Experimentation with Adaptive Streaming

Santeri Volkov
Student No. 52xxxx
Aalto University
santeri.volkov@aalto.fi

Eelis Kostiainen
Student No. 59xxxx
Aalto University
eelis.kostiainen@aalto.fi

***Abstract*—Media streaming applications have changed the nature in which people consume entertainment. The market has seen an increasing number of streaming content providers, such as Netflix, Hulu, and YouTube. A key factor in keeping customers happy, is the quality of service, which is heavily influenced by the buffering of the viewed media. In this paper, we analyze the adaptability of video streaming over DASH and HLS, under different network conditions.**

***Index Terms*—adaptive streaming, nginx, dash, hls**

I. Introduction

The streaming requirements of today's internet have become pressing on the bandwidth. Modern web browsers stream content over hypertext transfer protocol (http). Content is fetched on-demand, rather than all-at-once like in the earlier days of content streaming. This relates to the behaviour of streaming services consumers. Consumers tend to skip large chunks of content, which removes the need to download all content at once for the user.

In this paper, we investigate and analyze how adaptive streaming behaves under different network conditions. We investigate two different streaming protocols, DASH and HLS, and compare differences in their performances.

II. Experimentation setup

A. Video Data I DASH

We started setting up our adaptive streaming server by first preprocessing our video file to 5 different quality video tracks with corresponding bitrates and also one audio track. For this we used FFmpeg software [1] and the commands in listing 1.

Listing 1: Preprocessing a video for dash

```
Audio:
$ ffmpeg -i video.mp4 -c:a copy -vn
  video-audio.mp4

Video:
$ ffmpeg -i video.mp4 -an -c:v libx264
  -x264opts 'keyint=60:min-keyint=60'
  -b:v 5300k -maxrate 5300k -bufsize 2650k
  -vf 'scale=-1:1080' video-1080.mp4
$ ffmpeg -i video.mp4 -an -c:v libx264
  -x264opts 'keyint=60:min-keyint=60'
  -b:v 2400k -maxrate 2400k -bufsize 1200k
  -vf 'scale=-1:720' video-720.mp4
$ ffmpeg -i video.mp4 -an -c:v libx264
  -x264opts 'keyint=60:min-keyint=60'
  -b:v 1060k -maxrate 1060k -bufsize 530k
  -vf 'scale=-1:478' video-480.mp4
$ ffmpeg -i video.mp4 -an -c:v libx264
  -x264opts 'keyint=60:min-keyint=60'
  -b:v 600k -maxrate 600k -bufsize 300k
  -vf 'scale=-1:360' video-360.mp4
$ ffmpeg -i video.mp4 -an -c:v libx264
  -x264opts 'keyint=60:min-keyint=60'
  -b:v 260k -maxrate 260k -bufsize 130k
  -vf 'scale=-1:242' video-240.mp4
```

- -i - specifies the name of the input file
- -vn / -an - tells ffmpeg not to extract video or audio track
- -c:a / -c:v - tells what audio and video codecs to use
- -x264opts - specifies H.264 specific options to use
- -b:v -maxrate - are the bitrate we want to encode the video to
- -bufsize - is the buffer size we want for the videos
- -vf - tells what format we want the video be in

The input video [2] we used is 5 minutes and 13 seconds long, 1920 by 1080 resolution with H.264 encoding, 60 frames per second and an original bitrate of 4369kbps. The audio on the video is MPEG-4 AAC stereo with 44kHz sampling rate and 127kbps bitrate. The file size of the video is 177MB.

With FFmpeg we extracted from the original video first its audio track in its original quality, and then the video track, encoding it with H.264 codec to 5 different resolutions with corresponding bitrates. We also forced the video to have a keyframe every 60 frames with the -x264opts option 'keyint=60:min-keyint=60', meaning once every second, to allow the video to be segmented later to one second chunks.

Next we genereted a Media Presentation Description (MPD) file with a program called MP4Box [3]. MPD file references to the DASH player the different video and audio tracks and also tells it the interval of segments in the tracks. We specified to the program that our segment size is 1000ms and referenced our video and audio files to it.

```
MPD File:
$ MP4Box -dash 1000 -rap -frag-rap -profile
onDemand -out video.mpd video-1080.mp4
video-720.mp4 video-480.mp4 video-360.mp4
video-240.mp4 video-audio.mp4
```

After this the video is ready to be played by a video client supporting DASH by pointing the video.mpd file to it with the video files in the same directory.

