MPEG-DASH

Implementation in Web Applications

Sean MacDougall
Mathematics & Computing Science
Saint Mary's University
Halifax, Nova Scotia, Canada
Sean.MacDougall@smu.ca

ABSTRACT

Project source code can be located at:
https://github.com/seanmacd/mpeg-dash-project

A web application was created that implements an end-to-end MPEG-DASH ABR streaming that handles encoding and delivering user uploaded video files. A back-end was implemented using Typescript and Express with FFMPEG command line arguments to encode user given video files into various bitrates. A front-end was implemented using Typescript, React, Mantine, and Dash.js to create a ABR streaming video player and various functions relating to the video player. The web application is hosted on a Ubuntu server using Docker compose. A Cloudflare container is used to access the web application over a zero trust network connection.

Various videos were uploaded to the web application to test the consistency of encoding and delivering video streams. Tests included videos of different resolutions, aspect ratios, and lengths to ensure proper encoding and playback. Network analysis tools were used to verify the video player requests correct encoded segments, and adapts the quality based on throughput, latency, and buffer level.

Based on the tests the web application successfully handles user uploads, video file encoding, and ABR streaming. The software consistently generated correct DASH manifest files, chunks at all defined bitrates, and delivered smooth play back that adjusted quality based on network performance. These results demonstrate that the implementation meets the requirements of a functional end-to-end MPEG-DASH ABR streaming workflow, providing a strong foundation for future improvements and exploration.

KEYWORDS

Dynamic Adaptive Streaming over HTTP (MPEG-DASH), Adaptive Bitrate (ABR), FFMPEG, Web Application, Encoding, Video Player, Playback

1. BACKGROUND

Dynamic Adaptive Streaming over HTTP, otherwise known as MPEG-DASH is a video streaming and encoding protocol to deliver video that adapts to a user's network conditions.

The MPEG-DASH protocol involves encoding a video format file by using FFMPEG or a wrapper class of FFMPEG to convert the video format into multiple encoded chunks at various bitrates, typically three to eight seconds in length. FFMPEG is an open source multimedia toolkit, in the case of this project FFMPEG command line arguments are used to encode video files. The small segments get encoded into M4S file format, a small, individual media segment used in streaming video over the internet, often part of a larger collection of segments. The collection of encoded M4S files are stored alongside a MPD file, the manifest file. The manifest file contains information about the available streams, which correspond to the chunks encoded at different resolutions and bitrates.

A DASH player is a video player that has the capabilities of reading and using the manifest file paired with the encoded chunks. For the scope of this project, the Dash.js library is used for the DASH video player. Dash.js is an open source Javascript based multimedia player library that implements the MPEG-DASH protocol for video playback in a web browser environment.

2. IMPLEMENTATION

2.1 FFMPEG ARGUMENTS

FFMPEG uses a large collection of arguments to perform a task such as converting or encoding a video file. In this project the FFMPEG arguments are compiled into a single file (*ffmepg-args.ts*) to be called upon by the encoding function.

Within the FFMPEG arguments file we define the resolutions and bitrates to be used for the encoding. Nine resolutions and bitrates were chosen, ranging from 180p (320x180 20k) to 1080p (1920x1080 5300k).

To begin the functionality of the FFMPEG arguments, a check is performed for any audio files present in the video. This is done by taking in the video file path as input, and using execSync to run ffprob. Ffprob is a command line tool within FFMPEG to analyze and extract detailed information on a video file. Specifically this is done to check for the presence of audio in an uploaded video file.

Next is the FFMPEG arguments function to define the encoding for a given input of a video file path and a manifest file path. First the function creates a filter for the given video with each resolution. This filter resizes the video to fit the resolution, forces the aspect ratio ensuring there is no distortion, and creates padding on the video if the original does not fit the newly defined resolution (See Figure 1). The remaining list of arguments in the function do the following: create a specific output stream for each resolution, set the file format to DASH, split the video into 4 second chunks, and write the manifest file.

```
RENDITIONS.forEach(({w, h, br}, i) => {
  filterComplexes.push(
    `[0:v]scale=w=${w}:h=${h}:force_original_aspect_ratio=decrease,pad=${w}:${h}:(ow-iw)/2:(oh-ih)/2[v${i}]`
  )
  mapArgs.push('-map', `[v${i}]`, '-b:v:' + i, br, '-c:v:' + i, 'libx264')
})
```

