Low Overhead Container Format for Adaptive Streaming

Haakon Riiser[1], Pål Halvorsen[2,3], Carsten Griwodz[2,3], Dag Johansen[4]
[1]Netview Technology AS, Norway [2]University of Oslo, Norway
[3]Simula Research Laboratory, Norway [4]University of Tromsø, Norway

ABSTRACT

Current segmented HTTP streaming systems provide scalable and quality adaptive video delivery services to a huge number of users. However, while they support a wide range of bandwidths and enable arbitrary content-based composition, their current formats have shortcomings like large overheads, live streaming delays, etc. We have therefore developed an adaptive media player that works around these problems while still using standard components like H.264/AVC for video, and MP3 for audio. The system's adaptivity allows the player to pick a quality level that makes good use of available bandwidth and CPU resources while at the same time maintaining smooth uninterrupted playback, as well as offering near instant seek and startup times.

This paper presents an appropriate way of coding the segments and a simple multimedia container format that is optimized for adaptive streaming and video composition over HTTP. We show that our format is sufficiently advanced to contain any payload type, while being trivial to parse and translate to other container formats. Additionally, we show that our format is second to none in terms of overhead, without incurring any penalties on live streaming.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Applications*; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*Video*

General Terms

Experimentation, Measurement, Performance

Keywords

adaptive streaming, container format, file format, http, low latency, low overhead, multiplexing, video

1. INTRODUCTION

Next generation entertainment and streaming systems will have strong efficiency requirements – they must support a huge number of concurrent users while at the same time allowing the users to make and share personalized video playouts. This is possible even today using video sites like YouTube, which provides indivisible user-generated video objects. However, we envision systems where users may query for video snippets. Upon receiving the results of the query, small fragments of video from multiple video sources would be combined dynamically in any order into a seamless, personalized video playout. Additionally, the range of user devices is even today wide, ranging from small mobile phones and PDAs to large HD-capable home cinema systems. Thus, the system must be able to combine content arbitrarily and scale the quality according to end-device and resource availability. In particular, bandwidth fluctuations in unstable mobile networks may be large [10] making adaptive video streaming a necessity.

In this context, HTTP streaming solutions have many of the required features, and the current commercial success of such solutions has several reasons. One reason is that the use of IP multicast for concurrent delivery to several receivers is still blocked by most Internet Service Providers. Another is that the integration of video into social networking sites individualizes access patterns, which reduces the potential of IP multicast-based approaches. Given these problems and the general impression that backbone bandwidth is abundant and free, HTTP streaming as an implicitly TCP- and firewall-friendly end-to-end streaming solution with minimal needs for infrastructure deployment has become a very desirable option. In [8], we demonstrated [1] a system using an HTTP-based protocol that is able to adapt dynamically to resource availability and supports on-the-fly composition of content-based videos from multiple sources. This system achieves high-quality video presentations with a very flexible bit rate adaptation scheme. Like the systems from Move Networks [4], Microsoft [5] and Apple [9], we use small segments downloaded from possibly different locations, even exploiting web caches, which reduces the need for a dedicated overlay network. However, the requirement of seamless video playout of arbitrarily concatenated video objects and the supported bandwidth range introduce additional requirements with respect to coding and container formats.

An important consideration when deciding on a media encoding strategy and container format is how the chosen technology performs in low-bandwidth scenarios like streaming over wireless networks to mobile devices. Take, for example, Apple's HTTP live streaming system [9]: it uses MPEG-2

Transport Streams [6], a format known for large overheads due to frequent repetition of metadata. Considering that it is used for streaming to bandwidth-limited mobile units (iPhones and iPods), overhead on low-bitrate streams may affect users' perception.

