Performance of Low-Latency HTTP-based Streaming Players

Bo Zhang
Brightcove, Inc
Boston MA USA
bzhang@brightcove.com

Thiago Teixeira
Brightcove, Inc.
Boston MA USA
tteixeira@brightcove.com

Yuriy Reznik
Brightcove, Inc.
Seattle WA USA
yreznik@brightcove.com

ABSTRACT

Reducing end-to-end streaming latency is critical to HTTP-based live video streaming. There are currently two technologies in this domain: Low-Latency HTTP Live Streaming (LL-HLS) and Low-Latency Dynamic Adaptive Streaming over HTTP (LL-DASH). Many players support LL-HLS and/or LL-DASH protocols, including Apple's AVPlayer, Shaka player, HLS.js Dash.js, and others. This paper is dedicated to the analysis of the performance of low-latency players and streaming protocols. The evaluation is based on a series of live streaming experiments, repeated using identical video content, encoders, encoding profiles, and network conditions, emulated by using traces of real-world networks. Several performance metrics, such as average stream bitrate, the amounts of downloaded media data, streaming latency, as well as buffering and stream switching statistics are captured and reported in our experiments. These results are subsequently used to describe the observed differences in the performance of LL-HLS and LL-DASH-based players.

CCS CONCEPTS

• Multimedia information systems → Multimedia streaming

KEYWORDS

HTTP Adaptive Streaming, HLS, DASH, Low-Latency live streaming, Video players, Performance evaluation

ACM Reference format:

Bo Zhang, Thiago Teixeira, and Yuriy Reznik. 2021. Performance of Low-Latency HTTP-based Streaming Players. In *Proceedings of ACM Multimedia Systems conference (MMSys'21). ACM, New York, NY, USA, 6 pages.* https://doi.org/10.1145/1234567890

1 INTRODUCTION

The operation under unknown or changing network conditions has been one of the most fundamental challenges that adaptive bitrate streaming systems have been trying to solve since their birth in the 1990s [20-22]. This challenge still exists today, although in a somewhat simplified setting, allowed by the use of HTTP-based Adaptive Streaming (HAS) architectures [23, 39-47]. In such architectures, the network adaptation logic resides in streaming clients, effectively driving the selection and loading of segments of media streams. In the past decade, many advanced methods have been proposed for the design of stream selection algorithms. This includes throughput-based methods [39-40], buffer-level-based heuristics [31-43], control-theoretic approaches [44-45], as well as machine-learning algorithms [5-6].

However, the comparison of different network adaptation algorithms presents a technical challenge. Some of the proposed algorithms have only been evaluated in a simulated environment using very basic bandwidth throttling tools in web browsers. Such tools can only control video players' download bandwidth at the application layer and have no means for accurately simulating highly fluctuating network bandwidth changes or packet loss statistics present, for example, in mobile networks.

To better evaluate HAS systems, references [29-34] proposed frameworks for measuring video streaming Quality of Experience (QoE) using real networks or fine-controlled network links. For instance, Talon et al. [29] have implemented several HAS players and evaluated them in a campus network from different performance perspectives. Ayad et al. [32] took a similar approach and conducted a practical and in-depth evaluation of HAS players. Particularly, the authors of [32] have built an experimental framework emulating wired network links using Netem and Linux Traffic Control (TC). Their experimental study and code-level analysis revealed how different HAS players operate in detail. This study was limited to the use of wired networks, however. References [33-35] have proposed a framework for automating video streaming testing and QoE evaluation. The framework integrates with the Mobile Broadband Networks in Europe (MONROE) project. The players run in docker containers with managed network connections and the environment metadata collection functionalities that are built into MONROE nodes. The framework enables running experiments on a cloud infrastructure. These proposed frameworks, however, focus more on automation and simplification of player evaluation, but they do not ensure a fair comparison of different players, because there is no guarantee that different players experience the same network conditions. Raca et al. [31] have proposed DASHbed, a framework for simulating large-scale empirical evaluation of DASH players. However, the mobile network traces it relies upon [48] have limited sampling granularity and thus don't capture the essential fine-grain dynamics of such networks.

To ensure a more accurate and fair evaluation of different players, in this paper we introduce a custom evaluation framework, incorporating the Mahimahi network emulator [19, 36-38]. Our framework guarantees a fair comparison of different players by replaying the same network traces across all the playback sessions. This allows the comparison of multiple players side by side under the same conditions. The Mahimahi network simulator can accurately emulate mobile network links using the physical network traces recorded from different mobile operators. Specifically, in this study, we will use network traces from T-Mobile and Verizon 4G LTE networks [19].

In recent years, reducing the latency of live streaming has received tremendous interest in both the research community and the industry. DVB low-latency DASH [10] was the first effort in the low-latency HAS domain which has later evolved into the low-latency DASH/CMAF recommendation [9]. Apple also released the Low-Latency HLS specification [7] in 2020. The low-latency DASH and HLS technologies share many common design principles: first, divide a whole video segment into smaller chunks, then deliver a segment as soon as the first chunk becomes available to download. In this way, a player can start rendering a segment right after its first chunk is received. As more chunks are produced, they are sent to the players one by one. As a result, the latency is reduced from one whole segment long (e.g., 10 seconds for legacy v3 HLS) to only one chunk long (e.g., a few hundred milliseconds).

