HTTP-Based Video Streaming Service

Hyun Bum Cho (4195368)

Abstract

Real time music and video streaming is now more prominent than ever before. In 2015, more than 70% of North America's internet traffic in peak-evening hours came from streaming video and audio sites like Netflix and YouTube [1]. Due to this rise in usage, there is competition among audio and video content providers to provide the most "on-demand" streaming service. Users are now increasingly sensitive to delay and errors during playback, with startup delays beyond 2 seconds causing viewers to abandon the video entirely [2]. In order to provide such a service, we will need to employ technologies and protocols that efficiently throttle the quality and rate of data based on network conditions such as MPEG-DASH. In this project, I will be implementing an on-demand HTTP-Based video streaming web service.

1. Introduction

A video streaming service is, quite simply, an application that allows users to watch videos from their mobile or web browser anytime and anywhere as long as they have a connection to the internet. In order to provide the service to all mobile and web users, we will be basing our video streaming service on the HTTP protocol. This allows us to bypass rulesets that block other alternative protocols that are used for video streaming like UDP. In order to provide certain quality of service requirements, the industry standard is currently set on implementing DASH-MPEG, which is a technique that players implement to provide the best video quality at the bandwidth requirement of the user. This works by adaptively toggling and switching video qualities when bandwidth changes. The player is able to find out about the different video quality endpoints through a manifest file, which lists the endpoints, quality, and bandwidth levels for each video segment.

Our HTTP-based video streaming service consists of the client player and video content distribution server. The client player is a HTML5 based video player that'll request for video chunks from the server. As soon as the player receives the first chunk, it'll play the video until it runs out of data to play. When the server receives a request for a video, it will serve the content based on the range header provided. Currently, the videos are stored in the webm format.

DASH is implemented to provide the best quality for the bandwidth requirements of the client. I will now explain briefly how DASH works with our service. I created a bash script that splits the video into audio and mute video. It then converts the video into 3 different levels of quality. The reason for doing this is to provide a single quality audio, since audio is much lower in size compared to video. This script also generates the manifest and metadata files used to map bandwidth requirements to video urls and timestamps to byte ranges. The script is located at `bin/generate_dash.sh`. This script is based on the ffmpeg for adjusting video quality and mse_json_manifest for creating timestamp:byte mappings.

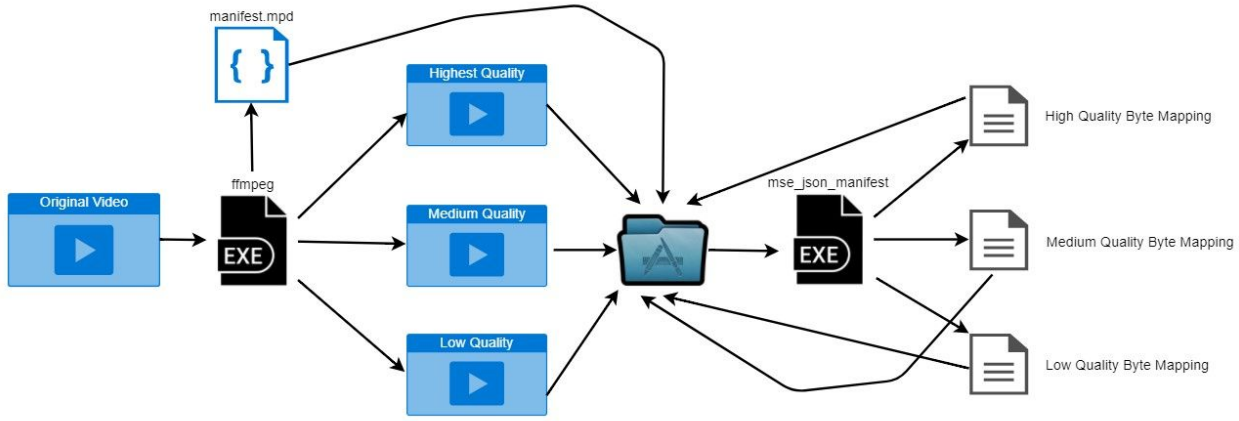

Figure 1: generate_dash.sh flow (created with draw.io)

Once I have generated the videos, the client player will work by requesting for the manifest file and the timestamp:byte mappings for each quality. Once the client receives the manifest and metadata files, it will start playback based on the speed of the network. As the server supports range based video requests, there will be no problem switching between quality.

