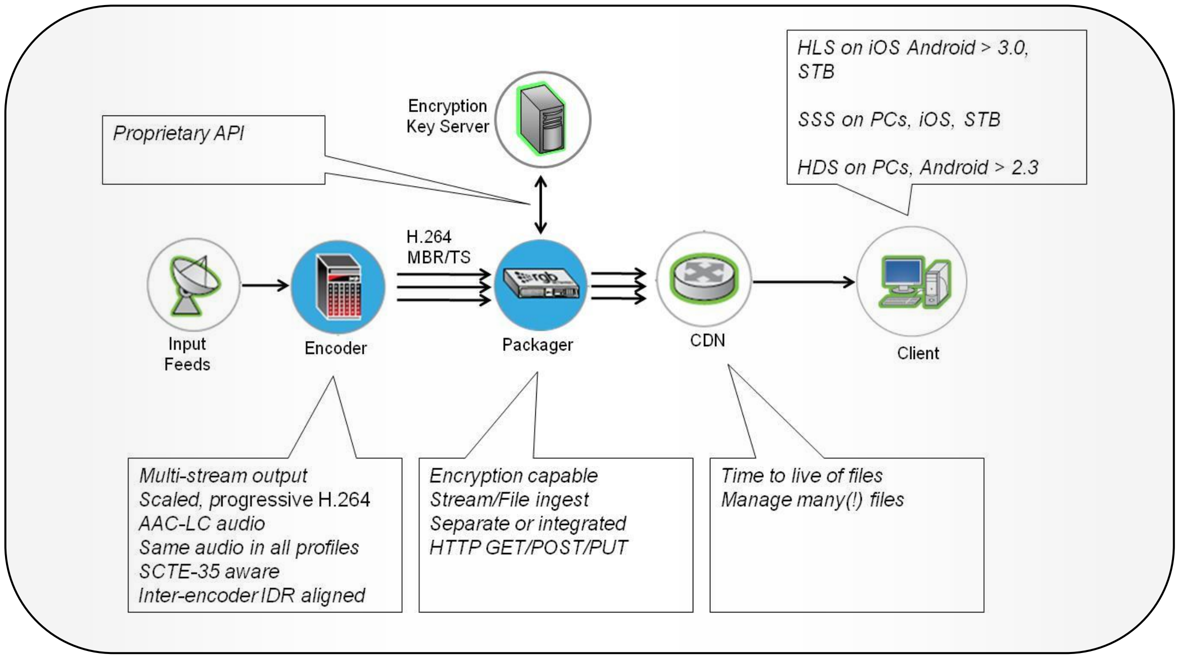

Figure 2. The components of an HTTP streaming system, with some highlighted features.

The Encoder/Transcoder

The transcoder (or encoder, if the input is not already encoded in another format) is responsible for ingesting the content and preparing it for segmentation. The transcoder must process the video in the following way:

- The output video must be in progressive format. The transcoder must therefore de-interlace the input.
- The output video must be scaled to resolutions suitable for the client device.
- The different output profiles must be IDR-aligned so that the client playback of the chunks created from each profile is continuous and smooth.
- Audio must be transcoded into AAC audio, the codec used by HLS, HDS, and MSS.
- The same encoded audio stream needs to be streamed on all the output video profiles; this avoids clicking artifacts during client-side profile changes.
- If SCTE 35 is used for ad insertion, it is desirable to have the transcoder add IDR frames at the ad insertion points so that the video is ready for ad insertion. It is then possible to align the chunk boundaries with the ad insertion points, so that ad insertion can be done more easily (via substitution of chunks) as compared to the traditional stream splicing.
- A desirable fault tolerance mechanism allows two different transcoders that ingest the same input to create identically IDR-aligned output. This can be used to create a redundant backup of encoded content in such a way that any failure of the primary transcoder is seamlessly backed up by the secondary transcoder.

Because the quality of experience of the client depends on having a number of different profiles, it is desirable to have an encoder that can output a large number of different profiles for each input. Deployments may use anywhere from 4 to 16 different output profiles for each input, with more profiles resulting in more supported devices and a better user experience. The table below shows a typical use case for the different output profiles:

Width	Height	Video Bitrate
1280	720	3 Mbits/s
960	540	1.5 MBits/s
864	486	1.25 MBits/s
640	360	1.0 MBits/s
640	360	750 Kbits/s
416	240	500 Kbits/s
320	180	350 Kbits/s
320	180	150 Kbits/s

The Packager

The packager is the component that takes the output of the transcoder and packages the video for a specific delivery protocol. The packager should have the following features:

- Encryption capability – the packager should be able to encrypt the outgoing chunks in a format compatible with the delivery protocol.
- Integration with third party key management systems – the packager should be able to receive encryption keys from a third party key management server that is also used to manage and distribute the keys to the clients.
- Packagers may ingest live streams or files, depending on whether the workflow is live or on-demand.
- Packagers should support multiple ways to deliver the chunks, either via HTTP pull or by pushing the chunks via a network share or HTTP PUT / POST.

