Live streaming latency of streaming protocols

Max Crone

max.crone@aalto.fi

Jack Henschel

jack.henschel@aalto.fi

***Abstract*—In this report we set up a live streaming server using a combination of stateless and stateful protocols, based on which we evaluate the performance of these different protocols with regards to latency. As expected, we find that a stateful protocol like RTMP has the lowest latency while suffering from scalability limitations. Stateless protocols, such as HLS and DASH, exhibit a higher latency but are easier to scale, e.g. by using a CDN which we also analyzed. Specific fine-tuning of the streaming parameters for the stateless protocols is found to bring down their end-to-end-latency.**

***Index Terms*—video, live, streaming**

I. Introduction

Over the last decade live streaming has become extremely popular following the success of video-on-demand services. Nowadays, there are many professional live streaming services on the Internet (e.g. broadcasts from sports events) but also a large amount of streaming services for amateurs (e.g. Twitch), though the distinction between these is becoming increasingly blurred. In addition to the goals of video-on-demand (quality of experience, delivery cost), live streaming has another important metric: latency.

The lower the latency is, the more the audience is engaged and feels part of the event. Especially for applications that contain feedback mechanisms, such as live chat among multiple users and the stream source producer, the quality of engagement is extremely important. Thus, we will closely examine this special metric across multiple streaming protocols in this report.

There exist two different types of streaming protocols: stateful and stateless protocols. Stateful protocols, of which RTMP is an example, provide lower latency compared to stateless protocol, but scale poorly with increasing number of users. RTMP in particular pushes new video data to clients in real time, resulting in a larger CPU utilization and thus increased delivery costs for the operator. [1]

Stateless protocols, like DASH and HLS that are both based on HTTP, require the client to periodically pull new video data from the server. Though with HTTP/2 server push it is possible to directly transfer segments from the server to the client, the underlying concepts of chunked encoding is still the same. DASH, or the MPEG-DASH standard to be precise, is an open format for adaptive bitrate streaming. The DASH manifest is commonly stored in a `.mpd` file (Media Presentation Description) and is formatted as XML. New video segments are added to this manifest file by the server and the client fetches these new segments from the server, like a playlist. HTTP Live Streaming (HLS) is a streaming video protocol developed by Apple Inc. Just like DASH, HLS uses HTTP transactions which traverse firewalls, proxies, and can be distributed through CDNs with ease. Therefore, both DASH and HLS technology are able to reach a much larger viewing audience than RTMP or other any other streaming protocol. Most of the live streaming video online today is done via HTTP.

II. Experimentation setup

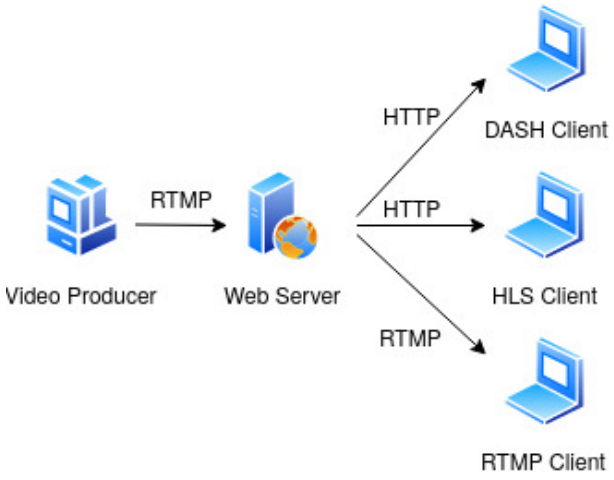

Fig. 1. Overview of live streaming setup with RTMP, DASH and HLS

Nginx is an extremely versatile HTTP web server, reverse proxy, load balancer and HTTP cache and is available as Free Software [2]. Most importantly for us, its functionality is easily extensible by using “modules”. To serve the video stream to our clients, we set up an Nginx web server (version 1.14) with the RTMP module on a Google Cloud Compute instance of type `n1-standard-1` with 1 vCPU, 3.75 GB main memory and a 10 GB Debian Buster (10.2) disk image. [3]

This gives us full control over the web server and streaming configuration. The round-trip time (RTT) to the server located in a datacenter in Finland is 32 milliseconds on average, with a standard deviation of 4.7 milliseconds, from our vantage point at the Aalto University, Espoo. We use these results to establish that the round-trip time will be negligible in the latency results for the rest of the report.