To avoid this problem, we demonstrate a working [1] media player that uses a tailor-made approach for coding and multiplexing that meets all our system requirements; i.e., it must:

1. Be able to use standard codecs (e.g., H.264/AVC and MP3) and regular web servers.
2. Support quality adaption according to end-devices and oscillating resource availability (see figure 1).
3. Scale to a huge number of simultaneous users.
4. Support fast and easy seeking to specific positions in the stream.
5. Use a format with minimal multiplexing overhead.
6. Be able to play segments from different clips seamlessly in any order (see figure 2), without picture artifacts and noticeable inter clip delays (one smooth playout from various sources).

The motivation for goals 1–5 needs no explanation; goal 6, however, is important because it allows for personalized playouts without requiring expensive transcoding and redundant storage of video. Example applications include personalized video clips returned by search engines, user-generated videos that can easily be shared on social networks, and much more powerful video editing in web based content management systems.

In this paper, we describe the container format we used for interleaving streams of different types into one. Compared to the standard MPEG-2 Transport Stream container, we have drastically reduced the bandwidth overhead for low-bitrate streams, and unlike MP4, our container format does not introduce extra latency when streaming live content.

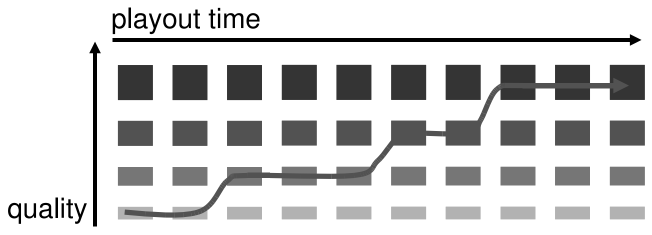

Figure 1: Video quality and bit rate adaption

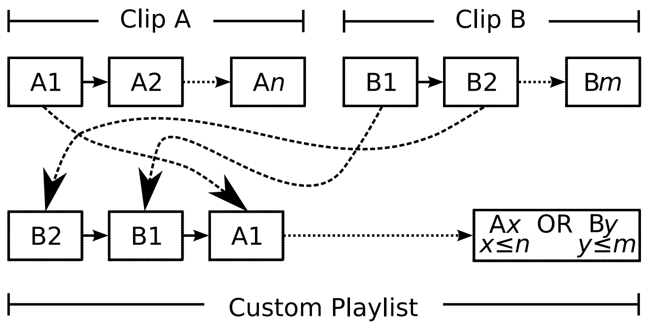

Figure 2: Custom playlists with out-of-order playback

2. SEGMENTED HTTP STREAMING SYSTEMS

Traditionally, video streaming solutions based on HTTP make the assumption that a video playout will be linear and therefore download the entire video stream as one file. For these solutions, container formats such as MP4/Quicktime and ASF are used, and media is packaged into a contiguous multiplexed stream. Essential, but non-changing, information required by the decoder (such as sequence and picture parameter sets in the case of H.264/AVC) is stored once at the beginning or end of the stream. Such an approach does however not support the required interaction and free composition of video snippets into a continuous playout. Playback can still start at random locations, but there is no easy way to adapt quality, the client must download a metadata header or trailer before seeking to the intended location in the stream, and the client does not know the byte position to which it needs to seek without also downloading an index that describes the entire stream.

Targeting better scalability, segmented, adaptive HTTP streaming systems have recently gained a lot of commercial interest, and have proved very scalable with millions of concurrent users [4]. Their approach is to chop a video object into segments, where segments are available for a wide range of qualities, and thus bit rates. Quality can be switched on segment boundaries by downloading the next segment in a different quality than the current one, as is illustrated in figure 1. Such an approach is today widely used by systems like Move Networks [4], Microsoft's Smooth Streaming [5] and Apple's Live HTTP streaming [9]. However, although they all support a wide range of bandwidths, and they all (in theory) support arbitrary combinations of segments (see figure 2), their current technology still fails in one way or another:

Microsoft's Smooth Streaming [5] uses an MP4 extension called "fragmented MP4" [12] for its wire format. This streaming technology is included with their Silverlight media player. The system can freely combine segments, but, according to our tests, the player seems to need a long reset delay; specifically, there will be a noticeable latency when switching from one video clip to another (note that this is most likely an implementation issue, not a protocol issue). Additionally, the fragmented MP4 format, although efficient in terms of overhead, introduces extra delay when used for live streams. The reason is that every MP4 fragment starts with a complete index of every frame in the segment. The index cannot be generated before the size of every frame in the segment is known, which means that a live encoder must completely fill a segment before it can be shipped. Although live streaming is not the most obvious application of our system, it is still desirable to support high quality adaptive live streaming using the universally supported HTTP protocol, and in some use cases, such as sports and business related news, every second of delay counts.