B. Video Data II HLS

For HLS video data we did roughly the same preparations as for the DASH video data. With HLS we found it simpler to use only FFmpeg and some extra HLS specific options to create the video data and separate manifest.m3u8 files for each

video quality with the audio data included in each of them. We added '-hls_time 1 -hls_playlist_type vod' optio ns to again specify the segment length of 1 second and video profile/type as video on demand.

The HLS video data differs from the DASH most noticeably by how it is structured into hundreds of one segment long *.ts* (transport stream) video files and a separate manifest file for each quality. And these files are then all referenced to the player via a main manifest.m3u8 file.

The used commands (4) and the main manifest.m3u8 file (5) can be found in the appendix.

C. Web Server

The experiment included setting up a streaming server, and a streaming client. The server was set up using *nginx*. Nginx is a technology, often used as a web server or a load balancer. For the experiment, we used nginx together with nginx-rtmp [4], which is an nginx module for media streaming with ready-made support for DASH and HLS streaming.

Listing 2: nginx streaming server configuration

```
rtmp {
    server {
        listen 1935;
        notify_method get;
        allow play all;

        application dash_streaming {
            dash on;
            dash_path /var/lib/stream/dash;
        }

        application hsl_streaming {
            hls on;
            hls_path /var/lib/stream/hls;
        }
    }
}

http {
    include /etc/nginx/mime.types;
    server {
        listen *:8080;
        server_name adaptive_streaming_server;
        access_log /var/log/nginx/access_log.log;
        error_log /var/log/nginx/error_log.log;

        root /usr/share/nginx/html;

        location / {
            autoindex on;
            add_header Access-Control-Allow-Origin *;
            add_header Cache-Control no-cache;
        }

        location /dash {
            root /var/lib/stream;
            add_header Cache-Control no-cache;
        }

        location /hls {
            types {
                application/vnd.apple.mpegurl m3u8;
                video/mp2t ts;
            }
            root /var/lib/stream;
            add_header Cache-Control no-cache;
        }
    }
}

events {}
```

With the configuration in listing 2, we ran the streaming server on local machines. The server was ran locally to ensure maximal control of the (emulated) network conditions. The server was ran in a *Docker* container to minimize dependencies to the host machine, and to enable easy deployment regardless of the host platform. The container was based on an image [5] with nginx, and nginx-rtmp readily installed. The container was mounted with the custom server configuration from listing 2.

To ease development and deployment, a script for automated container building and deployment was used. The script included cleaning possible old versions of the image, building the image, running it, and finally mounting the media resources inside the container. The used script can be seen in appendix 3.

To ensure the reliability of our video streams' performance readings we also set them up on an Apache2 HTTP server (configuration in appendix 6) serving the DASH and HLS videos and clients. The Apache server was also set up on local machines to get a good reference point for the nginx server.

D. Video Clients

In addition to serving video over HLS and DASH, the web server also needed to serve client players for the media. In lack of native browser support for DASH and HLS, external libraries were used for the web clients. Dash.js [6] was used for DASH video streaming, whereas hls.js [7] was used for HLS streaming. Both of the player libraries' projects had a debugging reference client available for testing adaptive video streaming with their respective protocols. We utilized these players in testing and analyzing our videos' playback from the servers. The clients show live statistics of current buffer levels, download speed, video quality and other miscellaneous values, which do not add value to this experiment.

III. Measurement

To test the behavior of our adaptive streaming protocols, DASH and HLS, under different network conditions, we emulated different network conditions using Google Chrome developer tools, which natively support bandwidth throttling, and observed the results in the video playback with statistics on the debug video players.

The measurements were conducted using the debug players on full-screen mode with a *full hd* (1920x1080) resolution. This was to ensure the player would attempt to use the best possible video quality within given network conditions.

The first measurements were conducted with five distinct network download throttles. The throttles were implemented with Google Chrome's network throttling profiles, which are a built-in feature for the browser. Browser caching was also disabled to ensure the quality of the measurements. The throttling profiles were configured based on video bitrates, so that the download throttle was slightly above a given video quality bitrate. The bitrate to network throttle mapping shown in listing III.

- 246 kbps (430x242) video, 260 kbps network
- 547 kbps (640x360) video, 560 kbps network

- 946 kbps (850x478) video, 960 kbps network
- 2126 kbps (1280x720) video, 2140 kbps network
- 4794 kbps (1920x1080) video, 4810 kbps network

Next we wanted to test how much overhead each protocol and client needed on the bandwidth to play the respective quality. This we tested by incrementally increasing the allowed bandwidth by 50-100kbps and waiting 10 to 20 seconds between increments for the client to react. Then we repeated the same procedure in the opposite direction to confirm the cut-off point. At the same time we observed short and longer term changes in the clients behavior caused by the bitrate changes.