The full Nginx configuration file is shown in Figure 11. In the `rtmp` block we configure the RTMP server to listen on port 1935, the IANA standard port for RTMP connections. Then, in the `application` block we allow all clients to stream our video without authorization (line 16) and enable

Fig. 2. RTMP Stream Ingestion

```
ffmpeg -re -i $VIDEO_FILE \
$CODEC_OPTIONS -f flv \
rtmp://example.com/live/stream-key
```

Fig. 3. Viewing the Video Stream

```
# Stream HLS
mpv http://example.com/hls/stream-key/index.m3u8

# Stream DASH
mpv http://example.com/dash/stream-key/index.mpd

# Stream RTMP
mpv rtmp://example.com/live/stream-key
```

live streaming mode (line 17). Next, we set up HLS streaming (line 19), configure a path where Nginx can store the segments (line 21) and set the segment size to 5 seconds (line 22). We repeat the same for DASH streaming (line 23-27) and finally disable permanent storage of the video segments (line 29).

It is important to set the correct Cross-Origin Resource Sharing (CORS) headers for the web server, otherwise modern web browser will refuse to load the video segments (line 64-74). For DASH and HLS, the "Content-Type", "Origin" and "Range" headers need to be allowed for HTTP GET, HEAD and OPTIONS methods.

This setup allows RTMP for ingesting live video streams (Figure 2) and RTMP, DASH as well as HLS for viewing the video stream (Figure 3). It is important to allow TCP traffic on port 1935 in the host's firewall to allow RTMP streaming.

In our test setup we used FFmpeg version 4.1 (`4.1.4-1 deb10u1`), MPV version 0.29.1 and VLC media player 3.0.8 (Vetinari, revision `3.0.8-0-gf350b6b5a7`).

To conduct our latency measurements, we used two laptops side-by-side (Figure 4) The left one is streaming the video file to the server and at the same time playing the video on-screen. The right one is playing the three video-streams from the server (from left to right: HLS, DASH, RTMP). We then record a video of both laptops and analyze the time differences between scene changes in video editor. This setup allows us to precisely determine the end-to-end latency of all involved components: stream ingestion, processing on the server, stream download and playback client-side.

For our tests we used a standard test video with easily identifiable scene changes, resolution of 1920x1080 pixels and 25 frames per seconds. [4] We streamed this video with FFmpeg to the server. The exact command is shown in Figure 12. We also experimented with using live footage from a webcam, but that made it more difficult to precisely measure the delay (Figure 13). Therefore we based our experiments on the test video instead.

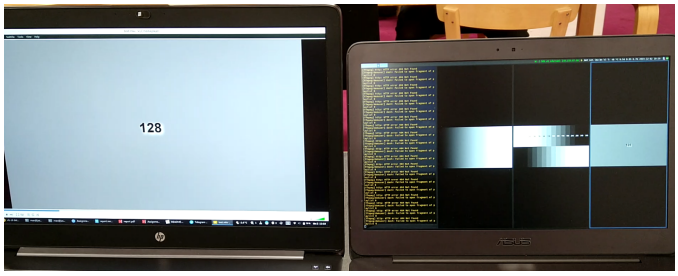

Fig. 4. End-to-end latency measurement setup

III. Analysis

As outlined in the introduction, one of the most important parameters for live video streaming are the keyframe interval together with the segment size (for HTTP streaming). The keyframe interval specifies how often a keyframe is inserted into the video stream. Keyframes contain the entire picture of a video without reference to any other frames of the video. As they incorporate all of the information about the pixels in each image, they take up a lot more space and are much less frequent than the other types of encoding frames. After encoding, the video needs to be divided into short segments where each segment has to start with a keyframe. The keyframe interval is commonly set through the "group of pictures", or GOP, parameter. A GOP in a compressed video stream means that the decoder doesn't need any previous frames in order to decode the next ones.

The rest of this section will report on three different experiments. We first analyze the effect of varying segment size and GOP size on the latency for all protocols. Secondly, we determine the impact a CDN has when leveraging it our live streaming. Lastly, we analyze how the latency develops over time in a live stream.