Apple's Live HTTP streaming [9] uses plain MPEG-2 Transport Streams for packaging, which is a very bandwidth hungry format (see section 3.2). Segments can still be played out in arbitrary order, provided that all segments are coded so that no picture references cross segment boundaries. (It is interesting to note that the test clips we found [3] on Apple's web site were not coded in this way, causing artifacts when they were played out-of-order.)

Move Networks [4] is a closed system, and important details are hard to find, so a comparison of the relevant parts could not be made. However, because Move is the most well know provider of adaptive streaming, we feel it is worthwhile to mention a few of Move's drawbacks in other aspects compared to our system. First, Move typically uses On2's proprietary VP6 and VP7 codecs, both of which are inferior to most H.264 implementations in both quality per bit [2], encoding speed and decoding speed. Comparing On2's VP7 codec to well-known open source implementations of H.264 (x264 for encoding and FFmpeg for decoding) using comparable encoding parameters, we found the open source H.264 implementations to be more than twice as fast as the VP7 codec for both encoding and decoding. Regarding overhead, a thorough analysis could not be done due to the lack of specifications, but Move estimates [4] a packaging overhead of 10 % that probably includes protocol headers. More importantly, because it is a closed system, it is not possible to extend the player with customized Javascript interfaces, which limits integration with the search and personalization services we have built using standard web languages like HTML, CSS, Javascript and PHP.

Because all of the above technologies have limitations with respect to multiplexing overhead, arbitrary combination of segments, live content, or other issues, we have designed and implemented a similar system that meets every requirement stated at the end of section 1.

By encoding the video stream so that the number of pictures per group of pictures (GOP) is fixed, we know that, if we encode the same stream in multiple qualities, GOP n in quality a will contain exactly the same frames as GOP n in quality b, only with a different picture quality. This means that we can switch quality on a GOP boundary without shifting the playout position, achieving goal 2. Note that keeping the number of frames/GOP exactly constant is not a strict requirement for indexless seeking, but it makes it easier to implement. Using a variable frame rate, one can still try to keep GOPs approximately the same duration. As long as GOPs are encoded to avoid drift, that is, a deviation from the target duration in one GOP is corrected by later GOPs, GOP indexes will still map closely to presentation time.

Goal 3 is achieved through the distributed nature of segmented video stored on regular web servers. By using a segment tracker that provides a client with a list of URLs for every segment, segments can be hosted redundantly on an unlimited number of web servers. Additionally, transparent web caching also helps off-load the servers.

To achieve goals 1 and 4, we store each GOP as a separate file on the server. Its file name contains the GOP index number, and the client can instantly access any part of the video stream with a simple HTTP GET request on the desired GOP(s). This simple approach means that no index is required when seeking, neither by the server nor the client. The GOP index number n easily maps to its presentation time T in the movie by multiplying n with the fixed GOP length, enabling fast seeking to specific points in time on streams hosted on regular web servers. Additionally, the player starts downloading low quality, low rate segments in order to achieve a fast startup and seek. Note that this approach implies that only GOP *duration* must be fixed, *not* its size in bytes. What we do is completely compatible with variable bitrate coding (VBR).

To achieve support for arbitrary concatenation of segments and still have a smooth, seamless playout (goal 6), every GOP must be closed[1]. This means that GOPs must be encoded so that no frames in any GOP reference frames from another GOP.