IV. Results and Analysis

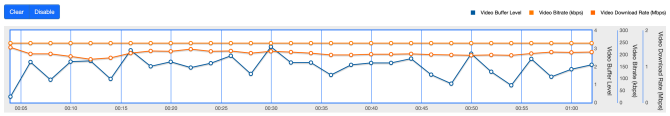
In this section, we present the results of our measurements. First, we present the measurements from the DASH streaming under varying network conditions, with both continuous and intermittent throttling. Then, we take a look at how this compares to the measurements of the HLS streaming.

A. Dash

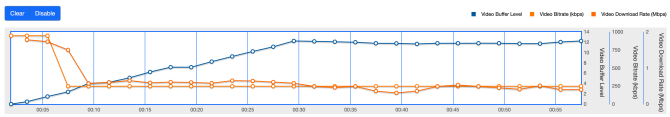
The results look rather uniform. Figure 1 presents results from different continuous network throttlings. In each of the cases, the video bitrate remains stable. However, in 1b, and 1c there seems to be some throttle startup latency, which causes the initial video bitrate to be slightly higher than what can be upheld with the given network bitrate. While it does show a clear correlation between the video bitrate and the available network conditions, we'll interpret the anomaly as an issue with the Chrome throttle profile.

In 1a, we see that the video buffer fluctuates with a pattern: when the buffer level sinks to level 1 (with some variation caused by measurement accuracy), the video stops playing until the buffer is increased by 2 levels. Once the buffer has increased, the video starts playing, which can be seen from the buffer graph, which seems to be sinking under certain conditions. This seems be caused by increased concentration of audio within that the part of the video.

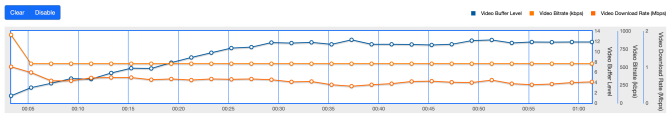
In 1b, we notice that when the available bandwidth is slightly above the videos bitrate (560kbps available, for 547kbps video), dash prefers a lower quality video (246kbps bitrate), and a high buffer level. The buffer level reaches level 12 with a linear rise, and caps there. After the buffer caps, it remains steady, which means that the buffer is receiving data at the same rate it's clearing it, which is the desired behaviour. For 1c, we see the same behaviour, but the for the subsequent video bitrate, i.e. 547kbps. In fact, each of the throttle bitrates tend towards the lower bitrate video.



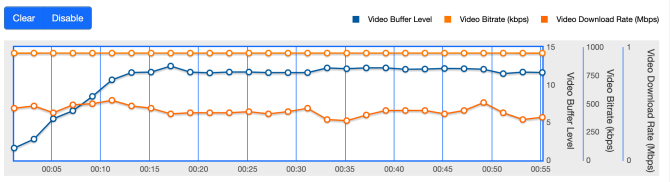
(a) 260 kbps



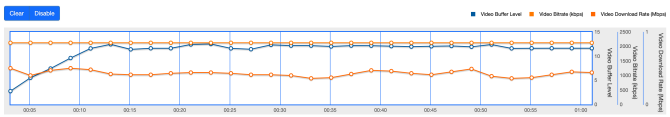
(b) 560 kbps



(c) 960 kbps



(d) 2140 kbps



(e) 4810 kbps

Fig. 1: Continuous throttling with download throttles from listing III

The results for the required bitrates for playing the respective video qualities were also pretty uniform. The required bandwidth overhead was about 10 to 15% of the sum of video and audio bitrate. The audio bitrate was 127kbps and the video qualities and their required bitrates are shown in listing IV-A.

- 246 kbps (430x242) video, 400kbps, buffer caps at level 12
- 547 kbps (640x360) video, 800kbps, buffer caps at level 12
- 946 kbps (850x478) video, 1250kbps, buffer caps at level 12
- 2126 kbps (1280x720) video, 2600kbps, buffer caps at level 12
- 4794 kbps (1920x1080) video, 5500kbps, buffer caps at level 30

While testing the thresholds of the video adaption the client reacted to the bandwidth changes in varying ways. Firstly it reacted within 2-3s by increasing the video quality after increasing or decreasing the bitrate over the thresholds in listing IV-A. Also when varying the bandwidth between the different qualities the buffer level capped at maximum at 12, but when the best quality was in use the buffer filled up to level 30. Secondly when using a bitrate of about 100 to 300

under the thresholds the player tried to start loading the better quality when its buffer level increased to 12, but couldn't hold the quality much longer than 30 seconds before it needed to lower back to the original quality as it's buffer level had gone too low and needed to be refilled as is happening in figure 2. With greater difference to the threshold the fluctuations in quality occurred much more frequently as in figure 3 with bitrate 300kbps under the stable rate. So the long term changes were the clients reactions to the buffer level's changes, and the short term changes were reactions to the available bitrate.

