Live streaming

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

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

***Abstract*—Live streaming has been a popular form of entertainment for quite some time. Many things such as sporting events and comedy shows are streamed live for people to enjoy in real time. In the recent years, social media has also seen rapid adoption of live streaming capabilities in their platforms. Video live streaming brings challenges to upholding an acceptable QoE for the users. In this paper, we set up a live streaming server and evaluate its performance with different configurations and protocols, and present pros and cons of different setups.**

***Index Terms*—live streaming, nginx, cdn, rtmp, hls, aws, cloudfront**

I. Introduction

Live streaming over the internet has become popular over the recent years. Live video streaming is popular in fields such as gaming and sports. The practice of streaming live video has been around for quite some time, but in the recent years, social media has made it possible for a large amount of users to live stream video themselves over the internet.

Video live streaming is seeing rapid adoption on many platforms, such as Facebook and Snapchat, which launched live streaming capabilities on their platforms in 2015 and 2014 respectively [1]. According to a study [1] conducted in 2017, people find video live streaming a very engaging form of entertainment, which suggests that live streaming is here to stay.

Different methods for live streaming exist, but some of them perform differently than others, and each of them provide their own set of pros and cons. In this paper, we experiment with live streaming and discuss the performance of different live streaming systems and the factor of protocols, as well as the system design.

A. Used protocols

In the experiment, our live streaming setup employed two different protocols: Real Time Messaging Protocol (RTMP), and HTTP Live Streaming (HLS).

RTMP is a protocol designed for streaming media on top of Transmission Control Protocol (TCP). RTMP has some variations, such as RTMPS, which is TLS encrypted RTMP. For our experiment, we use plain RTMP. RTMP is a stateful protocol, which allows low-latency streaming, with latencies generally around 5 second, with some variation.

HLS, as the name suggests, is an HTTP-based streaming protocol. The protocol is designed for steady streaming. It works by segmenting, encoding and buffering the the stream before publishing it. Generally, HLS has a streaming latency around 30 seconds.

II. Experimentation Setup

Our goals were to first setup a live video streaming system and then evaluate the latency with different configurations. First we setup a local version of the system after which we deployed it to the AWS cloud.

A. Local Setup

The local live streaming system consists of three main components: a video streamer, a streaming server, and live stream viewers.

We found the Open Broadcaster Software studio (OBS) an easy way of setting up the streaming client. OBS allows the use of various sources such as webcam, screen sharing, and static video. OBS is also used by a large amount of streamers on platforms such as Twitch and YouTube. The streaming software was configured as a video stream input for the server. The configuration required choosing a custom stream service, and specifying the server's address, port, path and a stream key. The configuratios are shown in table I. The path '/hls' is defined on the server as the streaming application's name, and it can be reached by the streaming client with RTMP on the default port 1935.

Service	Custom...
Server	rtmp://0.0.0.0:1935/hls
Stream key	stream

TABLE I: OBS local setup stream settings

The server was set up using *nginx*. Nginx is a technology, often used as a web server or a load balancer. For the setup, we used nginx together with nginx-rtmp [2], which is an nginx module for media streaming with ready-made support for RTMP and HLS streaming protocols.

With the configuration in appendix' figure 4, we ran the streaming server on a local machine. The server was ran in a *Docker* container to minimize dependencies to the host machine, and to enable easy deployment regardless of the host operating system. The container was based on an image [3] with nginx, and nginx-rtmp readily installed. The container was mounted with the custom server configuration from figure 4 by using the Dockerfile from listing 5, and running the 'start.sh' script from listing 4. This script builds and runs the container.

Nginx was configured to transform the client stream into multiple bitrates using *ffmpeg*. This enables optimization of the viewed stream, and faster delivery to clients over HLS. Moreover, processing of the live stream with ffmpeg is generally not very processing-heavy, and can improve QoE by providing more optimized content delivery.