Finally, goal 5 aims at saving network bandwidth, and is mainly achieved by ensuring that session-wide codec parameters are sent only once per session, separate from the multiplexed stream data. The same approach is used by the Real Time Streaming Protocol (RTSP) [11], which uses the Session Description Protocol (SDP) to send this kind of information. With this method, the only data required in the multiplex is that which may change from frame to frame (or segment to segment). The details of the coding and container format are explained next.

3. A NEW CONTAINER FORMAT

To support smooth playback on a wide range of networks, it is important to be able to select very low bitrate streams, and as such, the container format used for interleaving audio and video should be efficient.

The next subsections describe in detail a container format we developed for our media player implementation. It is especially designed for our purpose, and emphasizes low overhead, simplicity, fast startup and low delay for live content.

3.1 Metadata stored in the container stream

Since most of the parameters that control the decoding can be kept fixed for all segments, the only metadata that needs to be stored in the actual segments is that which can change from one segment (or frame) to another. Assuming that every video clip available is encoded with the same codecs and audio sample rate[2], the only parameters that can change from one payload unit to another are:

- Payload type (audio, video, subtitles, etc). Represented in the frame header (figure 3) as two bits: audio is 00, video is 01, 10 is currently unassigned, and the final value (11) indicates that an additional 32-bit header extension field follows after the frame header, and guarantees support for future formats.

[1]For a more coarse-grained segmentation strategy, one can envision that a segment consists of several GOPs – in that case, the requirement is only that the segment's first and last GOP's do not have frame references that cross the segment boundary.

[2]Audio sample rate is fixed, not because we want to save a couple of bytes per segment, but because changing sample rates during playback is hard to do properly without resampling the decoded audio before sending it to the sound card.

- Presentation timestamps (PTS) of the first audio and video presentation units in the segment. A single bit in the frame header indicates if this 32-bit value is present.

- Picture aspect ratio. A single bit in the frame header indicates if this 32-bit value is present.

- Frames per second (only used to minimize overhead for segments with a constant frame rate). A single bit in the frame header indicates if this 32-bit value is present.

- Number of bytes until the next frame header (or, in the case of the last header, to the end of the stream).

For variable frame rate video, every picture needs a PTS value, but the extremely common case of constant frame rate video and uninterrupted audio makes it possible to discard all PTSes except the first audio and video PTS in a segment. The first PTS values in a segment *must* be present, since audio and video might not start at the same time, so after a seek, we need these initial PTS values in order to preserve audio/video synchronization. The following PTS values in the segment can, however, be generated. For audio, we simply use the fact that audio plays uninterrupted, so the next PTS is always given by the current time plus the duration of the last decoded audio frame. For video, we need to know the number of frames per second.

The aspect ratio must also be included for every segment if one wishes to be able to seamlessly play out segments from different video sources (which may not have the same picture aspect ratio).

The purpose of the payload type field is obvious, but note that it is simply a broad categorization; that is, it tells the client if the payload is audio or video, but not which codec is used. The specific codec information is located in the session-wide metadata file.

The last field is the byte offset to the next frame header. It is included in every frame header, and this number tells us where one frame ends and another begins. Having this information attached to each frame is important, mainly because it enables video streams to be generated and transmitted without any segmentation delay. Most container formats distribute frame size fields in this way, but some, such as the MP4 format used by Smooth Streaming, keep all frames fused together in one inseparable chunk, and force the demultiplexer to consult an index with byte ranges for every access unit. The index-based method does not work well with live video because one does not know the size of each frame in advance, and thus, segments can not be shipped until every frame in the segment is encoded). Furthermore, since we use TCP as the carrier protocol, there is no danger of loosing critical metadata that would make the rest of the segment unparseable, so there is really no reason for structuring the data in such a suboptimal way.

Finally, we note that the byte count is the offset to the next frame header. This means that it includes not only the payload bytes, but also the metadata fields. By this interpretation we improve backwards compatibility. For example, if new payload types require additional fields that an old version of the client does not recognize, it does not need to understand the contents of that payload, it can simply ignore it and use the offset to move to the next frame header.