The CDN

The content delivery network (CDN) needed for HTTP streaming is not specialized – it is HTTP-based and doesn't require any special streaming servers. For live delivery, it is beneficial to tune the CDN to age out older chunks rapidly, as there is no need to keep them around long. The actual duration depends on the duration of the chunks and latency in the client, but a minute is normally sufficient.

It is important to note that the number of chunks can be very large. For example, a day's worth of 2-second chunks delivered in 10 different profiles for 100 different channels creates 43 million files! Thus the CDN must be able to cope with a large number of files as well.

The Client

HLS is available on iOS devices, but availability on Windows is via third party products that are not always complete or robust. Interestingly, however, Android 3.0 supports HLS natively, though some features aren't well supported (e.g. stream discontinuity indications). MSS on a PC depends on a Silverlight runtime client which must be installed. But there are native Smooth Streaming clients developed for multiple devices, including iOS tablets, phones and iPods. HDS is native to flash 10.1 and later, and comes with the Flash plug-in on PCs, as well as on Android 2.3 and later devices.

Workflow Architecture

It is valuable to have an architecture that allows the transcoder and packager to be separate. This has the advantage that the input video can be transcoded just once at the core, delivered over a core network to the edge, and packaged into multiple formats at the edge of the network. Without this separation, all the final delivered formats must be delivered over the core network, unnecessarily increasing its bandwidth utilization. This is shown in Figure 3. Note, however, that packet loss on the core network would result in unrecoverable segment loses.

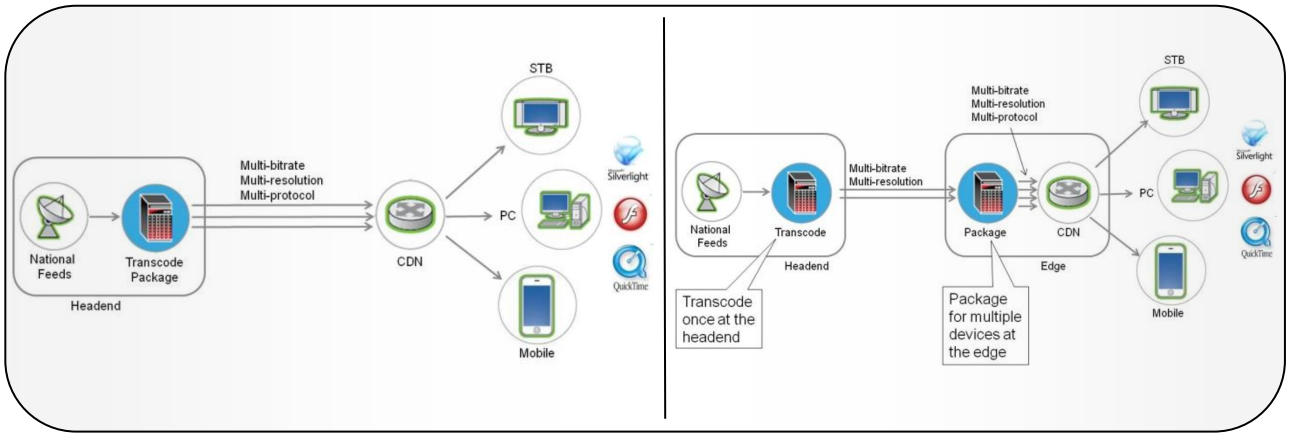

Figure 3. Integrated and remote segmentation of streams: When multiple formats are used, segmenting closer to the edge of the network (shown on the right) can save core bandwidth, as streams only need to be delivered once and can be packaged into multiple delivery formats at the edge. However, if the core network is susceptible to packet loss, segmenting at the core ensures that segments will always be delivered to the CDN (shown on the left).

Other Adaptive HTTP Streaming Technologies

There are a number of other adaptive streaming technologies available also, with varying market penetration:

- Move Networks went out of business and was purchased by Echostar – their team is now integrating the technology into their home devices. Move was instrumental in popularizing adaptive HTTP streaming and has a number of patents on the technology (though chunked streaming was used before Move popularized it).
- 3GPP's Adaptation HTTP Streaming (AHS) is part of 3GPPs rel 9 specification (see [AHS]). And 3GPP rel 10 is working on a specification called DASH as well.
- The Open TV Forum has an HTTP Adaptive Streaming (HAS) specification (see [HAS]).
- MPEG Dynamic Adaptive Streaming over HTTP (DASH) is based on 3GPP’s AHS and the Open TV Forum’s HAS and is close to completion. It specifies use of either fMP4 or transport stream (TS) chunks and an XML manifest (called the media presentation description or MPD) that is repeatedly downloaded. DASH may well become the format of choice in the future, but currently, the lack of client support makes this specification interesting only in theory. DASH

does make allowances for multiple scenarios, including separate or joined streaming of audio, video and data, as well as encryption. However, its generality is as much a drawback as an advantage, since it makes clients complex to implement.

- Many DRM vendors have their own variation of these schemes.

3. Adaptive Streaming Internals

In this section we review some of the details of HLS, HDS and MSS. Each of these protocols has strengths and weaknesses, which we discuss in the following sections.