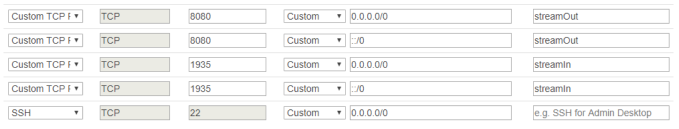
As stated before the streaming client uses a stateful RTMP protocol to input the stream to the server. The server in turn uses both stateful and stateless protocols namely RTMP and HLS for serving the stream to viewers. This is made possible with nginx-rtmp module which produces the HLS segments and manifest from the stream as it comes in.

For viewing the stream we provided simple client players that run on the browser. The clients use a JavaScript library, *videojs*, and only require the viewer to have a modern browser with the support for the protocols or ability to add them if required. The RTMP stream viewing client requires Flash to run, which might make it tedious to run on certain browsers, and required an additional videojs library, *videojs-flash*, for the player to work.

B. Cloud Setup

In our cloud deployment, we used an Amazon Web Services (AWS) free-tier EC2 instance for the streaming server, which was running our *dockerized nginx* server on Ubuntu LTS.

To allow connections to our EC2 instance we configured security rules to a security group to allow RTMP and HTTP connections to ports 1935 and 8080 as we had configured on our *nginx* server. These rules are shown in figure 1.



Custom TCP F ▾	TCP	8080	Custom ▾	0.0.0.0/0	streamOut
Custom TCP F ▾	TCP	8080	Custom ▾	::/0	streamOut
Custom TCP F ▾	TCP	1935	Custom ▾	0.0.0.0/0	streamIn
Custom TCP F ▾	TCP	1935	Custom ▾	::/0	streamIn
SSH ▾	TCP	22	Custom ▾	0.0.0.0/0	e.g. SSH for Admin Desktop

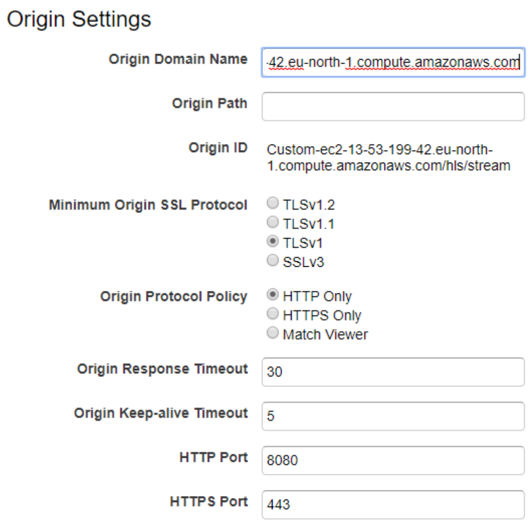
Fig. 1: AWS EC2 security rules

We also configured an SSH rule so we could access the server to install docker with an installation script found in listing 1 and clone our git repository, whose contents are presented in appendix' figure 5, on to the server and setup and start the server by running the 'start.sh' script (4) as in the local setup which starts the docker build and runs it. Ffmpeq software also needed to be installed to the docker image but this was taken care of in the Dockerfile (5).

Listing 1: Docker installation

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

To further experiment with live streaming, we set up a Content Delivery Network (CDN) in AWS, using the free-tier AWS Cloudfront deployment. We created a Web distribution and configured our EC2 instance as the origin for it to distribute the HTTP content. The CDN was setup with mostly default configurations as presented in appendix' figure 6 except specifying the origin address and non default port 8080 for the input. The settings for the origin are presented in figure 2.



Origin Settings

Origin Domain Name	42.eu-north-1.compute.amazonaws.com
Origin Path	
Origin ID	Custom-ec2-13-53-199-42.eu-north-1.compute.amazonaws.com/hls/stream
Minimum Origin SSL Protocol	TLSv1.2 / TLSv1.1 / [✓ TLSv1] / SSLv3
Origin Protocol Policy	[✓ HTTP Only] / HTTPS Only / Match Viewer
Origin Response Timeout	30
Origin Keep-alive Timeout	5
HTTP Port	8080
HTTPS Port	443

Fig. 2: CDN video streaming origin settings

The architecture of our AWS deployment is described in figure 3. The streaming client is an arbitrary machine streaming to our EC2 server, with the streaming URL from 2. A stream viewer could stream content from either the streaming server using HLS or RTMP, or via the Cloudfront CDN over HLS.