2. Literature Survey

Despite me working with web-based technologies, the development I did was mostly lower-level. I had to work with byte streams and piping data to a player. Therefore, there is less documentation on the bugs I want to fix and tasks I encounter. I mainly referenced the mozilla developer documentation. Here is a starting point to the documentation: https://developer.mozilla.org/en-US/docs/Web/API/MediaSource I also had to reference documentation on the HTTP specific information, such as Accept-Ranges and range headers. Here is the starting point for the documentation: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Ranges

Of course, I referenced stackoverflow to answer my questions related to bugs and tasks, but there are far too many of them to list. Here is an example of such a post: https://stackoverflow.com/questions/6682835/is-it-possible-for-an-ajax-request-to-be-read-before-the-response-is-complete

I did reference source code as well. The main repository I read source code for was: https://github.com/Dash-Industry-Forum/dash.js. I referenced this when initially thinking about how, architecturally, I should create the DASH-MPEG implementation. However, there is no need to worry about me copying code. In fact, the implementation of dash in this github repository is so sophisticated, that it'd take me more time to copy code from this repository than it'd to finish the project itself!

3. Methodology/Implementation

As noted in introduction, our client attempts playback of the video by requesting for a manifest file and metadata files. Once it has retrieved the appropriate information, we are able to intelligently handle byte range requests and quality adjustment based on network conditions. In a broader sense, the player is requesting for bytes from the video distribution server, and appends the buffers to the HTML5 player.