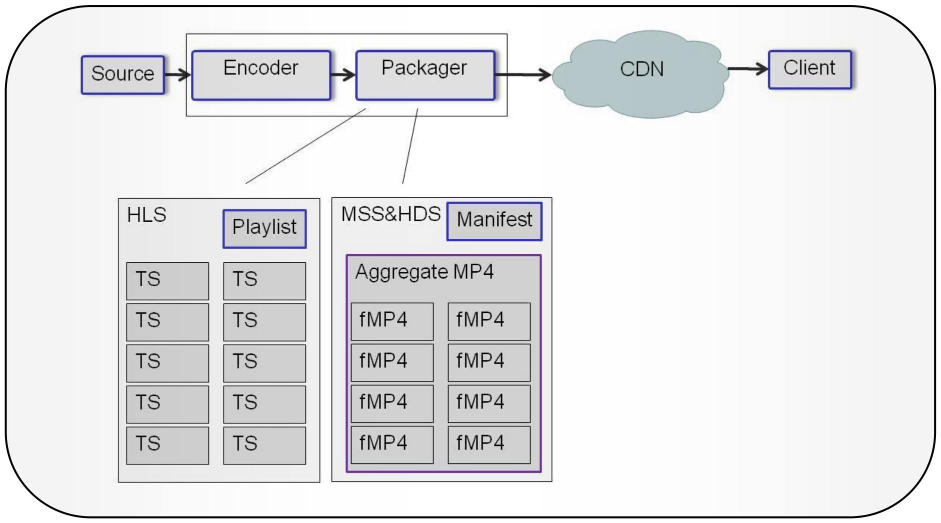

Figure 4. A comparison of HLS and MSS/HDS: The latter can create aggregate formats that can be distributed on a CDN for VoD, whereas for live video, all distribute chunks on the CDN. MSS allows audio and video to be aggregated and delivered separately, but HDS and HLS deliver these together.

Apple HTTP Live Streaming (HLS)

Ironically, unlike Microsoft and Adobe, Apple chose not to use the ISO MPEG file format – a format based on Apple’s MOV file format – in its adaptive streaming technology. Instead, HLS takes an MPEG-2 TS and segments it to a sequence of MPEG-2 TS files which encapsulate both the audio and video. These segments are placed on any HTTP server along with the playlist files. The playlist (or index) manifest file is a text file (based on Winamp’s original m3u file format) with an m3u8 extension. Full details can be found in [HLS].

HLS defines two types of playlist files: normal and variant. The normal playlist file lists URLs that point to chunks that should be played sequentially. The variant playlist files points to a collection of different normal playlist files, one for each output profile. Metadata is carried in the playlist files as comments – lines preceded by ‘#’. In the case of normal playlist files, this metadata includes a sequence number used to associate chunks from different profiles, information about the chunk duration, a directive signaling whether chunks can be cached, the location of decryption keys, the type of stream, and time information. In the case of a variant playlist the metadata includes the bitrate of the profile, its resolution, its codec, and an ID that can be used to associate different encodings of the same content. Figure 5 and Figure 6 show a sample HLS variant playlist file and normal playlist file.

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=531475
mystic_S1/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=381481
mystic_S2/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=531461
mystic_S3/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=781452
mystic_S4/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1031452
mystic_S5/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1281452
mystic_S6/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1531464
mystic_S7/mnf.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=3031464
mystic_S8/mnf.m3u8
```

Figure 5. An HLS variant playlist file showing eight output profiles with different bitrates. The URLs for the m3u8 files are relative, but could include the leading ‘http://…’. In this example, each profile’s playlist is in a separate path component of the URL.

