DASH COMPLIANT CLIENT-SERVER VIDEO HOSTING SERVICE

Abstract— **This report describes the design and implementation of a DASH compliant client-server video hosting service. The client is an Android device running Honeycomb 3.2, while the server is a Linux machine running the Apache server. This video hosting service brings out the advantages of using DASH (Dynamic Adaptive Streaming over HTTP) as a stream switching technique.**

Keywords – DASH; Client-Server Video Streaming; Android; LAMP

1. INTRODUCTION

Dynamic Adaptive Streaming over HTTP (DASH) is a stream switching technique currently under standardization [1]. It is a popular alternative to the traditional RTP/RTSP/RTCP approach to video streaming. It is expected to become an international standard in November 2011 [2].

DASH works by splitting a video stream into smaller independent pieces called streamlets, each about 10 seconds long. The Streaming client downloads the streamlets via HTTP, splices them, and plays the video continuously to the user. Initially, the server provides a playlist (also called media description file) to the client. The playlist contains information about each streamlet, and from where they can be downloaded. More importantly, the playlist can specify alternate versions of the streamlets for different bitrates. This allows the client to adaptively switch between different bitrates during the video play, by downloading the appropriate streamlet version.

The implemented video hosting service highlights the advantages of using DASH. In this project, the Android client application (running on ASUS Transformer TF101) allows a user to record a video and upload it to a central server running Apache. The video is divided into ten second streamlets, transcoded into three different bitrate qualities, and stored on the server. In addition, a playlist file is created on the server, and made publicly available. Any DASH Streaming client, such as Apple QuickTime Player, can use this playlist file to play the video stream. This report describes the design and implementation of the video hosting service.

The rest of the report is organized as follows. Section 2 describes related work, and provides motivation for this project. Section 3 discusses the design considerations of the video hosting service, and the implementation details. A comparison of this project with other streaming solutions is given in Section 4. Future work is discussed in Section 5. Finally, Section 6 concludes the report.

2. RELATED WORK AND MOTIVATION

2.1. TRADITIONAL STREAMING SOLUTIONS

Traditional streaming solutions utilize three protocols – RTP [3], RTSP [4] and RTCP [5]. RTP is a general packet format which encapsulates video and audio data, and provides services such as timestamps and sequence numbers. RTSP is a protocol for VCR type playback functions, such as start, play, pause, and stop. RTCP provides the client and server with useful statistical information regarding packet loss, jitter and Round Trip Time. While RTSP operates over TCP, RTP and RTCP typically operate over UDP.

These protocols, however, have many drawbacks. First, the server has to adaptively stream the RTP packets based on the bandwidth available to the client. The stream has to be TCP friendly, and not be too bursty. Calculating the bandwidth and corresponding appropriate transmission rate is complicated. Second, the server has to be able to provide various bitrate qualities of the video at the granularity of the RTP packet data. This is usually done using Multiple Description Coding [6], a complicated coding scheme for error resilience. Third, each proprietary media type may need a special-purpose streaming server. Fourth, the RTCP receiver report frequency decreases with larger number of receivers, making it difficult for the server to judge the client's bandwidth. Fifth, caching of videos in Content Delivery Networks (such as Akamai) is also complex, since each proprietary video stream has to be cached in a different way. Sixth, the client usually blocks UDP traffic using a firewall running on a NAT device. This makes it difficult to send RTP packets over UDP. Seventh, the client may have difficulty synchronizing the video and audio stream, since the RTP packets are sent separately, and may arrive with a lot of jitter. This is leads to the audio playback being

out-of-synch with the video (lip movement doesn't match the dialogues).

Despite these drawbacks, the RTP/RTSP/RTCP solution has benefits. The latency is very low, allowing it to be used for live streaming and video conferencing. Furthermore, the switching between video qualities can be done at a fine granularity level.

2.2. PROPRIETARY DASH STREAMING SOLUTIONS

The concept of dividing the video into streamlets was introduced by Move Networks [7]. Each streamlet is encoded into low, medium and high qualities, and downloaded individually and independently by the client over HTTP. The client can request a particular quality streamlet based on the available bandwidth, and switch qualities dynamically as the bandwidth changes. Such a streaming solution pushes the complexity from the server to the client.