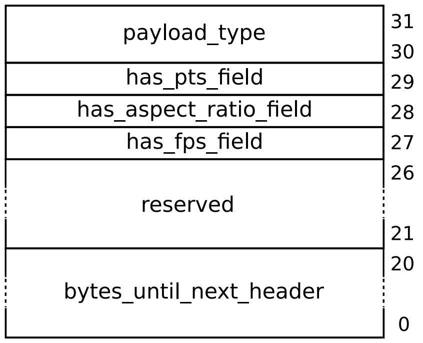

Figure 3: 32-bit frame header that leads in every frame in the multiplexed stream

We have done as thorough an analysis on overhead as possible with the documentation that was available to us. Our results are illustrated in figure 4 and described in this section. Figure 4 compares overhead in the case of Apple's low-bitrate sample video. This bitrate was chosen because it represents a realistic lower bound on quality, which is where overhead matters most (multiplexing overhead is mostly proportional to the stream's duration, not the stream's size in bytes; i.e., overhead becomes unimportant for high bitrates). Based on our findings in the container format analysis, we also present a graph (figure 5) that shows how the container overhead varies as function of the payload bitrate.

We start by noting that Move Networks' technology lacks specifications, so it could not be properly analyzed. However, in their own technical overview [4], we found an estimation of 10 % overhead including protocol headers over the bandwidth that is consumed for audio and video data. We have thus included Move's number in figure 4 because it presents a very low bitrate. I.e., it is a best case scenario for Move's 10 % overhead. Move is not included in figure 5 because we do not know how that number would vary as a function of the payload bitrate.

Apple was easier to analyze, since they recently suggested a new standard for adaptive streaming based on their use of MPEG-2 Transport Streams and HTTP [9]. To determine the overhead of Apple's approach, we have used low-bitrate sample streams presented on Apple's web site [3]. The audio bitrate was 32 kbit/s, and the video bitrate was 136 kbit/s. The segment length was fixed at 10 seconds, and every segment contained 215 compressed audio frames and 150 compressed video frames.

3.2 Overhead comparison

The layout of the stream is illustrated in figure 6. The transport stream packet headers occur every 188 bytes, and their size varies from 4 to 12 bytes. Atypically, overhead caused by these headers thus increases proportionally with the segment's size in bytes, meaning that relative overhead does not approach zero as the bitrate increases. Because of fluctuations in the audio/video bitrates, the segment byte size is not perfectly constant, but for the sample clips analyzed, the bitrates were remarkably stable, so we found a fairly constant overhead of about 7300 bytes/segment. In addition, figure 6 shows that the stream contains data other than audio and video, such as a Program Association Ta-

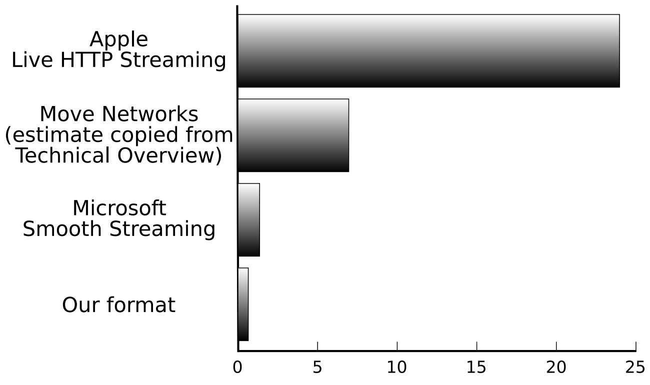

Figure 4: Overhead as a percentage of the combined audio/video data in a 168 kbit/s stream

