Low-Latency Live Streaming Platform Using DASH

Turgay Bulut[1] and Azra Oymaağaç[1]

[1]Engineering Faculty, Ozyegin University

Abstract

This project aims to develop a low-latency live streaming platform, leveraging cutting-edge technologies like FFmpeg, OBS, and DASH. The main objective is to achieve a latency of less than 5 seconds from capture to playback, enhancing real-time interaction for users. The platform integrates OBS as the media source, which encodes real-time video and streams it using the RTMP protocol to FFmpeg. FFmpeg then packages the content into a low-latency DASH format, optimized for quick delivery over the HTTP protocol. The resultant stream is tested and analyzed across various settings to ensure optimal performance and minimal latency. Innovatively, the project also explores the use of QR codes embedded in video frames to measure latency more accurately. The implementation details are guided by modifications in FFmpeg's source code and extensive configuration of streaming parameters.

1. Introduction

Live streaming technology has become a fundamental component of modern communication, widely used in various domains such as entertainment, education, and surveillance. However, one of the significant challenges in live streaming is minimizing latency to ensure that the end-user experiences near-real-time interaction. This project addresses the challenge of reducing streaming latency by employing a combination of Open Broadcaster Software (OBS) for media capture and encoding, and FFmpeg for media packaging and distribution using Dynamic Adaptive Streaming over HTTP (DASH).

The system architecture is designed to capture encoded media through OBS, which is then processed by FFmpeg to package the content into a DASH-compatible format. Key interventions include modifying FFmpeg's source code to enhance its efficiency for low-latency applications and optimizing streaming parameters such as keyframe interval and segment duration. Additionally, an important approach involves embedding QR codes in video frames during encoding. These QR codes represent the system clock time, which can be decoded on the player side to provide a precise measure of the latency experienced by the end-users.

By conducting testing and analysis with different parameters, this project explores the relationship between various encoding and packaging settings and their impact on stream latency.

2. Tools and Techniques Used

In this project, we employed several important tools and techniques to develop a low-latency live streaming platform, ensuring a latency of less than 5 seconds from capture to playback.

Tools

FFmpeg was a cornerstone of our implementation. It was used for ingesting and processing the live RTMP stream, encoding the video, and packaging it into the low-latency DASH format. To meet the project's requirements, we initially compiled the FFmpeg source code and modified it to remove the .tmp extension from output files, ensuring compatibility with the node-gpac-dash server.

OBS Studio served as our media source, capturing and encoding real-time video streams. We configured OBS to stream video using the RTMP protocol to FFmpeg, embedding the time in the video frames for accurate latency measurement. The setup included generating key frames at specified intervals (keyint values) to maintain consistent segment durations.

We utilized a modified version of **node-gpac-dash** to handle chunked transfer encoding, a critical component for achieving low-latency streaming. Adjustments were made to support chunked media segments, and the server was run with options like -chunk-media-segments, -cors, and -chunks-per-segment to manage chunked transfers effectively.

On the client side, **dash.js** was used to play back the low-latency DASH stream. The player was set up in low-latency mode, ensuring minimal delay from capture to playback. Additionally, it synchronized time using the specified UTC timing server, which is crucial for maintaining low latency. Moreover, we also implement our own player and run with it as well.

Developer Tools played a vital role in our network traffic analysis, helping us verify the proper functioning of chunked transfer encoding. This tool ensured that the streaming segments were arriving correctly and allowed us to troubleshoot any issues that arose during implementation.

Ubuntu 22.04.4 LTS served as the operating system for our development environment. Its robust support for open-source tools and ease of configuration made it an ideal choice for compiling FFmpeg from source, running OBS, and setting up our streaming server.

Techniques

To achieve low-latency streaming, we implemented **Low-Latency DASH (LL-DASH)**. By configuring FFmpeg with options like -ldash 1 and -streaming 1, and setting appropriate segment and fragment durations, we successfully reduced the latency to under 5 seconds. Various parameters, such as seg-duration and frag-duration, were meticulously adjusted to find the optimal configuration.

The Real-Time Messaging Protocol (RTMP) was used by OBS to stream live video to FFmpeg. This protocol enabled real-time transmission of media streams, which FFmpeg then processed and packaged for DASH delivery.

To measure latency accurately, we embedded timestamps into video frames using OBS. This embedding allowed us to calculate the time difference between the capture and playback stages precisely.