Apple later adopted this solution in the QuickTime Streaming Server, called HTTP Live Streaming [9]. Microsoft also adopted this approach in its IIS Server, called Smooth Streaming [10]. Microsoft supports only the Silverlight client, unlike Apple which supports any playback client implementing DASH. Apple also included DASH support in iphone3 [8]. Other popular DASH implementations are listed in [9], which includes Android Honeycomb 3.0 as a DASH-compliant client.

A comparison of Apple's Live Streaming, Adobe's Dynamic Streaming and Microsoft's Smooth streaming is given nicely in a tabular format in [21]. It claims that Smooth Streaming has a latency of 1.5 seconds as compared to Apple's 30 second latency. But, since the comparison is given in the IIS website, the comparison may not be neutral, and may be biased towards Smooth Streaming.

All the implementations described above are proprietary. Currently, there is a standardization effort for the DASH protocol. Once this becomes an open international standard, any client/server video streaming can implement DASH without licensing issues.

2.3. DASH AS AN OPEN STANDARD

The latest draft standard of DASH is given in [1]. It is expected to become a standard in November 2011.

Using DASH has many advantages. First, the server is simplified into a normal stateless HTTP web server like apache, which is free and has no licensing costs. Second, there are no firewall issues with DASH, since it uses the standard port 80 (http). Third, DASH works well with normal Web caching, since streamlet requests can be treated the same as other web requests. Fourth, the complexity is pushed from the server to the client. This is useful, since it is the client who ultimately knows about the bandwidth available, and can switch streamlets accordingly. Moreover, clients (such as smartphones) are becoming more powerful, allowing such complexity to be implemented practically. Fourth, synchronizing audio and video is simpler – there is no jitter, since both audio and video are contained in the streamlet as related tracks. Fifth, the DASH protocol is inherently simple, especially on the server side, gaining acceptance easily as compared to more complicated approaches such as layered video. Sixth, which is more important from the point of view of an open standard, is that once a playlist file is published, *any* DASH-compliant player can play the video stream. This is in contrast to problems playing proprietary video formats, for example playing Apple video file formats in Windows Media Player.

However, DASH does have drawbacks. The switching between different bitrates can happen only at streamlet boundaries, which can be as long as 10 seconds. In RTP, the switching can occur at the granularity of Application Data Units. In addition, DASH adds latency to the video distribution, making it unsuitable for real-time video conferencing or critical operations, such as remote doctor surgical operations. This is because the server has to constantly update the playlist files, and the client has to fetch the updated playlist files repeatedly. The update and fetch operations have timing constraints, which introduces the latency. The other disadvantage is that the client is now complicated – it has to download streamlets well in advance, monitor the bandwidth, buffer and splice streamlets, among other functionalities.

2.4. PROJECT MOTIVATION

The video hosting service attempts to portray the advantages of application developers using DASH. Previously, developers had to write their own streaming client, or deal with licensing costs and issues of using proprietary video servers. Now however, by storing the video in a DASH-compliant format on the server, *any* DASH streaming client can

play the video uploaded by the user. The developer does not have to write the streaming client himself.

In this project, the videos uploaded on the server can be played by Apple's QuickTime player, Android Honeycomb 3.0, VLC player, and other DASH streaming clients. This automatically broadens the viewing audience. Any improvement in the Apple streaming client automatically benefits the developed application.

In addition, the server-side implementation is simplified, since the well-known LAMP stack is used. On the other hand, the project does not support live streaming, since the server does not update the playlist files within the timing constraints of the Apple Draft Standard [11], and the client does not upload the video while recording.

Since the DASH protocol is being used to stream videos, all the advantages described previously also apply.

3. Design and Implementation

This section describes the design of the client-server video hosting service.

The client (an Android device) allows the user to record a video, and upload it to the server. The server makes the playlist file for that video available publicly. The uploaded video can then be viewed on a DASH-compliant streaming client such as Apple QuickTime Player by specifying the playlist file.