```
#EXTM3U
#EXT-X-KEY:METHOD=NONE
#EXT-X-TARGETDURATION:11
#EXT-X-MEDIA-SEQUENCE:494

#EXT-X-KEY:METHOD=NONE
#EXTINF:10,505.ts
505.ts
#EXTINF:10,506.ts
506.ts
#EXTINF:10,507.ts
507.ts
```

Figure 6. An HLS playlist file from a live stream showing the three latest available TS chunks. The `#EXT-X-MEDIA-SEQUENCE` tag identifies the sequence number of the first chunk, 505.ts; it is used to align chunks from different profiles. Note that the chunk name carries no streaming-specific information. The `#EXT-X-TARGETDURATION:11` tag is the expected duration (10 seconds) of the chunks, though durations can vary. The `#EXT-X-KEY:METHOD=NONE` tag shows that no encryption was used in this sequence. The `#EXTINF:10` tags show the duration of each segment. As in the variant playlist file, the URLs are relative to the base URL used to fetch the playlist.

In HLS, a playlist file corresponding to a live stream must be repeatedly downloaded so that the client can know the URLs of the most recently available chunks. The playlist is downloaded every time a chunk is played, and thus, in order to minimize the number of these requests, Apple recommends a relatively long chunk duration of 10 seconds. However, the size of the playlist file is small compared with any video content, and the client maintains an open TCP connection to the server, so that this network load is not significant. Shorter chunk durations can thus be used. This allows the client to adapt bitrates more quickly. VoD playlists are distinguished from live playlists by the `#EXT-X-PLAYLIST-TYPE` and `#EXT-X-ENDLIST` tags.

HLS is the only protocol that doesn’t require chunks to start with IDR frames. It can download chunks from two profiles and switch the decoder between profiles on an IDR frame that occurs in the middle of a chunk. However, doing this incurs extra bandwidth consumption, as two chunks corresponding to the same portion of video are downloaded at the same time.

HLS Considerations
Some advantages of HLS are:

- It is a simple protocol that is easy to modify. The playlists are easily accessible and their text format lends to simple modification for applications such a re-broadcast or ad insertion.
- The use of TS files means that there is a rich ecosystem for testing and verifying file conformance.
- TS files can carry other metadata, such as SCTE 35 cues or ID3 tags (see [HLSID3]).
- HLS is native to popular iOS devices, the users of which are accustomed to paying for apps and other services. That is, HLS is more easily monetized.

Some disadvantages of HLS are:

- HLS is not supported natively on Windows OS platforms.
- TS files mux the audio, video and data together. This means that multi-language support either comes at the cost of sending all the languages in the chunks or creating duplicate chunks with each language. Similarly for data PIDs, these are either muxed together or multiple versions of chunks are needed with different data PIDs.
- There is no standard aggregate format for HLS, which means that many chunk files are created. A day's worth of one channel with eight profiles at 10-second chunk duration will consist of almost 70,000 files. Managing such a large collection of files is not convenient.

Note: iOS 5 offers features that loosen some of these limitations, but the distribution of iOS versions means that these disadvantages persist in the market today.

Microsoft Silverlight Smooth Streaming (MSS)

Silverlight Smooth Streaming delivers streams as a sequence of ISO MPEG-4 files (see [MSS] and [MP4]). These are typically pushed by an encoder to a Microsoft IIS server (using HTTP POST), which aggregates them for each profile into an 'ismv' file for video and an 'isma' file for audio. The IIS server also creates an XML manifest file that contains information about the bitrates and resolutions of the available profiles (see Figure 7). When the request for the manifest comes from a Microsoft IIS server, it has a specific format:

```
http://{serverName}/{PublishingPointPath}/{PublishingPointName}.isml/manifest
```

The `PublishingPointPath` and `PublishingPointName` are derived from the IIS configuration.

Unlike HLS, in which the URLs are given explicitly in the playlist, in MSS the manifest files contain information that allows the client to create a RESTful URL request based on timing information in the stream. For live streaming, the client doesn't need to repeatedly download a manifest – it computes the URLs for the chunks in each profile directly. The segments are extracted from the ismv and isma files and served as 'fragmented' ISO MPEG-4 (fMP4) files. MSS (optionally) separates the audio and video into separate chunks and combines them in the player. The URLs below show typical requests for video and audio. The `QualityLevel` indicates the profile and the `video=` and `audio-eng=` indicate the specific chunk requested. The `Fragments` portion of the request is given using a time stamp (usually in hundred nanosecond intervals) that the IIS server uses to extract the correct chunk from the aggregate MP4 audio and/or video files.

```
http://sourcehost/local/2/mysticSmooth.isml/QualityLevels(350000)/Fragments(video=2489452460333)
http://sourcehost/local/2/mysticSmooth.isml/QualityLevels(31466)/Fragments(audio-eng=2489450962444)
```

In the VoD case, the manifest files contain timing and sequence information for all the chunks in the content. The player uses this information to create the URL requests for the audio and video chunks.

Note that the use of IIS as the source of the manifest and fMP4 files doesn't preclude use of standard HTTP servers in the CDN. The CDN can still cache and deliver the manifest and chunks as it would any other files. More information about MSS can be found at Microsoft (see [SSTO]) and various excellent blogs of the developers of the technology (see [SSBLOG]).

MSS Considerations

Some advantages of MSS are:

- IIS creates an aggregate format for the stream, so that a small number of files can hold all the information for the complete smooth stream.
- The use of IIS brings useful analysis and logging tools, as well as the ability to deliver more MSS and HLS content directly from the IIS server.
- The recommended use of a small chunk size allows for rapid adaptation during HTTP streaming playback.
- The segregated video and audio files mean that delivery of different audio tracks involves just a manifest file change.
- The aggregate file format supports multiple data tracks that can be used to store metadata about ad insertion, subtitling, etc.

Some disadvantages of MSS are:

- The need to place an IIS server in the data flow adds an extra point of failure and complicates the network.
- On PCs, MSS requires installation of a separate Silverlight plug-in.