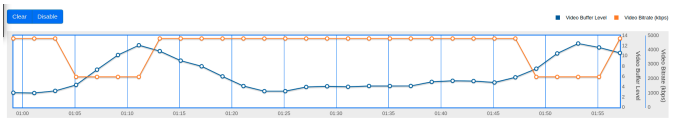


Fig. 2: 30 second fluctuation 100kbps under stable rate

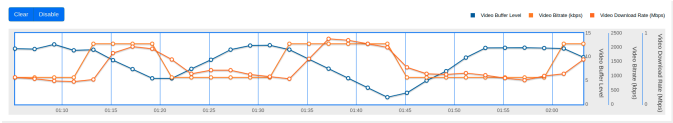


Fig. 3: about 10 second fluctuation 300kbps under stable rate

B. HLS

As the bitrates of the videos were the same between HLS and DASH also the thresholds of needed bandwidth were mostly the same with the exception of the best quality of 4794 kbps 1920x1080. With DASH the quality needed 5500kbps but with HLS the client wouldn't load the quality until the bandwidth was increased to 9000kbps, after which it stayed there even when lowering the bandwidth back to 8000kbps. The other thresholds 2600, 1250, 800 and 400 from listing IV-A were quite accurate when lowering the available bandwidth of the player as the video quality stayed the same without any lagging up to those points.

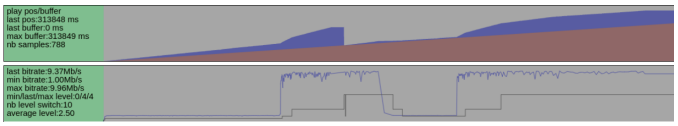


Fig. 4: HLS client's adaptive reaction speed to bandwidth change

The only fast changes that happened in just seconds were when the bandwidth was significantly increased and the client would hop to the next quality in just a couple of seconds. Otherwise the changes were either pretty slow taking close to 10 seconds when the change wasn't that big but increased the bitrate over the threshold by large enough margin. If the margin wasn't big enough, like with 1080p quality 1000kbps over the stable 8000kbps, the client would not even try change the quality as the DASH client did. A good example of the HLS client's ability to adapt at different speeds to how much the bandwidth got increased is shown in figure 4. In the figure the bandwidth was increased from 500 to 10 000 kbps and the first two changes happen in one and four seconds and the third change happens even slower after 14 seconds behind the others. After that the bitrate was changed to 1100 and again raised to 10 000 kbps and took the same four and 14 seconds to adapt.

V. Conclusion

This report aimed to implement a simple adaptive streaming server so that we could test and analyze two different adaptive video streaming protocols DASH and HLS under different network conditions. We set up local servers to serve videos via the protocols and then analyzed the playback of those said videos with browser based video player clients.

In the end Dash and HLS are very different adaptive streaming protocols. Dash is much more reactive to changes in the available network bandwidth and will try to start using higher qualities of a video before it has gained a bitrate with which it can sustain streaming for much longer with that quality. This is probably in anticipation that the bandwidth will keep increasing. HLS on the other hand waits longer for the bandwidth to increase a certain margin over the required bitrate it needs to stream the video, to ensure that the quality change wont be just a temporary hop but rather a longer lasting change. It also adapts how much time it takes before changing the quality to a better one according to the amount of bandwidth that is over the required threshold for a specific quality.

References

[1] FFmpeg Developers, "ffmpeg tool." [Online]. Available: http://ffmpeg.org/

[2] (2018) Jacob + katie schwarz: Costa rica in 4k 60fps hdr (ultra hd). [Online]. Available: https://www.youtube.com/watch?v=LXb3EKWsInQ

[3] GPAC, "MP4Box." [Online]. Available: https://gpac.wp.imt.fr/mp4box/

[4] R. Arutyunyan, "nginx rtmp module," https://github.com/arut/nginx-rtmp-module, 2019.

[5] S. Ramírez, "nginx rtmp module," https://hub.docker.com/r/tiangolo/nginx-rtmp/, 2018.