Listing 2: EC2 streaming server

```
rtmp://ec2-13-53-199-42.eu-north-1.compute.amazonaws.com
    ↪ :1935/hls/stream
```

The client programs from our local setup were also provided from the cloud streaming server, as well as the CDN. This made it possible to view the live stream by simply visiting the URL of the server. The CDN existed in URL from listing 3.

Listing 3: AWS Cloudfront CDN URL

```
dzpalxnsxqao8.cloudfront.net
```

III. Measurement and Analysis

We experimented with both local and the cloud deployment to get an idea of how different parameters affect the live streaming performance. First, we conducted experiments on the local environment.

We configured OBS to stream full HD (1920x1080) video at 30 fps, which was the maximum quality supported by our nginx configuration. The end-to-end latency on the local environment was around 2 seconds for the RTMP viewing client, with no noticeable variation. For the local HLS live stream with the same setup, the latency was around 10 to 15 seconds.

Running OBS on a 6-core Intel i7 processor, was quite heavy, and the laptop seemed to heat up quite quickly. The

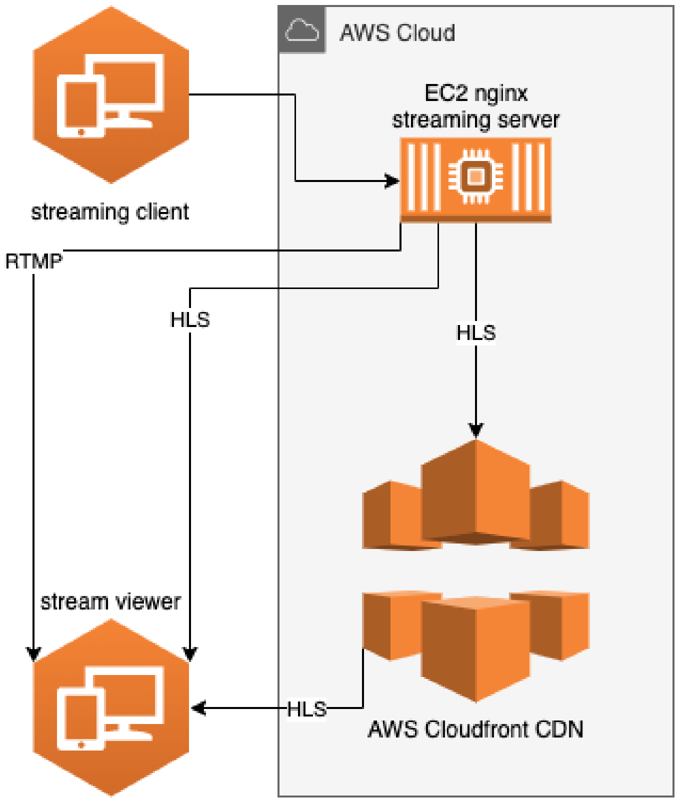


Fig. 3: Live video streaming AWS setup

CPU was running at 100% constantly when streaming, which could affect the performance of other components of the local streaming platform. CPU seemed to be the only bottleneck. Memory was sufficient, so there was no disk read/write latency caused by swap memory.

For the cloud deployment, the experiment provided large latency variations in HLS. When streaming with RTMP (see figure 3), the streaming latency seemed almost constant, much like with the local setup. We compared latencies using different machines: smart phones, and laptops. We set the viewing clients next to each other, and observer how the latencies varied by streaming a stopwatch and comparing the stopwatch values on different devices. The largest variation was about 1 second for the RTMP stream. However, HLS streaming latencies varied quite a lot. We tested with different machines like with RTMP, and observed the latency differences between machines. With larger initial bitrates, the stream seemed to halt quite easily. This is most likely because of the resource constraints of the EC2 streaming server (free tier). With HD ready video (around 2500 Kbps), the instance seemed to be able to handle the stream quite well. The startup time of the CDN was around 40-50 seconds, meaning the EC2 instance was able to stream with HLS 40-50 seconds before the CDN. Comparison between the streaming latencies, right after the streaming has started, are shown in table II. After this initial startup latency, the CDN and plain HLS stream seemed almost identical in end-to-end latency. In some tests, the CDN viewer received frames slightly before the HLS viewer. This is most likely due to network anomalies as we were unable to reproduce the behaviour on the local environment.