```
<SmoothStreamingMedia MajorVersion="2" MinorVersion="0" TimeScale="10000000" Duration="0" LookAheadFragmentCount="2"
IsLive="TRUE" DVRWindowLength="300000000">
    <StreamIndex Type="video" QualityLevels="7" TimeScale="10000000" Name="video" Chunks="14"
        Url="QualityLevels({bitrate})/Fragments(video={start time})" MaxWidth="1280" MaxHeight="720"
        DisplayWidth="1280" DisplayHeight="720">
        <QualityLevel Index="0" Bitrate="350000"
        CodecPrivateData="00000001274D401F9A6282833F3E022000007D20001D4C12800000000128EE3880" MaxWidth="320"
        MaxHeight="180" FourCC="H264" NALUnitLengthField="4"/>
        <QualityLevel Index="1" Bitrate="500000"
        CodecPrivateData="00000001274D401F9A628343F6022000007D20001D4C12800000000128EE3880" MaxWidth="416"
        MaxHeight="240" FourCC="H264" NALUnitLengthField="4"/>
        <QualityLevel Index="2" Bitrate="750000"
        CodecPrivateData="00000001274D401F9A6281405FF2E022000007D20001D4C1280000000128EE3880" MaxWidth="640"
        MaxHeight="360" FourCC="H264" NALUnitLengthField="4"/>
        <QualityLevel Index="3" Bitrate="1000000"
        CodecPrivateData="00000001274D401F9A6281405FF2E022000007D20001D4C1280000000128EE3880" MaxWidth="640"
        MaxHeight="360" FourCC="H264" NALUnitLengthField="4"/>
        <QualityLevel Index="4" Bitrate="1250000"
        CodecPrivateData="00000001274D40289A6281B07FF36022000007D20001D4C1280000000128EE3880" MaxWidth="864"
        MaxHeight="486" FourCC="H264" NALUnitLengthField="4"/>
        <QualityLevel Index="5" Bitrate="1500000"
        CodecPrivateData="00000001274D40289A6281E022FDE022000007D20001D4C1280000000128EE3880" MaxWidth="960"
        MaxHeight="540" FourCC="H264" NALUnitLengthField="4"/>
        <QualityLevel Index="6" Bitrate="3000000"
        CodecPrivateData="00000001274D40289A6280A00B76022000007D20001D4C12800000000128EE3880" MaxWidth="1280"
        MaxHeight="720" FourCC="H264" NALUnitLengthField="4"/>
        <c t="2489302977667"/>
        <c t="2489324332333"/>
        <c t="2489345687000"/>
        <c t="2489367041667"/>
        <c t="2489388396333"/>
        <c t="2489409751000"/>
        <c t="2489431105667"/>
        <c t="2489452460333"/>
        <c t="2489473815000"/>
        <c t="2489495169667"/>
        <c t="2489516524333"/>
        <c t="2489537879000"/>
        <c t="2489559233667"/>
        <c t="2489580588333" d="21354667"/>
    </StreamIndex>
    <StreamIndex Type="audio" QualityLevels="4" TimeScale="10000000" Language="eng" Name="audio_eng" Chunks="14"
        Url="QualityLevels({bitrate})/Fragments(audio_eng={start time})">
        <QualityLevel Index="0" Bitrate="31466" CodecPrivateData="1190" SamplingRate="48000" Channels="2"
        BitsPerSample="16" PacketSize="4" AudioTag="255" FourCC="AACL"/>
        <QualityLevel Index="1" Bitrate="31469" CodecPrivateData="1190" SamplingRate="48000" Channels="2"
        BitsPerSample="16" PacketSize="4" AudioTag="255" FourCC="AACL"/>
        <QualityLevel Index="2" Bitrate="31472" CodecPrivateData="1190" SamplingRate="48000" Channels="2"
        BitsPerSample="16" PacketSize="4" AudioTag="255" FourCC="AACL"/>
        <QualityLevel Index="3" Bitrate="31481" CodecPrivateData="1190" SamplingRate="48000" Channels="2"
        BitsPerSample="16" PacketSize="4" AudioTag="255" FourCC="AACL"/>
        <c t="2489301415778"/>
        <c t="2489322749111"/>
        <c t="2489344082444"/>
        <c t="2489365415778"/>
        <c t="2489386749111"/>
        <c t="2489408295778"/>
        <c t="2489429629111"/>
        <c t="2489450962444"/>
        <c t="2489472295778"/>
        <c t="2489493629111"/>
        <c t="2489514962444"/>
        <c t="2489536295778"/>
        <c t="2489557629111"/>
        <c t="2489578962444" d="21333334"/>
        </StreamIndex>
</SmoothStreamingMedia>
```

Figure 7. A sample MSS manifest file. The elements with '`t="249…"`' specify the time stamps of chunks that the server already has and can deliver. These are converted to `Fragment` timestamps in the URL requesting an fMP4 chunk. The returned chunk holds time stamps of the next chunk or two (in its UUID box), so that the client can continue to fetch chunks without having to request a new manifest.

Adobe HTTP Dynamic Streaming (HDS)

Adobe HDS was defined after both HLS and MSS and makes use of elements in each (see [HDS]). In HDS, an XML manifest file (of file type f4m) contains information about the available profiles (see Figure 8 and [F4M]). As in HLS, data that allows the client to derive the URLs of the available chunks is repeatedly downloaded by the client; in HDS this is called the bootstrap information. The bootstrap information is in a binary format and hence isn't human-readable. As in MSS, segments are encoded as fragmented MP4 files that contain both audio and video information in one file. HDS chunk requests have the form:

```
http://server_and_path/QualityModifierSeg'segment_number'-Frag'fragment_number'
```

where the segment and fragment number together define a specific chunk. As in MSS, an aggregate (f4f) file format is used to store all the chunks and extract them when a specific request is made.