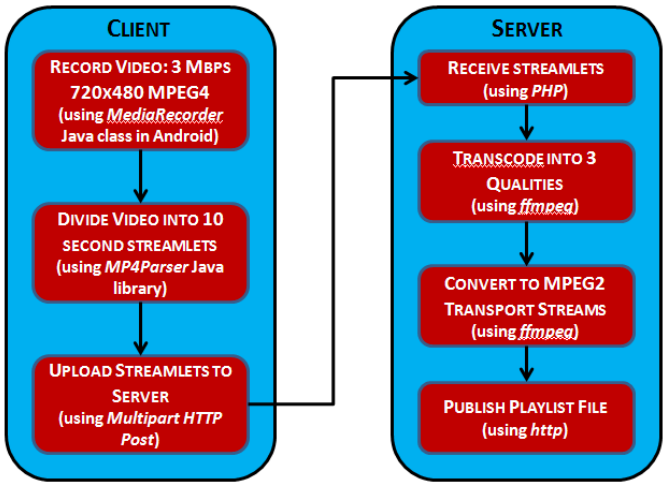


Figure 1: Data-flow diagram for the Client and Server functionality

The Android device used is an ASUS Transformer TF101 [12], running Android Honeycomb 3.2. The server used is a Linux server running Apache (the LAMP stack). The Android device allows us to experiment with video-streaming 'on-the-go' while the connectivity and bandwidth might change frequently. The LAMP server is used since it is free, stable, and comes with no licensing costs.

The functionality of both the client and server is shown in *Figure 1*, in the form of a data-flow diagram. The implementation details are shown in italics in the brackets.

Each step is discussed in detail below –

3.1. Record Video

Android Honeycomb provides a Java API for development of Android applications. Among the classes of the API is the *MediaRecorder* class [13], which allows the programmer to record a video, using the back camera or the front camera, and allows the encoder, bitrate, and resolution to be specified. In this project, the video is recorded at 3 Mbps, with a resolution of 720x480, in the MPEG4 encoding video format.

3.2. Divide into Streamlets

The resulting MP4 video file is then divided into 10 second streamlets. The choice of a proper time length of a streamlet is important. A very small streamlet (say 2 seconds) would lead to a large overhead while streaming the video, since each streamlet is itself an MP4 file containing file headers. However, a large streamlet (say 5 minutes) would reduce the granularity of stream switching, since the video quality can be switched only at the streamlet boundary. An optimal and typically used length is 10 seconds.

For dividing the video into streamlets, the *MP4Parser* Java library [14] has been used. It parses the MP4 Container into its constituent atoms [15], and allows the programmer to perform modifications. It also allows the programmer to manipulate the video at the track level.

Dividing the video tracks into chunks is not trivial. Each track consists of samples. The track can be cut only at the beginning of a sample. More importantly, we need to split the track at a *synch* sample, which is the beginning of an independent Application Data Unit (I-frame in Video). If there are more than one track (audio track and video track), then the synch samples of both tracks should intersect.

Figure 2 gives an overview of how the splitting takes place. The nearest synch sample before time 10 seconds is found using the Time-to-Sample and Synch Sample Container atoms [15] in the MP4 file. In this example, it can be seen that we cannot cut the track at exactly 10 seconds. Moreover, it can be cut only at the nearest synch sample before 10 seconds, in this case sample number 4 at time 9.5 seconds. The streamlets are now [1, 4) and [4, 7]. Sample 4 should not be repeated in consecutive streamlets.

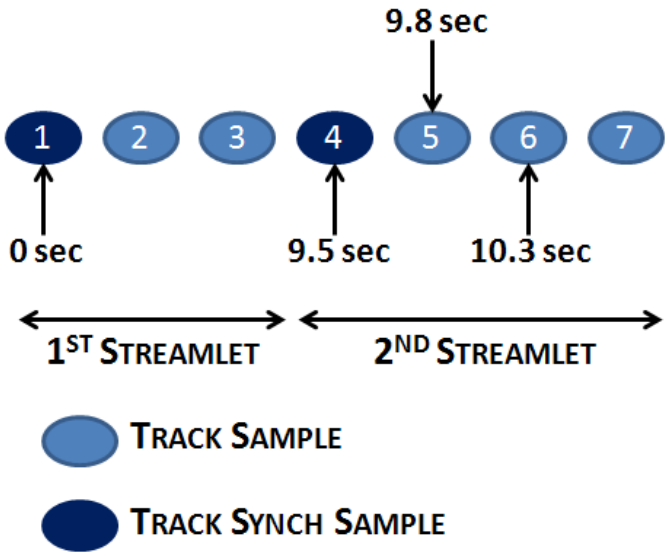


Figure 2: Splitting the track into streamlets based on the synch samples

Since the development of the application is on a tablet with limited memory, care has to be also taken about the memory used while splitting. If the video is first segmented completely into streamlets *before* uploading, then the memory (in RAM or SDcard) gets filled up with the numerous tiny streamlet files. Instead, a better approach is to upload on-the-fly. As each streamlet is created, it is uploaded, and then the memory for that streamlet is released before the next streamlet is created.