Figure 1: **The set of arguments to perform resizing, forcing aspect ratio, and adding padding**

In summary, a video file path and manifest file path are passed into the function, any audio from the video is extracted, the video is cloned 9 times and resized into the respective resolutions, encoded, split into 4 second chunks, and a manifest file is written. This function and its set of arguments is used in the command line invoking within the encodeFile function.

2.2 UPLOADING AND ENCODING

The encoding of a video file is handled through the encodeFile function (*encoder.ts*), that invokes the FFMPEG arguments discussed in the previous section.

FFMPEG is an external process, running outside the program on the operating system. The encodeFile function is wrapped in a promise allowing our call to the function to use async/await, allowing the program to wait until the video file has been processed. The encodeFile function is called from the *router.ts* file.

A child process is spawned, calling ffmpegArgs with the video file path and manifest file path (see Figure 2). A spawn was chosen instead of exec for the output. In a spawn the output, which in the case of this project is just logging information is directly streamed, providing real-time feedback. An exec buffers output, providing feedback only after the process is complete. A logger is created to pipeline the FFMPEG output to the console.

```
const proc = spawn('ffmpeg', ffmpegArgs(videoPath, manifestPath))
```

Figure 2: **Child process spawning**

The encodeFile function is an API request, but the encoding can take a variable amount of time based on the size of the file and how many resolutions are being encoded. To avoid keeping the HTTP request open for an extended period of time, we use an EventEmitter. This allows our app to listen for when this process is complete or failed (see Figure 3).

```
encoderEvents.on('complete', jobId => {
  jobs.set(jobId, JobStatus.Complete)
  setTimeout(() => jobs.delete(jobId), msHour)
})
encoderEvents.on('failed', jobId => {
  jobs.set(jobId, JobStatus.Failed)
  setTimeout(() => jobs.delete(jobId), msHour)
})
```

Figure 3: **Complete or failed of the process**

When the encodeFile function is complete it exits with a code, either zero or non-zero. A zero code indicates a successful execution. The success is logged, broadcasts the ‘complete’ jobId, and fulfills the promise. A non-zero code indicates there was a problem in the execution. The error code is logged, broadcasts ‘failed’ jobId, and throws an error in the promise.
In the front-end, the encoding can be performed by clicking the ‘Create new stream’ button (see Figure 4). This brings up a modal with two required fields: Stream name, and

Video file (see Figure 5). The Stream name field is where the user creates the name of their video stream. The Video file field is where the user uploads a video file to be encoded. After these fields are filled in, the user can click ‘Create stream’, which then makes an API request for encoding to be performed.

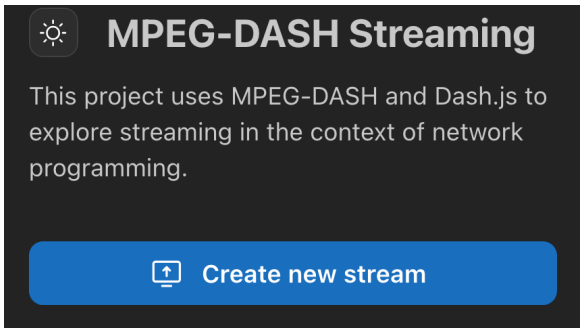

Figure 4: **Create new stream button**

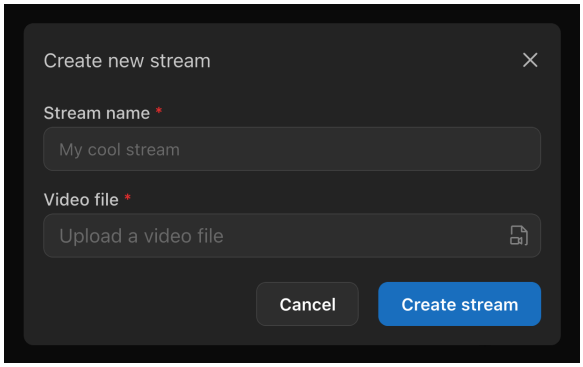

Figure 5: **Modal to create new stream**

Once the stream has been created, a notification is created in the bottom right of the web application (see Figure 6). This notification will remain on screen until the process is complete and the jobId updated. Once the process is done, the notification will update to indicate to the user that their stream has successfully been encoded and is ready to be streamed, or encountered an error (see Figure 7).