```
<manifest>
    <id>USP</id>
    <startTime>2006-07-24T07:15:00+01:00</startTime>
    <duration>0</duration>
    <mimeType>video/mp4</mimeType>
    <streamType>live</streamType>
    <deliveryType>streaming</deliveryType>
    <bootstrapInfo profile="named" url="mysticHDS.bootstrap"/>
    <media url="mysticHDS-audio_eng=31492-video=3000000-" bitrate="3031" width="1280" height="720"/>
    <media url="mysticHDS-audio_eng=31492-video=1500000-" bitrate="1531" width="960" height="540"/>
    <media url="mysticHDS-audio_eng=31492-video=1250000-" bitrate="1281" width="864" height="486"/>
    <media url="mysticHDS-audio_eng=31492-video=1000000-" bitrate="1031" width="640" height="360"/>
    <media url="mysticHDS-audio_eng=31492-video=750000-" bitrate="781" width="640" height="360"/>
    <media url="mysticHDS-audio_eng=31492-video=500000-" bitrate="531" width="416" height="240"/>
    <media url="mysticHDS-audio_eng=31469-video=350000-" bitrate="381" width="320" height="180"/>
</manifest>
```

Figure 8. A sample HDS manifest file.

HDS Considerations

Some advantages of HDS are:

- The Flash client is available on multiple devices and is installed on almost every PC in the world.
- HDS is a part of Flash and can make use of Flash's environment and readily available developer base.

Some disadvantages of HSS are:

- HDS is a relative late-comer and could suffer more from stability issues.
- Adobe's Flash access roadmap is rapidly changing, making it difficult to deploy a stable ecosystem.
- Adobe holds close the details of its format. Combined with the binary format of the bootstrap file, this limits the ecosystem of partners offering compatible solutions.

4. Feature Comparison

In this section, we compare HLS, HDS and MSS usability for a variety of common usage scenarios.

Delivery of Multiple Audio Channels

In HLS, TS files can easily carry multiple audio tracks, but this isn’t necessarily a strength when it comes to adaptive HTTP streaming because it means the TS chunks are bigger, even if only one audio stream is consumed. In places where there are multiple languages, multiple audio streams included in each chunk can consume a larger portion of the bandwidth than the video. For HLS to work well, packagers have to create chunks with each video/audio language combination, increasing the storage needed for holding the chunks.

MSS stores each audio track as a separate file, so it’s easy to create chunks with any video-audio combination. The total number of files is the same as with HLS, but the ability to store the audio and video once in aggregate format makes MSS more efficient.

HDS does not offer a compelling solution to delivering different audio streams.

Encryption and DRM

HLS supports encryption of each TS file. This means that all the data in the TS file is encrypted and there is no way to extract it without access to the decryption keys. All metadata related to the stream (e.g. the location of the decryption keys) must be included in the playlist file. This works well, except that HLS does not specify a mechanism for authenticating clients to receive the decryption keys. This is considered a deployment issue. A number of vendors offer HLS-type encryption, often with their own twist which makes the final deployment incompatible with other implementations.

MSS uses Microsoft’s PlayReady, which gives a complete framework for encrypting content, managing keys, and delivering them to clients. In PlayReady, only the payload of the fMP4 file is encrypted, so the chunk can carry other metadata. Microsoft makes PlayReady code available to multiple vendors that productize it, and so a number of vendors offer PlayReady capability (in a relatively undifferentiated way).

HDS uses Adobe’s Flash Access, which has an interesting twist that simplifies interaction between the key management server and the scrambler that does the encryption. Typically, keys must be exchanged between these two components, and this exchange interface is not standardized. Each DRM vendor/scrambler vendor pair must implement this pair-wise proprietary API. With Adobe Access, no key exchange is necessary – the decryption keys are sent along with the content, but are themselves encrypted. Access to those keys is granted at run time, but no interaction between the key management system and scrambler is needed. Adobe licenses its Access code to third parties, or it may be used as part of the Flash Media Server product suite.

Closed Captions / Subtitling

HLS can decode and display closed captions (using ATSC Digital Television Standard Part 4 – MPEG-2 Video System Characteristics - A/53, Part 4:2007, see [ATCA]) included in the TS chunks as of iOS 4.3 (the implementation in iOS 4.2 is more problematic). For DVB teletext, packagers need to convert the subtitle data into ATSC format or wait for clients to support teletext data.