CDN + HLS	$41.165s - 0.496s = 40.669s$
HLS	$41.165s - 22.957s = 18.208s$
RTMP	$41.165s - 35.822s = 5.343s$

TABLE II: End-to-end latencies of different stream viewing methods on startup

The measurements from II are on-par with the *nginx* streaming server setup in 4: The HLS playlist length is set to 30 seconds by default, which translates to a 30 second startup latency for the CDN, as it can only start receiving ts-fragments 30 seconds after the streaming client has started the streaming. Additional latencies to the startup can be explained with stream encoding and decoding, as well as network delays and playback buffer fillings. We tested the nginx server with lower playlist lenghts and verified the startup latency correlates with the playlist length. After the startup time, the CDN nodes will fill up their video buffers, and after some time it will be almost identical with the nginx streaming server state, with small network and buffering latencies.

While streaming, the EC2 instance was not able to deliver content to more than 7 concurrent clients viewing the stream. Additional clients did not receive streaming data from the server. This is most likely because of the resource constraints of the instance, as well as the required bandwidth for the single instance serving the stream. Here we really see the benefits of distributing content over a CDN. There is no single point of failure, as the network is able to serve the content from multiple endpoints throughout the CDN. We tested with 15 concurrent connections, and the CDN seemed to be able to handle them quite well. The testing bottleneck was really the stream source which slowed down quite a bit because of the CPU requirements of OBS. Most likely the free-tier CDN would not be able to handle much more traffic, as it is constrained to 50GB/month traffic and 2,000,000 HTTP requests [4].

IV. Conclusion

In this report we aimed to implement a local and a cloud based live video streaming system and then measure and compare the performance of these systems in relation to how the content is consumed by viewing clients. We set up the servers to serve video stream via RTMP and HLS protocols and via a CDN, and then compared the latencies of those said streams with browser based video player clients.

A RTMP only server is good for a small amount of users, since the protocol doesn't scale to too many concurrent users. HLS in comparison scales much better as it transfers the stream to segmented media files in various encoding qualities that it serves over HTTP which has much better scalability.

Based on our experiences a local setup is good for small setups that don't require scaling to bigger number of viewers and are aimed to be used only inside a network, at least if the network doesn't have higher end internet connectivity. A setup with a single EC2 instance provides better accessibility to the stream by having the endpoint on the cloud but still is limited by the constraints of scalability of the single instance. This can of course be solved by using bigger instances with more resources but the cost will also start scaling accordingly.

CDN together with HLS is the most scalable solution for a larger user base. It enables horizontal scaling of the platform by adding more end nodes rather than processing power and single-instance resources. Using a CDN for live streaming delegates content delivery responsibilities to multiple nodes that can be scaled on a global basis. This is how the AWS Cloudfront operates, but any other CDN, proprietary or private, may be used. The end-to-end latency is not significant after the initial startup latency is over. Using a CDN for streaming can even provide lower end-to-end latency compared to a HLS only cloud server: a streaming viewer may receive content from a node in the CDN that is physically closer to their location than the origin server is, while the origin server is under minimal load. Overall, the choise between these configurations comes down to the desired use case of the platform.