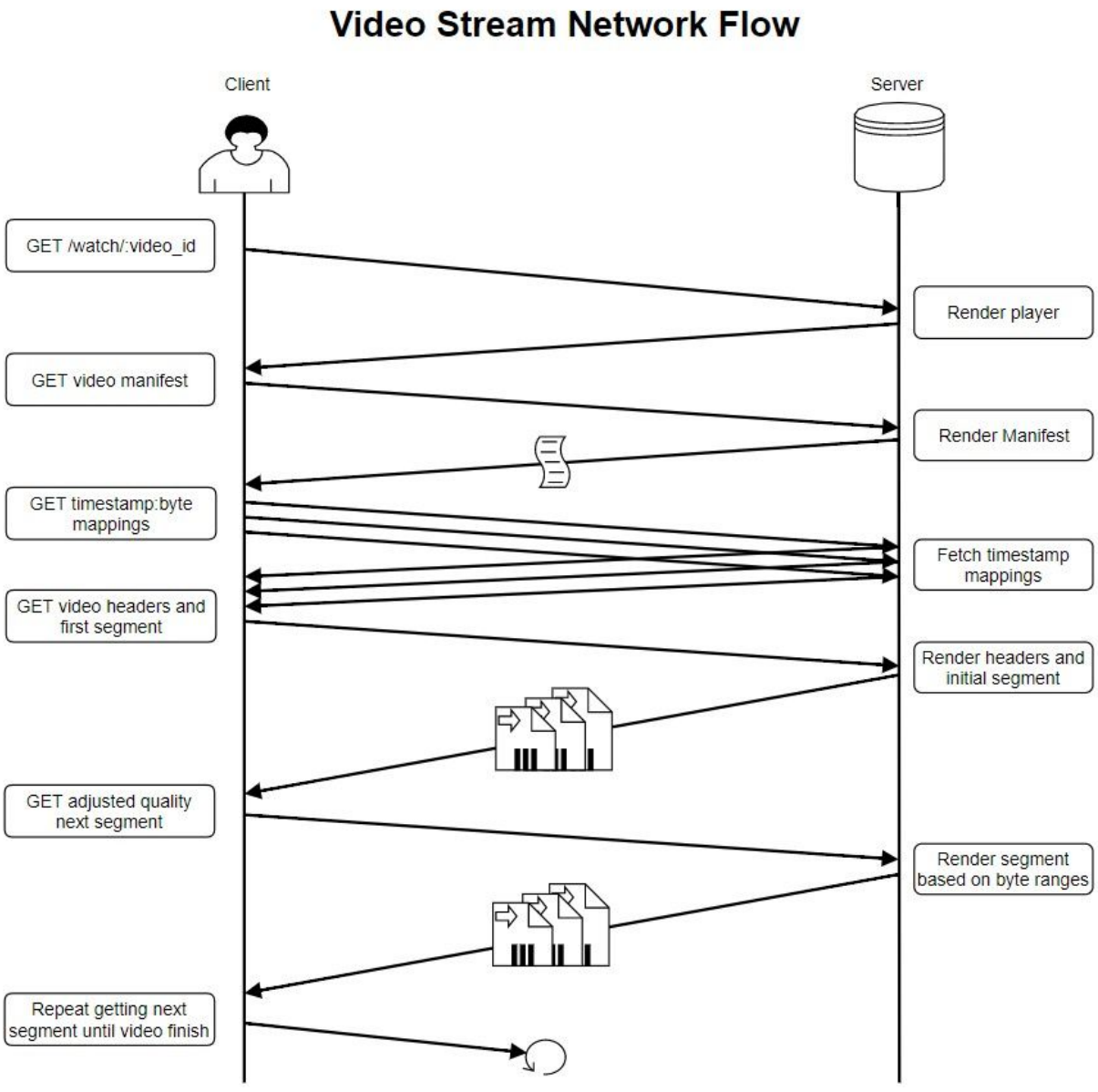

Figure 2: Interaction between the DASH player and server (created with draw.io)

I will now elaborate further on the dependencies for video playback. The client uses the manifest to know which video qualities are available. The manifest will also tell us the urls of the different video qualities. The player uses the manifest to request for what i'll refer to as

“timestamp:byte mappings”, which is also shown in figure 2. “Timestamp:byte mappings” is a json file that corresponds to a particular video quality. This file will tell us which bytes to request for the timestamp or range that we want. A benefit of this mapping file is to intuitively figure out which bytes to request when switching between quality levels. This is also particularly useful for playback, because each timestamp entry corresponds to a byte range that starts off with an I-Frame. If the player were to switch quality levels, it would need to make sure the bytes corresponding to the new quality following the older quality were part of an I-Frame.

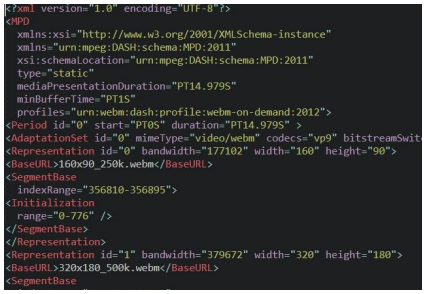

Figure 3: manifest.mpd

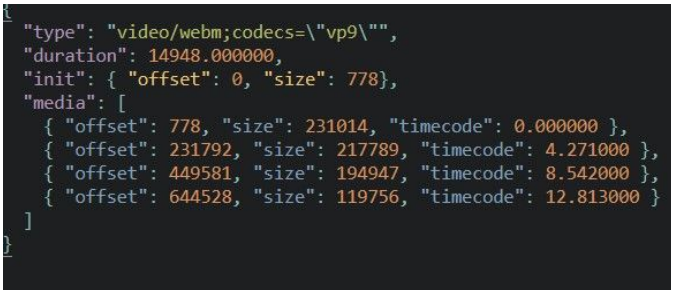

Figure 4: timestamp:byte mapping

I will now explain the different components of the application and how they were built.