3.3. UPLOAD STREAMLETS

The streamlet files are uploaded using a Multipart HTTP Post request [16] to the server. The Post request contains data in the Multipart MIME Message format, suitable for file upload. An example is given in *Figure 3*.

```
Content-Type: multipart/form-data; boundary=AaB03x

--AaB03x
Content-Disposition: form-data; name="submit-name"

Larry
--AaB03x
Content-Disposition: form-data; name="files"; filename="file1.txt"
Content-Type: text/plain

... contents of file1.txt ...
--AaB03x--
```

Figure 3: Example of MIME Multipart Message

The example shows how a file named "*file1.txt*" is uploaded, along with an attribute field called "*submit-name*". The message contains two parts, one for the attribute and one for the file. Using this type of HTTP Post Request is suitable for uploading the streamlet files, as well as attributes such as the streamlet duration and sequence number.

A Java library called *HttpComponents* [17] developed by Apache provides an API for easily creating a Multipart HTTP Post request, and sending it to an HTTP server. Since the streamlets are uploaded one by one in succession, it would be advantageous to keep the TCP connection open during the entire upload process. Otherwise, during each streamlet upload, TCP has to repeatedly do a lengthy handshake to set up the connection.

Using HTTP to do an upload is simple, since the protocol is stateless. However, to provide extra functionality (such as resuming a previous upload from the point it got disconnected), the server needs to maintain state information about the upload (such as the last sequence number of the streamlet it received).

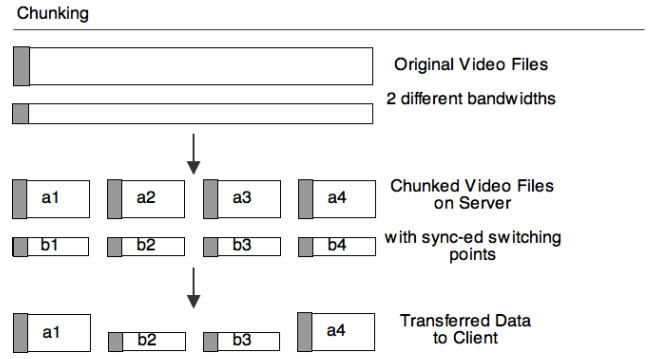
3.4. RECEIVE AND STORE STREAMLETS

The server side has been implemented in PHP, since it is free, available at no cost, and is easy to code. The server runs Linux Apache HTTP server [18]. The server's PHP scripts provide an HTTP interface to the client. This interface allows the client to upload a streamlet, query the list of videos, add and remove playlists, etc. The information about the videos uploaded is stored in MySQL database. It includes details such as the video name, duration, creation date, playlist file, and others.

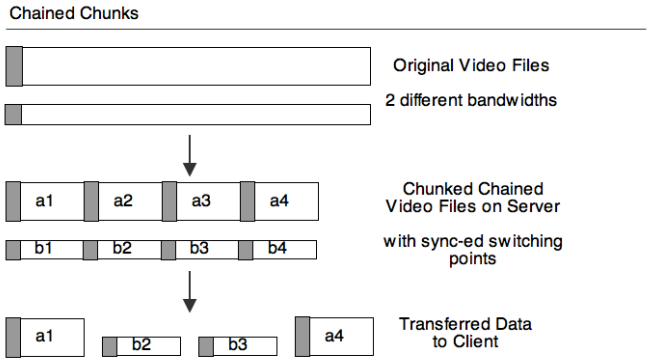
In particular, the server keeps track of the last sequence number of the streamlet it received. If the upload gets interrupted for some reason, the client can later query the server about the last streamlet it received, and resume the upload from the point it left

off. Using sequence numbers provides a rudimentary way of resuming partial uploads.