TABLE I
Test Scenarios

Scenario	Segment Size	GOP (frames)	CDN
Baseline	5s	50	No
2	1s	25	No
3	5s	125	No
4	10s	250	No
5	15s	375	No
With CDN	5s	50	Yes

1) GOP and Segment Size: The segment size for DASH and HLS chunks should be a multiple of the GOP size in order to guarantee that each segment can be decoded individually. Otherwise, if a segment gets delayed or lost the next segment may not be decodeable. We decided to set the GOP size and segment size equal to each other for our main scenarios. Because the GOP is expressed in frames instead of seconds, a size of 25 means that it is set to 1 second, because the video we used is encoded at 25 frames per second.

We varied these two parameters from very small values (1 second) to large values (15 seconds) and conducted our mea-

surements. The resulting latencies of the different technologies are plotted in Figure 5.

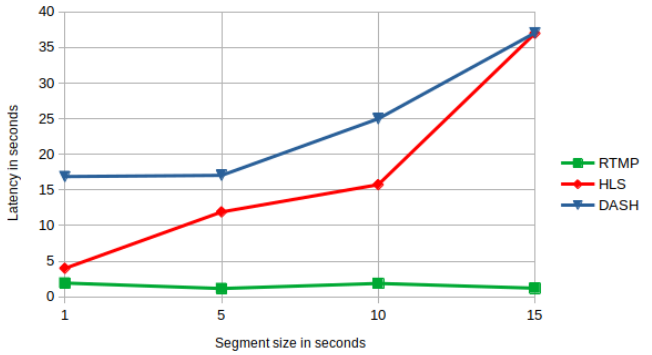

Fig. 5. End-to-end latency measurements for three different streaming protocols.

The first observation is that segment size and GOP do not affect the latency of RTMP streaming. This confirms our expectation since the streaming server is just relaying the individual packages it receives from the streaming source to the clients. Thus there is no buffering or processing involved and the latency is consistency very low, 1.56 seconds on average.

On the other hand, the latency of DASH and HLS streaming strongly correlates with the two parameters. With a segment size of 1 second we were able to achieve an end-to-end latency of 4 seconds with the HLS client and 16.89 seconds with the DASH client. With increasing segment size also the latency increased. For lower segment sizes HLS seems better suited than DASH, while for a segment size of 15 seconds the latency of these two converges to nearly the same value, about 37 seconds.

The increasing latency matches our expectation. When the stream starts the server first has to buffer the incoming video stream before it can produce the first chunk (e.g. 10 seconds). Then, when the clients downloads and plays this segment, the server is already generating the next segment. But since only a complete segment can be downloaded from the server, the client is always lagging behind. When the segment sizes become larger, these offsets in latency therefore naturally also begin to build up.

This is also reflected in the fact that during our experiments we observed that increasing the segment size also increases the stream startup latency, i.e. the time between when the streaming source starts uploading the video and the time when the first segment is available for the clients. Which of course is also due to the fact that the server and the streaming source first need to create a complete segment before it can be streamed by clients.

While short segments are good to quickly adapt to bandwidth changes and prevent stalls, longer segments have a better encoding efficiency and quality. As Bitmovin found in their research, small segments, such as 1 second ans below should generally be avoided since the PSNR ("perceived signal to noise ratio", i.e. subjective quality of the video) decreases quickly. [5]

In our experiments we identified the following sources of latency that contribute to the end-to-end latency.

- video stream producer encoding
- upload from video producer to server
- server buffering and processing
- client video segment download
- client video buffering

Depending on how the client is configured, the client-side buffering can have a major effect on the total latency. Having a large playback buffer on the client-side generally enhances the smoothness of video playback, i.e. if there are network issues the playback can still continue from the buffer. However, since this buffer first has to be filled (e.g. 10 seconds) it also adds the buffer length as additional latency.

2) Usage of CDNs: Another source of latency that we wanted to evaluate is the use of a CDN ("content distribution networks"). CDNs allow website operators to bring their content closer to their users by caching content on CDN proxies. In additions it also allows a CDN's customers to distribute their content worldwide and to reduce the load on their own web servers.