References

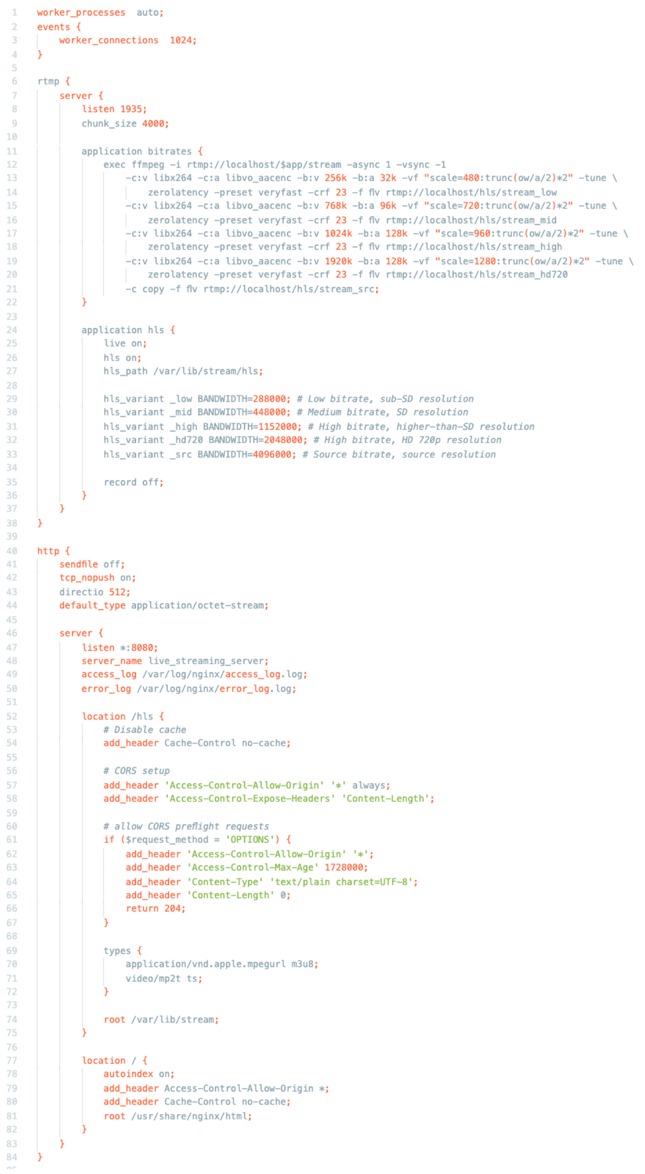
[1] O. L. Haimson and J. C. Tang, "What makes live events engaging on facebook live, periscope, and snapchat," in *Proceedings of the 2017 CHI conference on human factors in computing systems*. ACM, 2017, pp. 48–60.

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

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

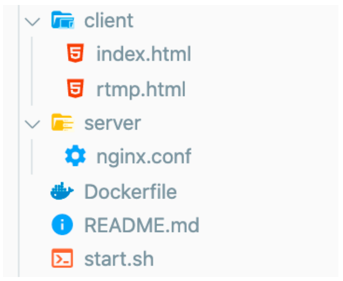
[4] (2019) Amazon cloudfront infrastructure. [Online]. Available: https://aws.amazon.com/cloudfront/features/?nc=sn&loc=2