We undertook extensive **parameter tuning**, testing different keyint values and corresponding segment durations to observe their impact on latency. We also varied the frag-duration values to study their effect on streaming performance. Through this process, we measured latency under different configurations and plotted graphs to visualize the relationship between segment duration, fragment duration, and latency.

For a more advanced latency calculation, we explored the use of **QR codes embedded** in video frames to indicate the system clock time as shown in Figure 1. Implementation is based on the jsQR open source library for the JavaScript. This technique provided a more accurate method for measuring end-to-end latency. By decoding the QR code on the client side and comparing it with the current system time, we achieved more precise latency measurements.

3. Performance Assessment

Before diving into the performance analysis of the system, we will define some of the default parameters used in the system. The FFmpeg is configured to expect an RTMP (flv) stream and operate as an RTMP server. The input source can be re-transcoded to two different resolutions which are $320x180$ and $384x216$. This enables FFmpeg to adaptively switch resolutions

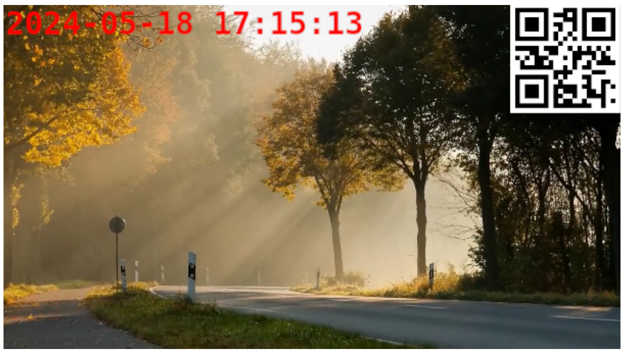

Figure 1: *QR Embedded on video frame*

by considering the connection quality to provide a lower latency stream. For the audio, AAC is used with $128k$ bit rate. Also, FFmpeg is configured for low-latency DASH output by enabling DASH low-latency mode and chunked streaming mode. One of the most important setting is about using the segment template to publish segments instead of the segment timeline because the segment timeline is not supported by LL-DASH. The default segmentation duration is set as 4 seconds. Since we configured GOP size to be 4 seconds in OBS, FFmpeg will package 1 GOP per segment. The chunk duration is used as the default value of FFmpeg which is 1 second duration. To avoid exhausting disk space, the DASH time-shift buffer depth (window size) is set to the value of 5 which means that any segments that fall out the window will be deleted automatically. The Akamai's timing server is used for the time source.

The only setting set on the OBS is keyframe interval which is set to 120. It points out that OBS will generate a 1 keyframe for every 120 encoded frame. The last configurations are on the modified gpac-dash server. It is configured to perform chunked transfer and 4 chunks per segment by default. This chunk for segment value is highly dependent on the keyframes.

To achieve a better performance in terms of low latency, many trials were conducted, and the effect of different parameters was examined.

keyint	seg_duration	latency (secs)
30	1	3.34
60	2	3.47
90	3	3.55
120	4	3.70
150	5	3.89
180	6	4.12

Table 1: *Latency Observed with Different Segment Durations*

Table 1 and **Figure 2** illustrate the relationship between segment duration and latency. The table lists various keyframe intervals (keyint), their correspond-