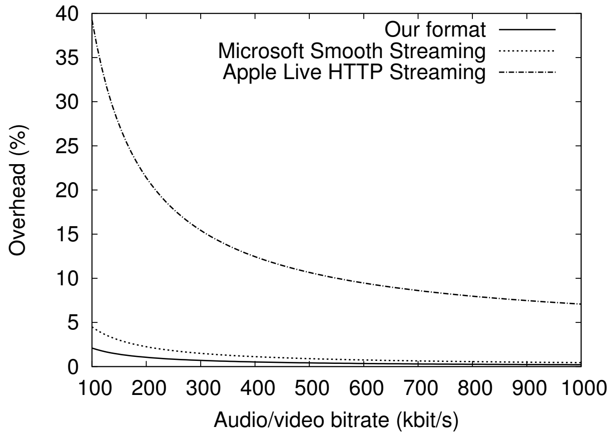

Figure 5: Relative overhead as a function of the audio/video bitrate

ble. The total overhead from non-audio/video streams was always exactly 37168 bytes/segment. The audio and video streams are additionally encapsulated in Packetized Elementary Streams (PES), which, in this sample, incurred an extra overhead of 14 bytes per audio frame, and 19 bytes per video frame. This amounts to a total of $215 \times 14 = 2850$ bytes for audio and $150 \times 19 = 3010$ bytes for video. The grand total overhead for a segment is the sum of all of the above numbers: $7300 + 37168 + 2850 + 3010 \approx 50$ kB/segment. Since the segment size is fixed at 10 seconds, we have an overhead of approximately 40 kbit/s. Considering that this is almost 24 % of the combined audio and video bitrates, the overhead is considerable.

Using our container, all audio and video frames are preceeded by a 4-byte frame header (shown in figure 3 and described in section 3.1). In addition, the segment's first audio frame has a 4-byte presentation timestamp field, and the first video frame has 4-byte fields for presentation time, aspect ratio and frame rate. This means that the first audio frame has an overhead of 8 bytes, the first video frame has 16 bytes overhead, and all the following audio/video frames in the segment have 4 bytes overhead. Total overhead for the case analyzed above (10 second segments, 215 audio

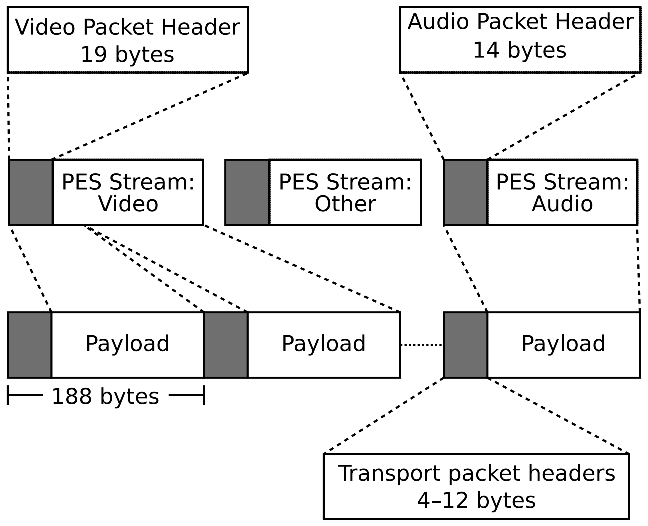

Figure 6: Layout and overhead in a typical MPEG-2 Transport Stream

frames, 150 video frames) is $8 + 16 + (214 + 149) \times 4 = 1476$ bytes/segment ≈ 1.2 kbit/s.

Smooth Streaming works similarly to our solution. Most metadata that is required for decoding is stored in an XML file that is downloaded separately from the media stream's data. Its segments are transferred over the network as fragmented MP4, which is a low-overhead way to use the standard MPEG-4 container [7]. Each segment starts with a fragment header whose size, in the samples we found, was 8 bytes per frame (4 bytes for frame duration, 4 bytes for frame size) plus a constant contribution that does not depend on segment length. Note that the format allows the frame duration to be excluded for constant frame rate clips, making overhead per frame 4 bytes (the same as with our format), but Microsoft has not used this feature in their constant frame rate demo clips. Regardless, the overhead with Smooth is small enough not to matter. The problem with fragmented MP4 is that the multiplexer requires the encoding of an entire segment's data before the frame index can be created. For live streaming, this mandates a delay equal to the segment duration (typically two seconds) that is avoided in our format.