In order to investigate its effect on the video streaming latency, we set up a Cloudflare CDN proxy in front of our web server. The HTTP streaming clients no longer directly access our web server, but rather go to Cloudflare's proxies which in return fetch the video segments and metadata from our server. The RTMP client still directly accesses our server since Cloudflare only proxies HTTP-based connections. This led us to remove the RTMP measurements from the results, as it is not different from the earlier experiments and thus would not add anything of analytical value.

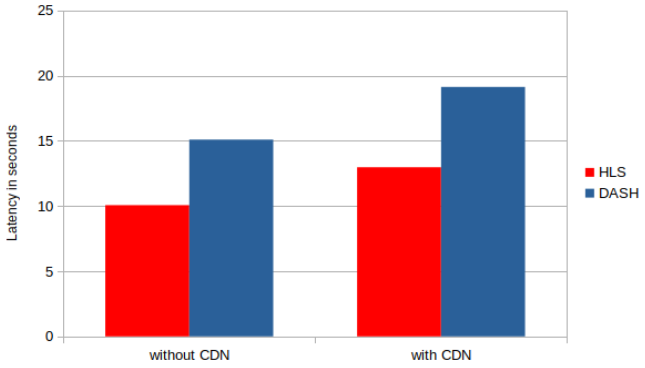

Fig. 6. End-to-end latency measurements for the baseline case with and without CDN.

Figure 6 presents the resultant latency for the baseline case as specified in Table I compared to the latency for both streaming clients for the scenario with the Cloudflare CDN in between.

The results clearly show that the streaming latency increases for both clients when a CDN is used, compared to a scenario where the clients could directly access the server. This is

caused by the fact that the placement of the CDN in this case merely adds extra network hops in between the server and the clients. For a scenario with a single client, by adding extra nodes on its connection to the server the end-to-end latency can only increase. Similar to the principle of triangle inequality. In that sense, we are not using the CDN to its full potential. It will only provide an advantage when many different clients are streaming a video stream that is cached by the CDN. In that case the CDN can return the same segments that are being requested by all the clients, without having to query the streaming server itself more than once for a segment. We did however not replicate these tests with a larger number of clients. This is something that could be done in future work.

3) Long-running streams: As a final test we observed the latency of all video streaming protocols over time for a longer streaming duration. Hypothetically, these results should not differ from the observations we made earlier that were only based on single measurements at the start of the stream. The latency is expected to remain relatively stable over time, because no extra burdens will be introduced. The GOP size was set to 75 frames, or three seconds, and the segment size for HLS and DASH was set to 6 seconds.

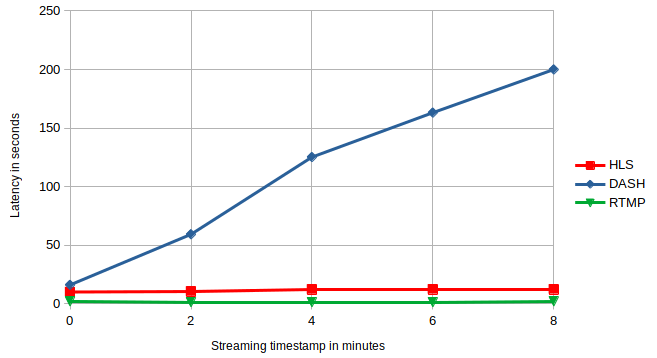

Fig. 7. End-to-end latency measurements for three different streaming protocols over time.

Figure 7 presents the results over an eight-minute time-frame. Both RTMP and HLS conform to our hypothesis. However, we observe a strongly increasing latency for the consumption of the livestream by the DASH client. The first latency observation of DASH that we made is similar to what we expected based on our observations in earlier tests. All measurements that we took later exhibit the pattern of increasing latency.

Since these results deviate surprisingly much from our expectations, we repeated the experiment over a longer time frame. However, the results, shown in Figure 8, were still the same.

We also experimented with disabling client side buffering in MPV (with the `--cache=no` parameter), the results are depicted in Figure 9. In this case we could see lower initial latency values for DASH, 10.42 seconds without buffer compared to 18.59 seconds with buffer. Nevertheless, the latency of DASH quickly accumulated to values of multiple minutes.