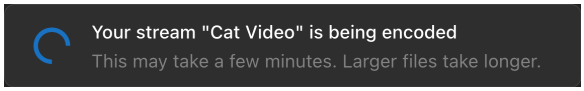

Figure 6: **Stream is being encoded notification**

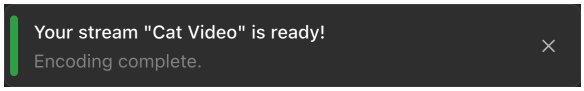

Figure 7: **Stream is ready notification**

2.3 DASH ABR VIDEO PLAYER

A DASH video player is able to read in encoded M4S chunks, and a MPD manifest file and use adaptive bitrate (ABR) algorithms. A DASH video player and functionality can be created from scratch, but for the scope of this project, the Dash.js library is used.

To create the DASH video player, a custom hook was created *useDashPlayer*, to encapsulate the initialization, control, and monitoring. The hook is used in the *App.tsx* file, where it returns four things: *playerRef, qualities, currentQuality, changeQuality* (see Figure 8). A *playerRef* is a React reference, which holds reference to the initialized player, which exposes its internal APIs. *qualities* is a list of the available qualities (the nine resolutions previously defined). *currentQuality* is a reactive variable that changes based on the resolution currently being used. *changeQuality* is a utility that acts on the video player, allowing the change of resolution being played.

```
1  const {playerRef, qualities, currentQuality, change
   Quality} = useDashPlayer(videoRef, {stream})
```

Figure 8: ***useDashPlayer* hook call**

In the front-end a video stream and quality can be selected from the drop downs. The ‘Stream’ drop down displays all video streams that have been previously encoded (see Figure 9). The ‘Quality’ dropdown displays all of the available qualities, including their resolution and bitrate (see Figure 10).

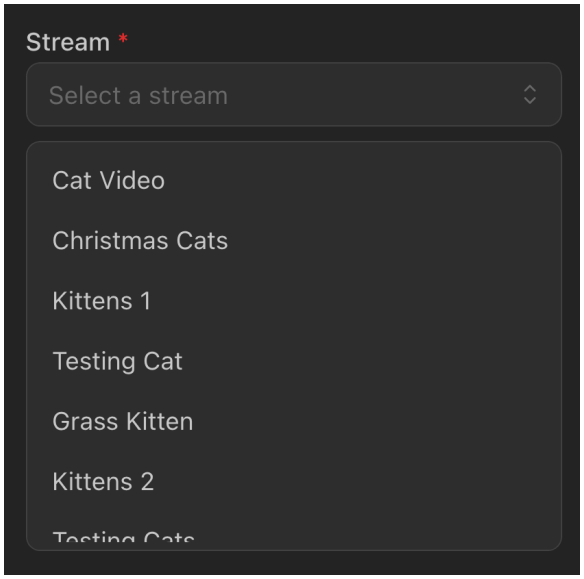

Figure 9: **Stream selection dropdown**

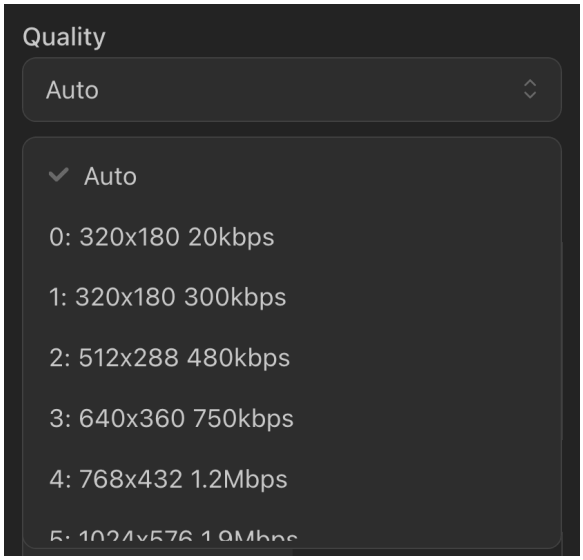

Figure 10: **Quality selection dropdown**

Once a stream is selected, the video player will change to the stream. The current quality will update with a table including the quality ID, resolution, and bitrate (see figure 11). The metrics are from the *useDashPlayer* hook.