[6] D. I. Forum, "dash.js," https://github.com/Dash-Industry-Forum/dash.js/, 2019.

[7] vide dev.org, "hls.js," https://github.com/video-dev/hls.js/, 2019.

APPENDIX

Listing 3: Script used for build and deployment automation

```
#!/bin/bash

confirm() {
    read -r -p "${1:-Are you sure? [y/N]} " response
    case "$response" in
        [yY][eE][sS]|[yY])
            true
            ;;
        *)
            false
            ;;
    esac
}

container() {
    echo $(docker ps | grep $1 | awk {'print $1'})
}

remove_container() {
    CONTAINER=$(container $1)
    docker stop $CONTAINER
    docker rm $CONTAINER
}

cwd=$(pwd)
cd $(dirname $0)

IMAGE=adaptive_streaming
while true; do
    remove_container $IMAGE 2> /dev/null
    if docker build -t $IMAGE . && \
        docker run -p 8080:8080 -p 1935:1935 \
        -d $IMAGE ; then
        echo "Container running"

        sources=("dashvideo/." "hlsvideo/.")
        destinations=("/var/lib/stream/dash/" "/var/lib/
            ↪ stream/hls/")
        c=$(container $IMAGE)

        for i in "${!sources[@]}"; do
            src=${sources[$i]}
            dst=${destinations[$i]}
            if confirm "Do you want to copy files from $src
                ↪ to $c:$dst? [y/n]" ; then
                echo "Copying files"
                docker cp $src $c:$dst
            fi
        done

        break;
    elif confirm "An error occurred while building image.
        ↪ Would you like to try again? [y/n]" ; then
        continue;
    else
        echo "Build failed but retry was rejected. The
            ↪ current running containers are:"
        docker ps
        break;
    fi
done

cd $cwd
```

Listing 4: Commands for HLS video

```
$ ffmpeg -i costaRica1080p60fpsHDR.mp4
-c:a copy -c:v libx264 -flags +cgop -g 60
-b:v 5300k -maxrate 5300k -bufsize 2650k
-vf 'scale=-1:1080' -hls_time 1
-hls_playlist_type vod -hls_segment_filename
hls/1080p60/1080p_%03d.ts hls/1080p60/manifest.m3u8
$ ffmpeg -i costaRica1080p60fpsHDR.mp4
-c:a copy -c:v libx264 -flags +cgop -g 60
-b:v 2400k -maxrate 2400k -bufsize 1200k
-vf 'scale=-1:720' -hls_time 1
-hls_playlist_type vod -hls_segment_filename
hls/720p60/720p_%03d.ts hls/720p60/manifest.m3u8
$ ffmpeg -i costaRica1080p60fpsHDR.mp4
-c:a copy -c:v libx264 -flags +cgop -g 60
-b:v 1060k -maxrate 1060k -bufsize 530k
-vf 'scale=-1:478' -hls_time 1
-hls_playlist_type vod -hls_segment_filename
hls/480p60/480p_%03d.ts hls/480p60/manifest.m3u8
$ ffmpeg -i costaRica1080p60fpsHDR.mp4
-c:a copy -c:v libx264 -flags +cgop -g 60
-b:v 600k -maxrate 600k -bufsize 300k
-vf 'scale=-1:360' -hls_time 1
-hls_playlist_type vod -hls_segment_filename
hls/360p60/360p_%03d.ts hls/360p60/manifest.m3u8
$ ffmpeg -i costaRica1080p60fpsHDR.mp4
-c:a copy -c:v libx264 -flags +cgop -g 60
-b:v 260k -maxrate 260k -bufsize 130k
-vf 'scale=-1:242' -hls_time 1
-hls_playlist_type vod -hls_segment_filename
hls/240p60/240p_%03d.ts hls/240p60/manifest.m3u8
```

Listing 5: manifest.m3u8

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-STREAM-INF:BANDWIDTH=260000,RESOLUTION=430x242
240p60/manifest.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=600000,RESOLUTION=640x360
360p60/manifest.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=1060000,RESOLUTION=850x478
480p60/manifest.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2400000,RESOLUTION=1280x720
720p60/manifest.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=5300000,RESOLUTION=1920x1080
1080p60/manifest.m3u8
```

Listing 6: Apache2 Configuration

```
<VirtualHost *:80>
ServerAdmin admin@sampledomain.com
ServerName sampledomain.com
ServerAlias www.sampledomain.com
DocumentRoot /var/www/sampledomain.com/html
ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```