MSS supports data tracks that hold Time Text Markup Language (TTML), a way to specify a separate data stream with subtitle, timing and placement information (see [TTML]). For MSS, packagers need to extract subtitle information from their input and convert it into a TTML track. Microsoft's implementation of MSS client currently offers support for W3C TTML, but not for SMPTE TTML (see [SMPTE-TT]), which adds support for bitmapped subtitles, commonly used in Europe.

HDS supports data tracks that hold subtitles as DFXP file data, based on the TTML format. Clients can selectively download this data, similar to MSS, but client support requires customization and additional development.

Targeted Ad insertion

HLS is the simplest protocol to use for chunk-substitution-based ad insertion. With HLS, the playlist file can be modified to deliver different ad chunks to different clients (see Figure 9). The `EXT-X-DISCONTINUITY` tag can be used to tell the decoder to reset (e.g. because subsequent chunks may have different PID values), and only the sequence ID must be managed carefully, so that the IDs line up when returning to the main program. HDS also supports a repeated download of bootstrap data used to specify chunks, and this can be modified to create references to ad chunks – but because the bootstrap data format is binary, and the URLs are RESTful with references to chunk indexes, the process is complex.

MSS is trickier when it comes to chunk-based ad insertion for live streams. The fact that chunks contain timing information used to request the next chunk means that all ad chunks have to have identical timing to the main content chunks (or that some other method is used to reconcile the program time when returning to the program). Nevertheless, a proxy can be used to redirect RESTful URL chunk requests and serve different chunks to different clients.

Both MSS and HDS can deliver control events in separate data tracks. These can be used to trigger client behaviors using the Silverlight and Flash environments, including ad insertion behavior. This is beyond the scope of this paper, which is focused on 'in stream' insertion. One difference between MSS and HDS is that MSS defines a client-side ad insertion architecture, whereas HDS does not.

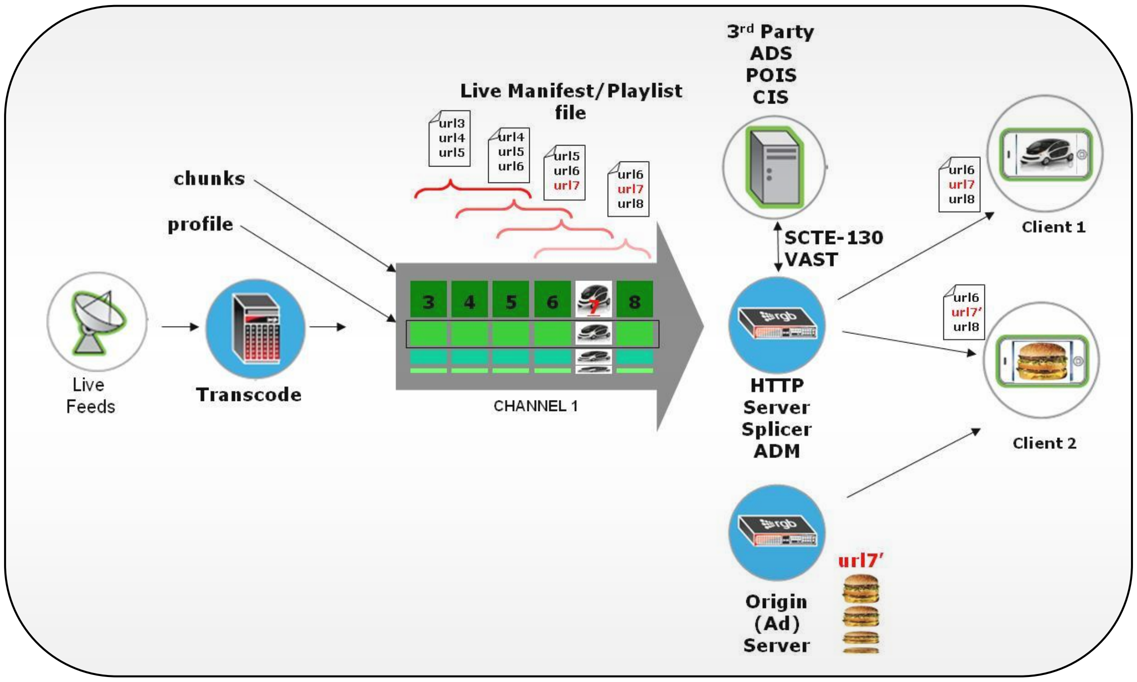

Figure 9. HLS ad insertion in which changes to the playlist file delivered to each client cause each client to make different URL requests for ads and thus receive targeted ad content.

Trick Modes (Fast-forward / Rewind)

VoD trick modes, such as fast-forward or rewind, are messy in all the protocols. None of the protocols offer native support for fast-forward or rewind. Some MSS variations have defined zoetrope images that can be embedded in a separate track. These can be used to show still images from the video sequence and allow seeking into a specific location in the video.

HDS supports a fast playback mode, but this doesn't appear to work well.

Custom VoD Playlists

It is convenient to be able to take content from multiple different assets and stitch them together to form one asset. This is readily done in HLS, where the playlist can contain URLs that reference chunks from different encodings and locations. In MSS and HDS, the RESTful URL name spaces and the references to chunks via time stamp or sequence number makes such playlists basically impossible to construct.

Fast Channel Change

Adaptive HTTP streaming can download low bitrate chunks initially, making channel ‘tune in’ times low. The duration of the chunk directly affects how fast the channel adapts to a higher bandwidth (and higher quality video). Because of this, MSS and HDS, which are tuned to work with smaller chunks, tend to work a bit better than HLS.

Failover Due to Upstream Issues

HLS manifests can list failover URLs in case content is not available. The mechanism used in the variant playlist file to specify different profiles can be used to specify failover servers, since the client (starting with iOS 3.1 and later) will attempt to change to the next profile when a profile chunk requests returns an HTTP 404 ‘file not found’ code. This is a convenient, distributed redundancy mode.

MSS utilizes a run-time client that is fully programmable. Similarly, HDS works within the Flash run-time environment. That means that the some failover capabilities can be built into the client. However, in both cases, there isn’t a built-in mechanism in the protocol to support a generic failover capability.