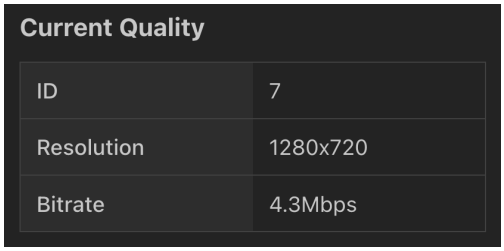

Figure 11: **Current Quality chart**

2.4 METRICS

The metrics chart is displayed below the current quality chart (see Figure 12). The metrics include the following: average throughput, average latency, dropped frames, and buffer level. To create the metrics, a custom hook was created *useDashMetrics*.

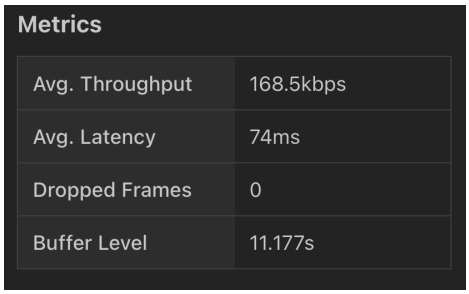

Figure 12: **Metrics chart**

The average throughput is calculated based on how fast the last chunks were downloaded. The average latency is calculated based on the average round trip time to request and receive the chunks. The dropped frames display how many frames were dropped so far during the playback of a stream. The buffer level displays how many seconds of the stream is currently buffered.

All of these metric representations are handled under the Dash.js library. The custom hook bundles these together, and provides a clean output for the front-end to use for displaying to the user (see Figure 13).

```
1  const throughput = player.getAverageThroughput('video')
2        const latency = player.getAverageLatency('video')
3        const droppedFrames = player.getVideoElement().getVideo
   PlaybackQuality().droppedVideoFrames
4        const bufferLevel = player.getBufferLength('video')
```

Figure 13: **Functions provided by Dash.js used in the custom hook *useDashMetrics***

2.5 DEPLOYMENT

A GitHub Actions (GHA) workflow was created *build.yaml* located in the *.github/workflows* directory. This workflow builds the front-end and API components, and creates a Docker image. Both components are copied into a single image: the front-end bundle is served on the back-end's web server. While separate containers for separate components is the standard for large production systems, a single-container approach is suitable for a small project with a simple front-end, making deployment simpler.

The project is deployed on an Ubuntu VM (Oracle Cloud) using Docker Compose. The compose file defines two services: project and cf (see Figure 14). The project service is the container that we built in the GHA workflow. It has a mounted volume: the streams directory. This corresponds to a directory on the host server, where the encoded streams are stored. By mounting a persistent volume, the uploaded streams survive container restarts. The cloudflared container is used as a connector to Cloudflare's zero trust system, allowing users to access the web application over a zero trust network connection (ZTNC).

```
1  services:
2    project:
3      image: ghcr.io/seanmacd/mpeg-dash-project:latest
4      volumes:
5        - ./streams:/app/streams
6    cf:
7      image: cloudflare/cloudflared:latest
8      environment:
9        TOKEN_KEY: redacted
10     command: tunnel run
```

Figure 14: **Docker Compose file**

3. RESULTS AND DISCUSSION

The web application and software follow MPEG-DASH as a media encoding protocol, and implements a DASH player to view such encoded media, and uses adaptive bitrate algorithms to adjust the quality. The web application successfully allows a user to upload a video file, give it a name, and properly encode it into the nine defined resolutions. Furthermore, the user can select a stream from their uploaded videos, and select a quality to view the video in. If auto is selected then the playback with adapt the quality level of the playback according to the user's network conditions.