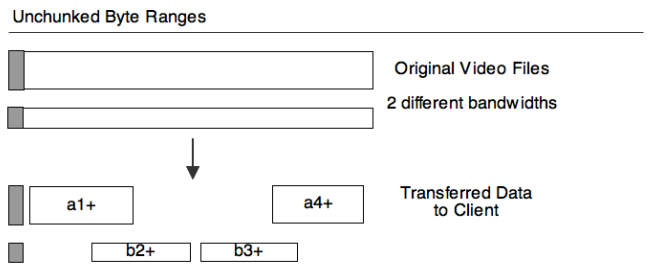
Once the streamlets are received on the server, there are various ways in which they could be stored and served to the video player. Three possibilities, along with their advantages and disadvantages, are given in [19]. *Figure 4* shows them pictorially.



(a)Normal Chunking



(b) Chained Chunks



(c) Byte Ranges

Figure 4: Alternatives for storing and serving streamlets to the client

One way is to store each streamlet as a separate independent file. The client requests each file as needed. The second way is to concatenate the streamlets into one big file. The client then requests byte ranges instead of files. However, each byte range still includes the streamlet header. The third way is to not use streamlets at all. The server simply stores the video in different qualities. The client can request arbitrary overlapping byte ranges. As can be seen, the complexity increases on the client side as we go from the first method to the last method, while the server becomes simplified.

In this project, the Normal Chunking technique is used, which is also the direction in which the DASH standard is going, and many streaming client implementations support. The streamlets are stored as independent files in a repository folder.

3.5. TRANSCODE STREAMLETS INTO 3 QUALITIES

Once a streamlet is received, it is transcoded into 3 qualities –

a. 720x480 3 Mbps H.264/AVC and AAC audio (High)

b. 480x320 768 kbps H.264/AVC and AAC audio (Medium)

c. 240x160 200 kbps H.264/AVC and AAC audio (Low)

Transcoding is done using the *ffmpeg* library [20]. Ffmpeg is a powerful library capable of interpreting almost any video/audio format, and performing operations on them.

The qualities range from 200 kbps to 3 Mbps, which covers a large range of typical bandwidths.

3.6. CONVERT TO MPEG2 TRANSPORT STREAMS

The next step is to convert each MP4 file of each quality into an MPEG2 transport stream, suitable for transfer over a network with packet loss. This conversion is also done using ffmpeg.

3.7. PUBLISH PLAYLIST FILE

Finally, the server creates a playlist file as per the Apple Live Streaming draft standard specifications [11]. There is a master playlist file, which points to 3 other playlist files, corresponding to the low, medium and high qualities. Each playlist file in turn points to the MPEG2 transport stream files obtained earlier.

An example of the master playlist file and a streamlet playlist file is given in *Figure 5*.

These playlist files, along with the streamlets, should be made available publicly. If the server provides a

link to the master playlist file, then any DASH-compliant streaming client should be able to open the playlist file and download the required streamlets.

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1280000
http://example.com/low.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=2560000
http://example.com/mid.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=7680000
http://example.com/hi.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=65000,C
http://example.com/audio-only.m3u8
```

(a) Master playlist file