All the protocols will failover to a different profile if chunks/playlists in a given profile are not available. This potentially allows any of the protocols to be used in a “striping” scenario in which alternate profiles come from different encoders (as long as the encoders output IDR aligned streams), so that an encoder failure causes the client to adapt to a different, available profile.

Stream Latency

Adaptive HTTP clients buffer a number of segments. Typically one segment is currently playing, one is cached, and a third is being downloaded – so that the end-to-end latency is minimally about three segment durations long. With HLS recommended to run with 10-second chunks (though this isn’t necessary), this latency can be quite long.

Of the three protocols, only MSS has implemented a low latency mode in which sub-chunks are delivered to the client as soon as they are available. The client doesn’t need to wait for a whole chunk’s worth of stream to be available at the packager before requesting it, reducing its overall end-to-end latency.

Ability to Send Other Data to the Client (Including Manifest Compression)

HLS and HDS can send metadata to the client in their playlist and manifest files. MSS and HDS allow data tracks which can trigger client events and contain almost any kind of data. HLS allows a separate ID3 data track to be muxed into the TS chunks. This can be used to trigger client-side events.

MSS also allows manifest files to be compressed using gzip (as well as internal run-length-type compression constructs) for faster delivery.

5. Conclusion

A review of how these three formats compare is shown in the table below. In spite of MSS's better performance, the technology that will gain the most market share remains to be seen. Ultimately, it may be DASH that succeeds in the market in the long run and not any of the technologies discussed here.

Feature	HLS	MSS	HDS
Multiple audio channels		☺	
Encryption		☺	☺
Closed captions / subtitling	☺	☺	
Custom VoD playlists	☺		
ability to insert ads	☺	😐	
trick modes (fast forward / rewind)		😐	
fast channel change & Stream latency		☺	☺
Client failover	☺		
Metadata	☺	☺	☺

In any case, online and mobile viewing of premium video content is rapidly complementing the traditional TV experience, and delivery over the Internet requires new protocols to produce a high quality of experience based on device type and network congestion. Apple HLS, Microsoft Smooth Streaming and Adobe Flash HDS represent adaptive delivery protocols that enable high-quality video consumption experiences over the Internet. Content providers must now equip network delivery infrastructure with products capable of receiving standard video containers, slicing them into segments, and delivering those segments along with encryption where required. As with all new technology, the choice of delivery protocol will be made based on a combination of technical and business factors.

6. References

[HLS] HTTP Live Streaming, R. Pantos, http://tools.ietf.org/html/draft-pantos-http-live-streaming-06

[HLS1] HTTP Live Streaming, http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/HTTPLiveStreaming/_index.html

[MP4] International Organization for Standardization (2003). "MPEG-4 Part 14: MP4 file format; ISO/IEC 14496-14:2003"

[SSBLOG] http://blog.johndeutscher.com/, http://blogs.iis.net/samzhang/ , http://alexzambelli.com/blog/, http://blogs.iis.net/jboch/

[SSTO] Smooth Streaming Technical Overview, http://learn.iis.net/page.aspx/626/smooth-streaming-technical-overview/

[ATCA] ATSC Digital Television Standard Part 4 – MPEG-2 Video System Characteristics (A/53, Part 4:2007), http://www.atsc.org/cms/standards/a53/a_53-Part-4-2009.pdf

[TTML] Timed Text Markup Language, W3C Recommendation 18 November 2010, http://www.w3.org/TR/ttaf1-dfxp/

[SMPTE-TT] SMPTE Time Text format, https://www.smpte.org/sites/default/files/st2052-1-2010.pdf

[MSS] IIS Smooth Streaming Transport Protocol, http://www.iis.net/community/files/media/smoothspecs/%5BMS-SMTH%5D.pdf

[F4M] Flash Media Manifest File Format Specification, http://osmf.org/dev/osmf/specpdfs/FlashMediaManifestFileFormatSpecification.pdf

[HDS] HTTP Dynamic Streaming on the Adobe Flash Platform, http://www.adobe.com/products/httpdynamicstreaming/pdfs/httpdynamicstreaming_wp_ue.pdf

[AHS] 3GPP TS 26.234: "Transparent end-to-end packet switched streaming service (PSS); Protocols and codecs".

[HAS] OIPF Release 2 Specification - HTTP Adaptive Streaming http://www.openiptvforum.org/docs/Release2/OIPF-T1-R2-Specification-Volume-2a-HTTP-Adaptive-Streaming-V2_0-2010-09-07.pdf

[HLSID3] Timed Metadata for HTTP Live Streaming, http://developer.apple.com/library/ios/#documentation/AudioVideo/Conceptual/HTTP_Live_Streaming_Metadata_Spec/Introduction/Introduction.html

[DVB-BITMAPS] Digital Video Broadcasting (DVB); Subtitling systems, ETSI EN 300 743 V1.3.1 (2006-11), http://www.etsi.org/deliver/etsi_en/300700_300799/300743/01.03.01_60/en_300743v010301p.pdf