3.1 Server

The server is implemented in Node.js. The server currently works as a program that serves the manifest file (for DASH), timestamp mappings, and a video stream. You can access both of these by hitting a specific endpoint.

METHOD	ROUTE
GET	/
GET	/watch/:video_id
GET	/watch/:video_id/manifest.mpd
GET	/watch/:video_id/timestamps/:filename
GET	/watch/:video_id/:filename

Figure 5: Routing table for server

In figure 5, we can see the routes that allow access to specific files. When requesting for `/watch/:video_id`, the client will receive the HTML and JavaScript for the player needed to watch the video. The filename in the endpoints corresponds to the quality, as the videos are named based on the quality of the video. For the video stream endpoint specifically, you can specify a range header in the HTTP request. If you specify this header, the response will send the video in the range specified. An example key value pair for the range header would be: `range: bytes=0-1023`, which would request for the first 1024 bytes in the video stream.

3.2 Client

The client is a JavaScript based webm player. The basis of this is using the MediaSource API to pipe buffers into the HTML5 player. We are able to access the lower level layer of the xhr requests by using the JavaScript fetch API. This allows us to access the parts of the response as they come, rather than after the request has finished. When the response stream starts, which happens when you receive the first part of data from a response, the player will pipe the buffers into the HTML5 player, which subsequently starts the playback of the video.

The data is piped to the HTML5 player via the MediaSource API. The MediaSource object takes in two sources of data: audio and video. The audio and video contents are fetched from the server and are piped to the player, which does the synchronizing of playout.

Quality decrements are performed when the time to fetch the buffer is greater than 75% of the playback time of the information itself. Quality increments are performed when the time to fetch the buffer is less than 50% of the playback time. Because the video is segmented, or more appropriately indexed, via the timestamp:byte mapping files, we are able to gauge the playback time:time to fetch ratio.

There is also a component of the player that implements exponentially back off with retries. This happens if the network goes down altogether, in which the player attempts to request for the correct byte ranges and doubles the time between requests on each failure.

The source code of the client/player is located in the player/ folder. This source code is concatenated together using browserify, which allowed me to modularize my code.

3.3 DASH conversion script

I created a bash script that splits a video into its audio and video component. It then converts the muted video into multiple qualities and generates a manifest file and timestamp:byte mappings. This script is available as `bin/generate_dash.sh`. This works by using the ffmpeg program and specifying different scales and dimensions of the video output for quality generation and using the mse_json_manifest program for creating segments, or indices, within the video denoting I-frames via timestamp:byte mappings.

Note: This script will only work on a windows machine that runs the Ubuntu subsystem. You do not need to run this script to start the service.

```
#! /bin/bash

EXEC_NAME=$( dirname $0 )"/ffmpeg.exe"
FILE_PATH=$1
OUTPUT_PATH_PREFIX=$2

if [ $# -ne 2 ]; then
    echo "Usage: ./generate_dash.sh <input_file_path> <output_folder>"
    exit
fi

mkdir -p ${OUTPUT_PATH_PREFIX}

EXEC_WITH_INPUT="${EXEC_NAME} -i ${FILE_PATH}"

# generate audio
${EXEC_WITH_INPUT} -vn -acodec libvorbis -ab 128k -dash 1 ${OUTPUT_PATH_PREFIX}/audio.webm
```

Figure 6: generate_dash script will only work on Windows Bash Subsystem

3.4 CDN

For the Geo-distributed CDN layer, I decided not to go with the standard deployment to AWS, due to the cost in keeping such a server up and the lack of true "implementation" when using AWS. What I decided to do instead was to deploy my application server to multiple regions via Heroku. Both applications forward traffic that is better suited to the other regions.

```
dannycho7@DESKTOP-TR59MC2:~$ curl --header "x-forwarded-for: 128.111.0.1" "https://eu-http-video-streaming.herokuapp.com"
Found. Redirecting to https://http-video-streaming.herokuapp.com
```