```
#EXTM3U
#EXT-X-TARGETDURATION:8
#EXT-X-MEDIA-SEQUENCE:2680

#EXTINF:8,
https://priv.example.com/fileSequence2680.ts
#EXTINF:8,
https://priv.example.com/fileSequence2681.ts
#EXTINF:8,
https://priv.example.com/fileSequence2682.ts
```

(b) Streamlet Playlist file

Figure 5: Example of Master and Streamlet playlist files

The server does not support live streaming. In case of a live stream, the playlist files have to be updated carefully as per the timing constraints given in [11]. If the timing constraints are not respected, then the client could download outdated playlist files, or not download them frequently enough to get information about the newly updated streamlets.

4. General Discussion and Comparison

The video hosting service enables the user to record and upload videos to a DASH-compliant server. Any DASH client such as Apple QuickTime Player can play such videos. This section gives a comparison of this solution to other streaming solutions, followed by the learning experience gained by doing this project, and the difficulties encountered.

Without DASH, the application developer would have to either use a proprietary video streaming server (and face licensing costs), or develop his own streaming server and client (and face the coding complexity). With DASH, the application is much easier to develop. The server side in particular is simplified, since it involves only transcoding and generation of playlist files. Moreover, the target audience is wider, since any client supporting DASH can play the uploaded videos, whether it is Apple QuickTime, or Android Honeycomb, or VLC player, and many others.

This project has been a learning experience in multiple ways. I understood how MP4 files are conceptually structured, and how live streaming works. In addition, I learnt a lot about Android programming, especially the newly introduced APIs. I faced two main difficulties in this project – First, I initially had difficulty understanding the MP4Parser library (since I didn't know what atoms were). However, after reading the Apple QuickTime spec, it was easy to understand what the library was doing. Second, I had difficulty checking whether my playlists were ok, since media players do not always play the media list properly (Honeycomb frequently crashes). Later I checked the playlists on VLC player. Overall the project was a reasonable milestone to finish in two months, and was a great learning experience in the process.

5. Future Work

The project can be improved in many ways. First, support can be added for live streaming. However, this is difficult since Android APIs do not make it possible to record and process the video at the same time. Second, the chunking can be done on the server side, since it has the processing speed and power supply. Third, the server-side should perform additional checks to be more robust, such as - the type of file uploaded should be MP4, the duration of the streamlet should be within limits, and the type of encoding of the streamlets should match.

Using DASH introduces latency to a live stream. The latency comes from the multiple TCP connections to download each streamlet, and the time to decode the header of each streamlet. The techniques shown in *Figure 4* could possibly reduce this latency significantly.

6. Conclusion

In the project, a client-server video hosting service was implemented using Android and a LAMP server. It allows a user to record and upload videos, which can later be streamed on a variety of DASH-compliant players. This project demonstrated the simplicity and advantages of using DASH as a stream switching solution.

REFERENCES

[1] **Dynamic Adaptive Streaming over HTTP (DASH) Part 1: Media presentation description and segment formats http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57623**

[2] **Dynamic Adaptive Streaming over HTTP http://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP**

[3] **Real Time Transport Protocol http://en.wikipedia.org/wiki/Real-time_Transport_Protocol**

[4] **Real Time Transport Control Protocol http://en.wikipedia.org/wiki/RTP_Control_Protocol**

[5] **Real Time Streaming Protocol http://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol**

[6] **Multiple Description Coding http://en.wikipedia.org/wiki/Multiple_description_coding**

[7] **Video Streaming by Move Networks http://cable.doit.wisc.edu/SMPTE/02-Infrastructure%20Considerations%20for%20Next%20Generation%20Media/Edwards.TP.01-15-2008.pdf**

[8] **Apple lauches HTTP Live Streaming standard in iphone3 http://www.appleinsider.com/articles/09/07/08/apple_launches_http_live_streaming_standard_in_iphone_3_0.html**

[9] **HTTP Live Streaming http://en.wikipedia.org/wiki/HTTP_Live_Streaming**

[10] **IIS Smooth Streaming http://www.iis.net/download/SmoothStreaming**

[11] **HTTP Live Streaming Draft Standard http://tools.ietf.org/html/draft-pantos-http-live-streaming-06**

[12] **ASUS Transformer TF101 http://www.asus.com/Eee/Eee_Pad/Eee_Pad_Transformer_TF101/**

[13] **Android Developer Reference: MediaRecorder http://developer.android.com/reference/android/media/MediaRecorder.html**

[14] **MP4Parser library http://code.google.com/p/mp4parser/**

[15] **Apple QuickTime File Format 2001 http://developer.apple.com/standards/qtff-2001.pdf**

[16] **MIME Multipart Message http://en.wikipedia.org/wiki/MIME#Multipart_messages**

[17] **Apache HttpComponents Library http://hc.apache.org/httpclient-3.x/**

[18] **Apache HTTP Server http://httpd.apache.org/**

[19] **Ginger's Blog: Adaptive HTTP Streaming for Open Codecs http://blog.gingertech.net/2010/10/09/adaptive-http-streaming-for-open-codecs/**

[20] **Ffmpeg library http://ffmpeg.org/**

[21] **Adaptive Streaming Comparison http://learn.iis.net/page.aspx/792/adaptive-streaming-comparison**