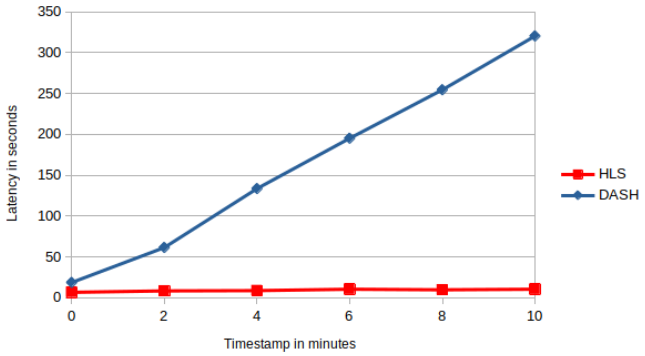

Fig. 8. Repeated, end-to-end latency measurements for DASH and HLS over time.

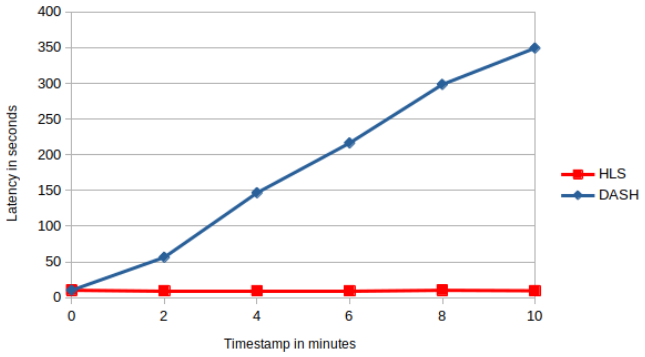

Fig. 9. End-to-end latency measurements without client buffer for DASH and HLS over time.

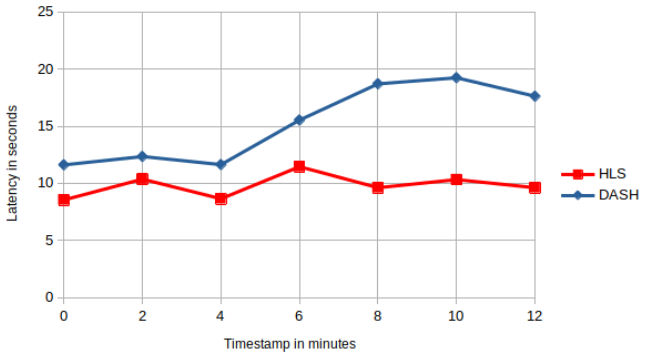

Fig. 10. End-to-end latency measurements with VLC for DASH and HLS over time.

We could not come up with a good explanation for this behavior except that the DASH implementation in MPV is erroneous, i.e. failure to produce non-gapless playback between segments. Thus we decided to test the DASH streaming with another popular media player, VLC. The long term results are shown Figure 10. The results of both DASH and HLS also behave the way we expected: the latency varies slightly over time (few seconds) and does not drastically increase. But again we can see that overall HLS performs slightly

better (average of 9.8 seconds) than DASH (15.2 seconds), leading us to the conclusion that HLS is better optimized for live streaming, hence the name "HTTP Live Streaming", than DASH. Additionally, we also saw that using VLC for streaming had lower CPU utilization than MPV which might also be related to MPV's increasing latency.

Further optimization of DASH delivery, for example changing DASH manifest attributes such as `availabilityStartTime` and `minBufferTime`, could further improve the end-to-end latency. [6] However, this was outside of the scope of this report.

IV. Conclusion

In this report we set up our own live streaming server and investigated the performance of a set of stateful (RTMP) and stateless (HLS, DASH) streaming protocols with regard to latency. As expected, using RTMP for the client consistently offers the lowest end-to-end latency. HLS and DASH streaming are an order of magnitude "slower" compared to RTMP, depending on the exact configuration. However, since they are based on HTTP, they provide an easy solution to distribute the video data among thousands of users by employing a CDN in front of the origin server. RTMP on the other hand struggles to scale to so many clients due to the stateful nature of the protocol.

Another disadvantage of the RTMP protocol is that it is relatively obscure in the internet landscape and therefore using it may not be possible in some networks due to firewall restrictions.