Additionally, the software is designed for error prone cases such as the following: incorrect file type, incompatible sizing, multiple encodings at once, and error in encoding. The software allows only uploads of video media, and properly doesn't allow the upload of any other file type. If an uploaded video is not of the correct aspect ratio the software adjusts the video to fit, avoids any distortion, and adds top/bottom or side padding to fit the defined aspect ratio (see Figure 15). The software handles multiple encodings at once by performing jobs in parallel. If there is an error in the encoding of a video, and the video is not successful, then the software will delete the stream instead of uploading the broken stream.

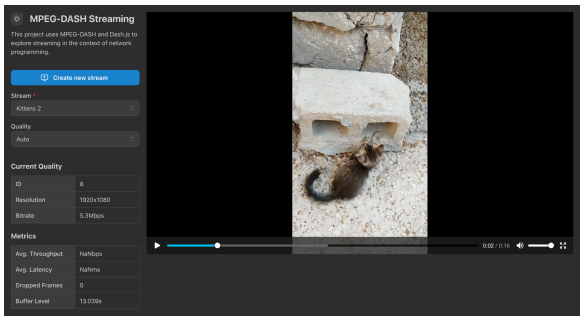

Figure 15: **Incompatible size adjusted to fit**

Upon playing a steam, any quality level can be chosen. The video will immediately switch to that new quality by using the initial chunk of the quality chosen. The auto feature will enact the adaptive bitrate algorithms to determine which quality to initialize, and which quality to continue to stream. When a new quality has been deemed needed, the video will immediately switch again using the initialization chunk (see Figures 16 and 17).

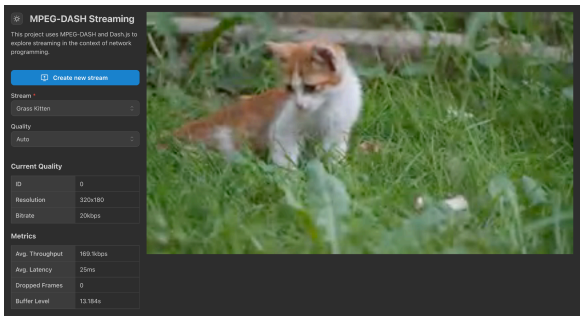

Figure 16: **Quality level 0 (320x180 20kbps)**

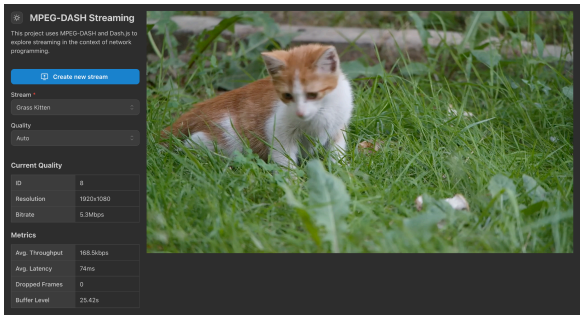

Figure 17: **Quality level 8 (1920x1080 5.3Mbps)**

We can analyze the network requests to further look at how the video playback is being adjusted. The chunk file name is broken down into *chunk_<quality ID><chunk sequence number>.m4s. The initialization chunk file name is broken down into init_<quality ID>.mp4*. Chunks to be played are grabbed from the stream directory on the server from a HTTP GET request.

Looking into the chunks and not viewing the metrics and current quality, we can see the switch of qualities. The software successfully lowers the quality when necessary (see Figure 18), and increases the quality when necessary (see Figure 19). The switch in quality is determined from the adaptive bitrate algorithms.