Figure 7: Getting redirected to the NA site when requesting the EU site

3.5 Infrastructure Setup

In order to setup the repository, take a look at the README.md supplied in the source code. It is, however, very simple at the moment to set up the project. I can imagine that by the time I submit my final project, there may be a few more steps for setup. You can also access a version of the program deployed on heroku at https://http-video-streaming.herokuapp.com/.

3.6 Libraries/Tools Used

Though I had mentioned the libraries I used through the previous steps, I will list them here to provide a more cohesive and comprehensive list of libraries I used. Here is a list of the libraries I used to develop the HTTP-Based video streaming service:

- Node.js (Used to create the server)
    - Express.js (A web framework that saves me from writing a lot of boilerplate code)
    - Browserify.js (Used to modularize player)
- JavaScript (Powers both Node.js and is used heavily to build the client player)
    - MediaSource API (A way to alter low level HTML5 player)
    - Fetch API (Used to access low level XHR information to stream response)
- FFmpeg (Used for DASH - A program used to convert videos into different qualities)
- Mse_json_manifest (Used to generate timestamp:byte mappings for videos)

4. Results

We can simulate bandwidth changes by using the chrome's bandwidth throttling option. I used this to debug DASH, as I can notice quality decreases when switching to slow 3g.

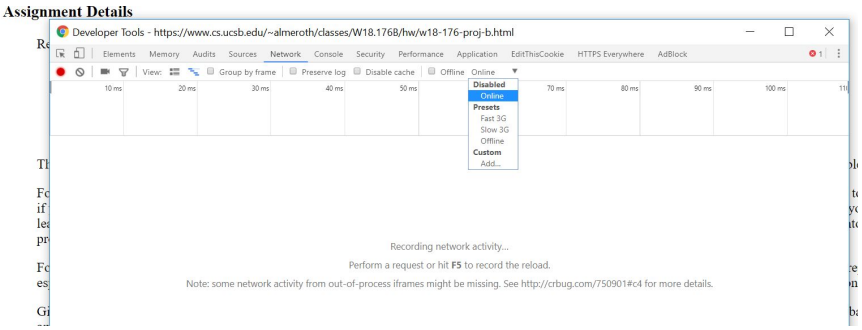

Figure 8: Chrome Developer Tools Network Throttling

For our experiments, we will be observing two metrics: video start time and lag ratio. Video start time refers to the time it takes from the start of when the client requests for the content to the time of initial playback. As noted in [2], a long startup delay (such as 2 seconds) could result in the user abandoning the video altogether. Therefore, video start time is an important metric to gauge. Lag ratio refers to the amount of time spent waiting for video buffers vs time spent watching the video. This is an important metric to minimize, as it directly correlates to user enjoyment. We will be running tests to measure both video start time and lag ratio by using the chrome network throttling tool. We will run our tests on network download speeds of: 10 Mbits/s, 1 Mbits/s, 100 kbits/s, 10 kbits/s, and 1 kbits/sec. We have set the lower bound of bandwidth to be 100 kbits/s, because the lowest available quality has a bit rate of 250k, meaning it has a bitrate of 250 kbits/s. In any case, we are not expecting a good playback experience on our lower bound bandwidth.