HLS and DASH stay relatively stable in their latency over time, except for the MPV media player, where we discovered that there must be an error in the DASH client, because the latency built up way beyond normal behavior. Additional experiments showed that this erroneous behavior does not occur in other media players.

We also showed that with some tuning of the streaming parameters on the producer, the server and the client side, it is possible to bring the latency of stateless protocols closer to the level of stateful ones.

When choosing a streaming protocol it is up to producers and distributors to find a balance between quality of experience, latency and operating cost depending on their particular use-case. If the intended audience will be small and latency minimization is important, then we recommend RTMP. In all other circumstances we recommend the use of HLS, because this protocol provides the scalability advantages of HTTP streaming and performs with a lower latency compared to DASH, based on our experiments.

References

[1] Bolun Wang et. al., "Anatomy of a Personalized Livestreaming System", Proceedings of the 16th ACM SIGCOMM Internet Measurement Conference (IMC) , 2016.

[2] "nginx", http://nginx.org/en/, 2019.

[3] Roman Arutyunyan, "NGINX-based Media Streaming Server: nginx-rtmp-module", https://github.com/arut/nginx-rtmp-module, 2017.

[4] Florin1up, "Full-HD Testcharts", https://www.youtube.com/watch?v=tPdLZ9g0l28, 2014.

[5] Stefan Lederer, "Optimal MPEG-DASH & HLS Segment Length", https://bitmovin.com/mpeg-dash-hls-segment-length/, 2015.

[6] Romain Bouqueau, "Diving into ultra-low latency for live using MPEG-DASH", https://www.gpac-licensing.com/2014/07/09/lowering-dash-live-latency-240ms/, 2014.

Fig. 11. Nginx web server configuration

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;

events {
 worker_connections 768;
}

rtmp {
 server {
  listen 1935;
  chunk_size 4096;
  application live {
   # enable live streaming for all clients
   allow play all;
   live on;
   # enable and configure HLS
   hls on;
   hls_nested on;
   hls_path /var/www/hls;
   hls_fragment 5s;
   # enable and configure DASH
   dash on;
   dash_fragment 5s;
   dash_path /var/www/dash;
   dash_nested on;
   # do not store stream permanently
   record off;
  }
 }
}

http {
 sendfile on;
 tcp_nopush on;
 tcp_nodelay on;
 keepalive_timeout 65;
 types_hash_max_size 2048;
 include /etc/nginx/mime.types;
 default_type application/octet-stream;
 types {
  application/vnd.apple.mpegurl m3u8;
 }
 access_log /var/log/nginx/access.log;
 error_log /var/log/nginx/error.log;
 gzip on;

 server {
  listen 80;
  root /var/www/;
  index index.html index.htm;
  server_name _;

  # show nginx connection status
  location /stats {
   rmtp_stat all;
   rmtp_stat_stylesheet stat.xml;
  }

  location / {
   # disable cache
   add_header Cache-Control no-cache;
   # CORS setup
   add_header 'Access-Control-Allow-Origin' '*' always;
   add_header 'Access-Control-Expose-Headers'
              'Content-Length,Range,Content-Type,Origin';
   # allow CORS preflight requests
   if ($request_method = 'OPTIONS') {
    add_header 'Access-Control-Allow-Origin' '*';
    add_header 'Access-Control-Max-Age' 1728000;
    add_header 'Content-Type' 'text/plain charset=UTF-8';
    add_header 'Content-Length' 0;
    return 204;
   }
  }
 }
}
```

Fig. 12. Video Streaming with FFmpeg

```
ffmpeg -re -i $VIDEO_FILE \
-c:v libx264 -preset veryfast \
-maxrate 3000k -bufsize 6000k \
-pix_fmt yuv420p -g 50 \
-c:a aac -b:a 160k -ac 2 -ar 44100 \
-f flv rtmp://example.com/live/stream-key
```

Fig. 13. Webcam Streaming with FFmpeg

```
ffmpeg -re -f v4l2 -framerate 30 \
-video_size 1280x720 -i /dev/video0 \
-c:v libx264 -preset veryfast \
-maxrate 3000k -bufsize 6000k \
-pix_fmt yuv420p -g 50 \
-f flv rtmp://example.com/live/stream-key
```