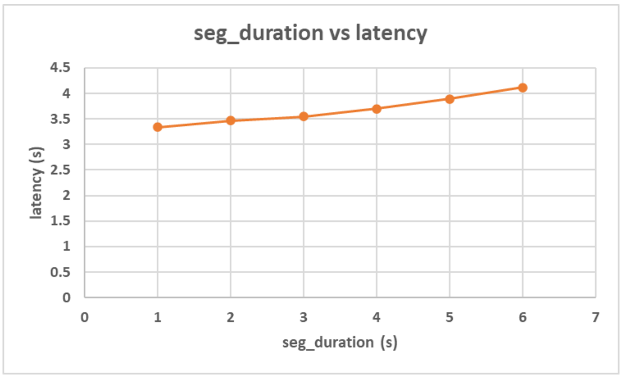

Figure 2: *Segment Duration vs Latency*

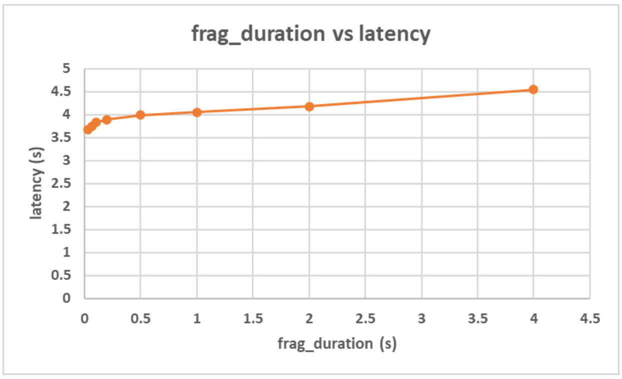

Figure 3: *Fragment Duration vs Latency*

ing segment durations, and the resulting latencies measured in seconds. As observed, there is a consistent increase in latency with increasing segment duration. For instance, at a segment duration of 1 second, the latency is 3.68 seconds, while at a segment duration of 6 seconds, the latency rises to 4.12 seconds. This trend is clearly depicted in the plot, where the latency increases almost linearly as the segment duration extends. This indicates that longer segment durations result in higher latencies, suggesting a trade-off between segment duration and latency performance. Understanding this relationship is crucial for optimizing video streaming and other time-sensitive applications where latency is a critical factor.

frag_duration	latency (secs)
0.033	3.68
0.066	3.75
0.100	3.83
0.200	3.89
0.500	3.99
1.000	4.05
2.000	4.18
4.000	4.54

Table 2: *Latency Observed with Different Fragment Durations*

Table 2 and **Figure 3** presented to examine the impact of fragment duration on latency when the keyframe interval (keyint) is set to 120 and the segment duration is 4 seconds. As shown, the latency varies with different fragment durations. The shorter fragments are transmitted more frequently which allows quicker updates and reduces the time for new data to reach the video player. In contrast, longer fragments increase the time of the player to receive the next piece of stream and this situation increases the overall latency. Even though shorter fragments are beneficial to reduce latency, their frequent requests can cause network overhead. In most cases, this overhead is ignored when considering the benefits of the reduced latency. These factors highlight the importance of selecting an appropriate fragment duration to minimize latency effectively, taking into account the trade-offs between processing overhead and latency.

4. Conclusion

In this project, we developed a low-latency live streaming platform utilizing FFmpeg, OBS, and DASH technologies to achieve a low-latency from capture to playback. Through extensive testing and parameter optimization, we examined the impact of keyframe intervals, segment durations, and fragment durations on latency. Our findings indicate that both segment duration and fragment duration significantly influence latency.

The implementation of QR codes embedded in video frames provided an efficient method for measuring latency, enabling precise adjustments to streaming parameters. Our work highlights the importance of carefully selecting and tuning encoding and packaging settings to optimize latency performance in live-streaming applications.

Overall, the successful development and testing of this low-latency live-streaming platform underscore the potential for significant improvements in real-time media delivery. By leveraging advanced encoding techniques and meticulous parameter tuning, we have demonstrated that it is possible to achieve remarkably low latencies, thus enhancing the user experience in various real-time applications.

References

[1] B. Zhang, "Low Latency DASH Streaming Using Open Source Tools," Medium, 2020. `https://bozhang-26963.medium.com/low-latency-dash-streaming-using-open-source-tools-f93142ece69d`.

[2] FFmpeg, "FFmpeg Formats Documentation". https://ffmpeg.org/ffmpeg-formats.html.

[3] DASH Industry Forum, "Low-Latency DASH". https://dashif.org/news/low-latency-dash/.

[4] DASH Industry Forum, "5th Edition". https://dashif.org/news/5th-edition/.

[5] DASH Industry Forum, "Low-Latency Streaming Issues" GitHub, 2023. https://github.com/Dash-Industry-Forum/dash.js/issues/3994.

[6] FFmpeg, "FFmpeg Streaming Guide - Latency". https://trac.ffmpeg.org/wiki/StreamingGuide#Latency.

[7] DASH Industry Forum, "Low-Latency Streaming," GitHub. https://github.com/Dash-Industry-Forum/dash.js/wiki/Low-Latency-streaming.

[8] M. Vidonis, The Broadcast Bridge, "Measuring Latency on Incoming Video Streams", 2023. https://updates.broadcastbridge.app/measuring-latency-on-incoming-video-streams/.

[9] M. Lim et al., "When they go high, we go low: low-latency live streaming in dash.js with LoL", 2020. https://dl.acm.org/doi/abs/10.1145/3339825.3397043.

[10] A.Bentaleb et al., "Want to play DASH?: a game theoretic approach for adaptive streaming over HTTP," 2018, https://dl.acm.org/doi/abs/10.1145/3204949.3204961.

[11] JSQR-ES6. npm. (2019). https://www.npmjs.com/package/jsqr-es6