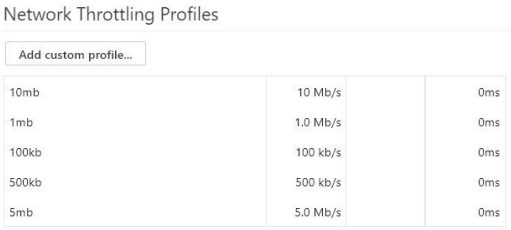

Figure 9: Custom Network Throttling Profiles

We are able to measure the lag ratio by measuring how long it takes to play out the video. We use this value and subtract the video duration from it and divide that by the video duration to retrieve the lag ratio.

```
get lagRatio() {
    return (this.timeElapsed - this.videoDuration) / this.videoDuration;
}
```

Figure 10: Calculating lag ratio

We calculate the video start time by observing the network utility in the chrome developer tools. By taking a look at how long it took to finish the first segment of the video, we know the video start time, since our player starts as soon as it receives bytes.

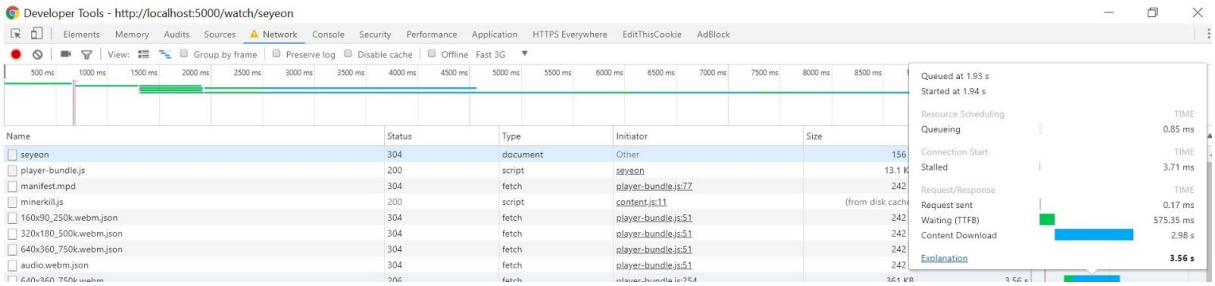

Figure 11: Measuring video start time

I ran the varied network bandwidth tests on a 15 second video of 250k, 500k, and 750k bitrate encoding. I generated the graphs using the matplotlib library available in python. Below are the results from the tests:

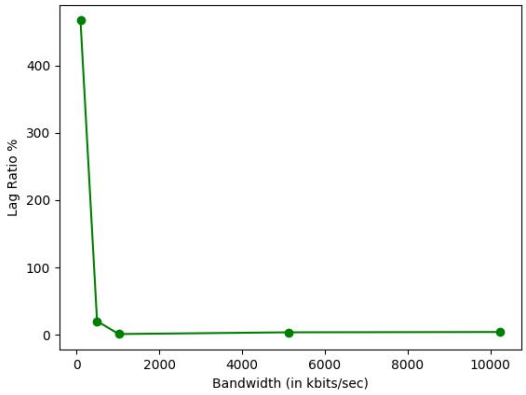

Figure 12: bandwidth vs lag ratio (in %)

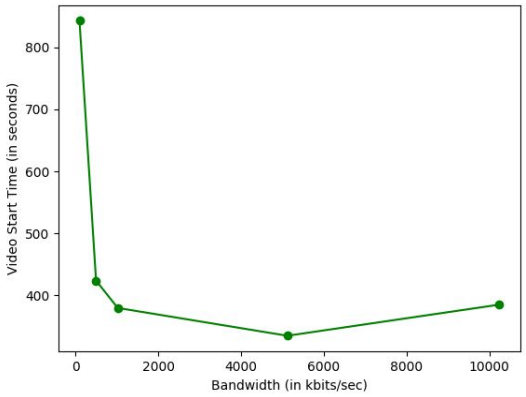

Figure 13: bandwidth vs video start time

As noted from the results, there is not much of a different in lag ratio or video start time between the higher end bandwidths (from 500 kbits/sec - 10 Mbits/sec). Though it isn't obvious through the data, this is a result of the adaptive streaming. Because the lower end bandwidth connections requested for lower quality, they were able to stream the video without any pauses, which resulted in no difference in playback speed or lag between the higher end bandwidth connection.