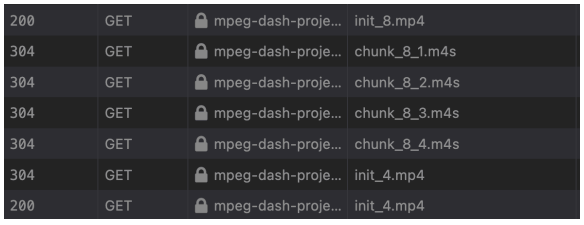

Figure 18: **A decrease in quality being played based on adaptive bitrate algorithms**

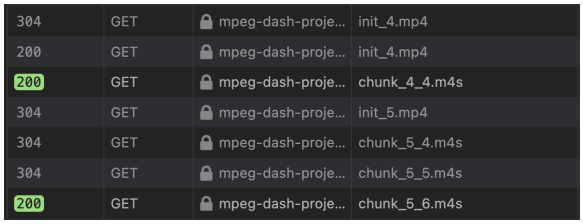

Figure 19: **An increase in quality being played based on adaptive bitrate algorithms**

When a new stream has been created, a HTTP POST request is sent, which invokes the spawning of a child process with FFMPEG, and creates a job to track to completion of the encoding (see Figure 20). During the encoding process, a GET request is repetitively sent to check the status of the job. Once the job is complete a final GET request is sent, and the notification can update to indicate the stream is ready (see Figure 21).

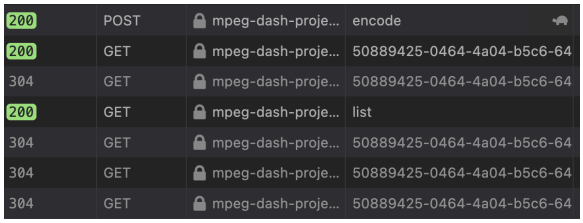

Figure 20: **Encoding POST request**

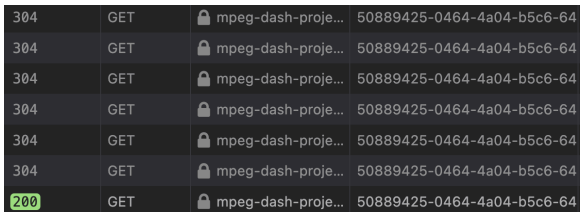

Figure 21: **Encoding GET request to indicate job is complete**

4. CONCLUSION

The goal of this project was to design and implement an end-to-end MPEG DASH workflow capable of accepting user given video files, encoding them into multiple resolutions, and delivering playback through an adaptive bitrate video player. The results demonstrate that the software successfully meets these objectives. The web application reliably handles video file uploads, processes videos through FFMPEG into nine defined resolutions and bitrates, generates valid DASH manifests, and allows the adaptive playback of such videos.

The implementation of Dash.js and the custom hooks provided a clean front-end for monitoring playback quality, throughput, latency, dropped frames, and buffer levels. The metrics behaved consistently with that expected of an adaptive bitrate software, offering information to the user on their network conditions. A network analysis confirmed that the player requests the correct chunks, switches qualities when needed, and maintains clean playback through variable network conditions.

Overall the web application provides a complete and reliable demonstration of MPEG-DASH encoding and adaptive streaming within a modern web browser environment. It forms a strong foundation for future exploration and development including: exploring deeper into ABR algorithms, subtitle handling for videos, selecting which qualities you want to encode in, improving overall encoding performance, and more. The successful performance across all tests confirms that this project meets all requirements for end-to-end MPEG-DASH ABR streaming that handles encoding and delivering user uploaded video files.

REFERENCES

[1] Cloudflare. *Cloudflare Docs.* https://developers.cloudflare.com/

[2] Cloudflare. *What is MPEG-DASH> | HLS vs. DASH.* https://www.cloudflare.com/en-ca/learning/video/what-is-mpeg-dash/

[3] Dash.js. *Dash.js Documentation.* https://dashif.org/dash.js/

[4] Docker. *Docker Documentation.* https://docs.docker.com/

[5] FFMPEG. *FFMPEG Documentation.* https://www.ffmpeg.org/documentation.html

[6] Github. *Github Actions Documentation.* https://docs.github.com/en/actions

[7] Mantine. *Mantine Documentation.* https://mantine.dev/

[8] Pexels. *Free Kitten Videos.* https://www.pexels.com/search/videos/kitten/

[9] React. *React Reference.* https://react.dev/reference/react

[10] TypeScript. *TypeScript Documentation.* *https://www.typescriptlang.org/docs/*