Appendix



```
worker_processes  auto;
events {
    worker_connections  1024;
}

rtmp {
    server {
        listen 1935;
        chunk_size 4000;

        application bitrates {
            exec ffmpeg -i rtmp://localhost/$app/stream -async 1 -vsync -1
                -c:v libx264 -c:a libvo_aacenc -b:v 256k -b:a 32k -vf "scale=480:trunc(ow/a/2)*2" -tune \
                    zerolatency -preset veryfast -crf 23 -f flv rtmp://localhost/hls/stream_low
                -c:v libx264 -c:a libvo_aacenc -b:v 768k -b:a 96k -vf "scale=720:trunc(ow/a/2)*2" -tune \
                    zerolatency -preset veryfast -crf 23 -f flv rtmp://localhost/hls/stream_mid
                -c:v libx264 -c:a libvo_aacenc -b:v 1024k -b:a 128k -vf "scale=960:trunc(ow/a/2)*2" -tune \
                    zerolatency -preset veryfast -crf 23 -f flv rtmp://localhost/hls/stream_high
                -c:v libx264 -c:a libvo_aacenc -b:v 1920k -b:a 128k -vf "scale=1280:trunc(ow/a/2)*2" -tune \
                    zerolatency -preset veryfast -crf 23 -f flv rtmp://localhost/hls/stream_hd720
                -c copy -f flv rtmp://localhost/hls/stream_src;
        }

        application hls {
            live on;
            hls on;
            hls_path /var/lib/stream/hls;

            hls_variant _low BANDWIDTH=288000; # Low bitrate, sub-SD resolution
            hls_variant _mid BANDWIDTH=448000; # Medium bitrate, SD resolution
            hls_variant _high BANDWIDTH=1152000; # High bitrate, higher-than-SD resolution
            hls_variant _hd720 BANDWIDTH=2048000; # High bitrate, HD 720p resolution
            hls_variant _src BANDWIDTH=4096000; # Source bitrate, source resolution

            record off;
        }
    }
}

http {
    sendfile off;
    tcp_nopush on;
    directio 512;
    default_type application/octet-stream;

    server {
        listen *:8080;
        server_name live_streaming_server;
        access_log /var/log/nginx/access_log.log;
        error_log /var/log/nginx/error_log.log;

        location /hls {
            # 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';

            # 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;
            }

            types {
                application/vnd.apple.mpegurl m3u8;
                video/mp2t ts;
            }

            root /var/lib/stream;
        }

        location / {
            autoindex on;
            add_header Access-Control-Allow-Origin *;
            add_header Cache-Control no-cache;
            root /usr/share/nginx/html;
        }
    }
}
```

Fig. 4: nginx configuration



```
client
    index.html
    rtmp.html
server
    nginx.conf
Dockerfile
README.md
start.sh
```

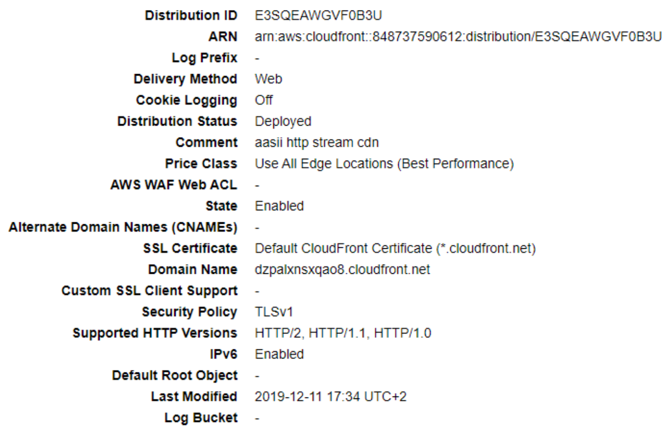
Fig. 5: Project files

Listing 4: start.sh

```
#!/bin/bash
TAG=live_streaming

cwd=$(pwd)
cd $(dirname $0)
docker build -t $TAG . && docker run -p 8080:8080 -p
    ↪ 1935:1935 $TAG
cd $cwd
```

Listing 5: Dockerfile

```
FROM tiangolo/nginx-rtmp

RUN apt update
RUN apt install -y ffmpeg

RUN mkdir -p /var/lib/stream/hls/
ADD server/ /etc/nginx/
ADD client/ /usr/share/nginx/html/
```



Distribution ID	E3SQEAWGVF0B3U
ARN	arn:aws:cloudfront::848737590612:distribution/E3SQEAWGVF0B3U
Log Prefix	-
Delivery Method	Web
Cookie Logging	Off
Distribution Status	Deployed
Comment	aasii http stream cdn
Price Class	Use All Edge Locations (Best Performance)
AWS WAF Web ACL	-
State	Enabled
Alternate Domain Names (CNAMEs)	-
SSL Certificate	Default CloudFront Certificate (*.cloudfront.net)
Domain Name	dzpalxnsxqao8.cloudfront.net
Custom SSL Client Support	-
Security Policy	TLSv1
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2019-12-11 17:34 UTC+2
Log Bucket	-

Fig. 6: CDN distribution's settings