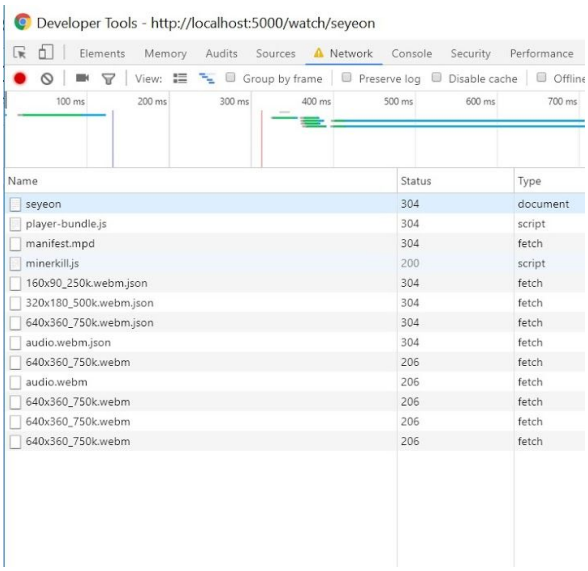

Figure 14: 10 mbit/s connection

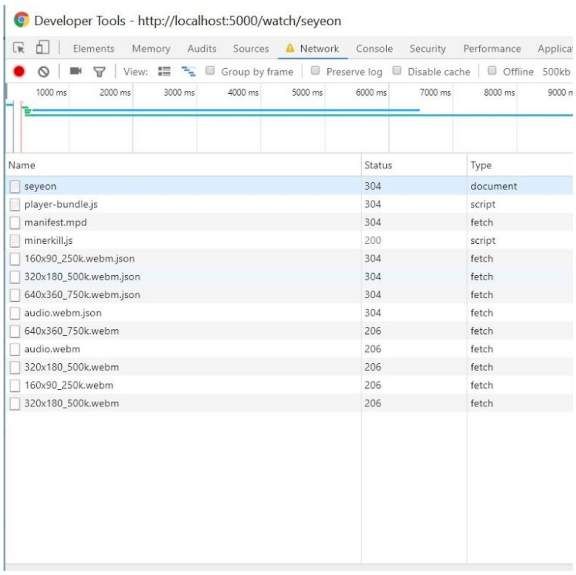

Figure 15: 500 kbit/sec connection

In the figures above, you can see the different qualities of video the two connections request. In the 500 kbit/sec connection, the quality throttles between the 250k and 500k bit rate,

while in the 10 mbit/sec connection, it sticks with a constant 750k bit rate. The conclusion to be made here is that our DASH-MPEG implementation has significantly improved the viewing experience for a certain range of bandwidth connections.

5. Comments

One of the challenges I faced was the restrictions the browser APIs place on the technologies I was using. For example, I wasn't able to request data and receive a streamed output of the data. The standard usage of the XMLHttpRequest API would return me a huge buffer that was the result of the entire response of a video request. This would prevent me from starting the video until after the video finished downloading. My workaround to this was using the experimental fetch API which allowed me to receive a stream that pipes buffers.

There were also a lot of concurrency bugs that I encountered. This wasn't surprising to me as I had worked with web based technologies and am used to the interesting bugs you can encounter when developing with JavaScript. It was also challenging to make sure that the correct bytes were being appended to the MediaSource objects, especially in consideration to quality changes and network failures.

References

[1] "More than 70% of internet traffic during peak hours now comes from video and music streaming",
http://www.businessinsider.com/sandvine-bandwidth-data-shows-70-of-internet-traffic-is-video-and-music-streaming-2015-12

[2] S. S. Krishnan and R. K. Sitaraman, *Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs*, in *IEEE/ACM Transactions on Networking*, vol. 21, no. 6, pp. 2001-2014, Dec. 2013.