3.3 Discussion

Since there is no standard for adaptive HTTP streaming, there is no reason to use any of the existing container formats. Existing players would not be able to make use of adaptive streaming even if we used a standardized format, since it is the client that makes all choices related to adaptation. The client must request each segment as the movie plays, and this is not how traditional HTTP streaming works.

Segmented streams stored in a non-standard container can still be used in a traditional fashion with old devices; what is important is that audio and video is encoded using standard codecs and encoding profiles. Repackaging the compressed data into a new container is easily handled with a server-side filter that remultiplexes the stream to a traditional format. This process is very cheap; disk I/O is the bottleneck, so it would be as fast as sending the stored stream directly.

It is also possible to go the other way, as Microsoft has done with Smooth Streaming: A stream encoded for segmentation can be stored on the server as a single file in a traditional container. Upon receiving a segment request, an intelligent web server can use the container's index to locate the appropriate data and send it back over the wire in a more bandwidth efficient container. In this case, one trades the significant benefit of supporting regular web servers for the questionable benefit of having fewer files on the server's file system.

4. CONCLUSION

We have created a segmented HTTP streaming system where segments are coded as closed GOPs with H.264 video and MP3 audio, and we have used a custom-made low-overhead container format. The system enables us to efficiently stream video to a wide bandwidth range because the video is composed of small video snippets that can be combined dynamically, on-the-fly, in any order or quality, from multiple video sources into a seamless, personalized video playout.

We have removed all redundancy in the transmission protocol: Static data is sent once per session, and frame indexes are not required. This greatly simplifies the multiplexing format, reduces the overhead compared to many other existing approaches, and enables low-latency live streaming. Thus, using our way of segmenting, coding and packaging, we support next generation entertainment systems, bringing better quality and more control to the users.

For our future research, we will do a thorough analysis on the optimal segment duration. With the segment duration parameter, there is a trade-off between coding efficiency and granularity of search, seek and quality adaption. We will use both subjective user tests and traditional signal processing tools to find the best compromise.

5. REFERENCES

[1] Demo: DAVVI – next generation entertaiment platform. http://home.ifi.uio.no/paalh/DAVVI.mp4.

[2] Codec shoot-out. http://www.doom9.org/index.html?/codecs-main-105-1.htm, 2005.

[3] Live HTTP streaming sample. http://devimages.apple.com/iphone/samples/bipbopall.html, 2009.

[4] Move Networks. http://www.movenetworks.com, 2009.

[5] SmoothHD. http://www.smoothhd.com, 2009.

[6] ISO/IEC 13818-1:2000. *Generic coding of moving pictures and associated audio information: Systems.*

[7] ISO/IEC 14496-12:2005. *Coding of audio-visual objects: ISO base media file format.*

[8] Johansen, D., Johansen, H., Aarflot, T., Hurley, J., Kvalnes, Å., Gurrin, C., Sav, S., Olstad, B., Aaberg, E., Endestad, T., Riiser, H., Griwodz, C., and Halvorsen, P. DAVVI: A prototype for the next generation multimedia entertainment platform (demo). In *Proceedings of ACM International Multimedia Conference (ACM MM)* (Oct. 2009), pp. 989–990.

[9] R. Pantos (ed). HTTP Live Streaming. http://tools.ietf.org/html/draft-pantos-http-live-streaming-01, 2009.

[10] Riiser, H., Halvorsen, P., Griwodz, C., and Hestnes, B. Performance measurements and evaluation of video streaming in HSDPA networks with 16QAM modulation. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)* (June 2008), pp. 489–492.

[11] Schulzrinne, H., Rao, A., and Lanphier, R. Real time streaming protocol (RTSP). *IETF RFC 2326* (1998).

[12] Zambelli, A. IIS smooth streaming technical overview. http://www.microsoft.com/downloads/, 2009.