Considering that low-latency streams are delivered chunk by chunk, and a much smaller buffer is available at the client-side, estimating the network bandwidth and making stream adaptation decisions becomes more challenging. Recent methods proposed to address these challenges include references [24-27] as well as references [5,6]. The last two references describe LoL and L2All machine-learning-based algorithms prototyped and included in the DASH.js streaming player [4]. Other variants of low-latency adaptation algorithms can be found in LL-HLS streaming players such as HLS.js [1], Shaka player [2], and Apple's AVPlayer [3].

This paper is dedicated to the evaluation of low-latency DASH and HLS players and their respective ABR adaptation algorithms in a common evaluation framework, ensuring an accurate and fair comparison. In Section 2, we will describe our evaluation setup. In Section 3, we will present and discuss the results of our measurements. In Section 4, we will drive conclusions.

2 EXPERIMENT SETUP

2.1 Streaming Tool-Chains

The overall diagrams of the tool-chains that we used for LL-HLS and LL-DASH streaming are shown in Figures 1 and 2 respectively. To generate LL-HLS streams we used Apple's HLS

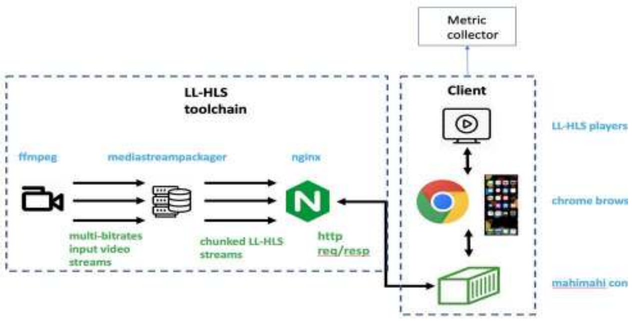

Figure 1: Tool-chain used for testing of LL-HLS players

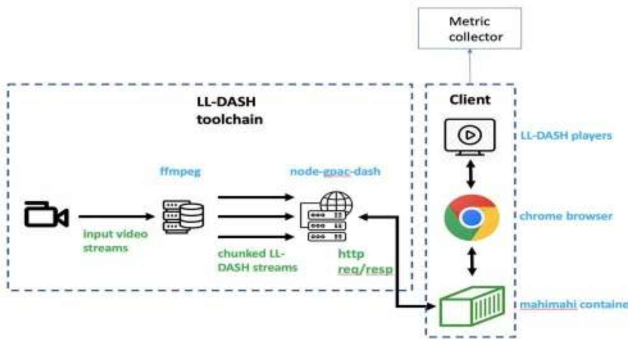

Figure 2: Tool-chain used for testing of LL-DASH players

reference tools [8] and FFmpeg [13]. To generate LL-DASH streams we used OBS studio [12], FFmpeg [13], and node-gpac-dash [14]. Additional details about our setups can be found in [17-18]. The LL-HLS stream was served dynamically by an NGINX web server [15]. The LL-DASH stream was served dynamically by node-gpac-dash [14].

As shown in Figures 1 and 2, the input video streams were sent to the low-latency packagers (mediastreamsegmenter [8] for LL-HLS, and FFmpeg [12] for LL-DASH). The outputs of low-latency packagers are the chunked video segments and manifest files informing the players on how to consume the streams in low-latency mode. Next, the output stream files are served by the low-latency media servers (lowLatencyHLS.php [8] for LL-HLS, node-gpac-dash [14] for LL-DASH) to players in a chunked manner.

On the player side, the web-based players run on the Chrome browser, and the iOS native player (HLS) runs on the AVPlayer framework on iOS. The Chrome browser and the AVPlayer run inside the Mahimahi container and connect to the media server via an emulated virtual network interface.

2.2 Test Content and Encoding Parameters

As a test video sequence, we used a 1080p version of the Big Buck Bunny [11] video. This sequence was looped to enable continuous testing. For streaming, 3 live transcoded variant streams have been subsequently generated, with parameters listed in Table 1.

To minimize fluctuations of encoding bitrates from their declared targets, constant bitrate (CBR) encoding mode has been utilized. To minimize encoding delays, H.264 encoder operating in Baseline profile has been used. Lookahead processing disabled.

Table 1: Encoding profile parameters

Parameter	Rendition 1	Rendition 2	Rendition 3
Bitrate (kbps)	279	925	1253
Frame rate (fps)	30	30	30
Video resolution (pels)	320x180	640x360	768x432
Seg. duration (sec)	4	4	4
Chunk duration (sec)	1	1	1
Video codec	H.264	H.264	H.264
Video codec profile	Baseline	Baseline	Baseline
Media format	ISOBMFF	ISOBMFF	ISOBMFF

The segment lengths and fragment durations were set to 4 sec and 1 sec, respectively, matching the default values used in Apple's streaming tools for LL-HLS [8]. The same encoding parameters have been used for the generation of both LL-DASH and LL-HLS streams.

The overall session duration that we used to test the performance of each player under each network was 10 minutes. Given selected chunk and fragment durations, this has allowed about 600 chunks or equivalently 150 segments to be downloaded per session.

2.3 Streaming Players

We have evaluated 6 implementations of low-latency streaming players. For LL-HLS, we used HLS.js [1], Shaka player [2], and Apple's AVPlayer [3]. For LL-DASH, we used Dash.js with three different low-latency ABR algorithms: Dash.js original [4], Dash.js with LoL algorithm [5], and Dash.js with L2All algorithm [6]. We have implemented simple test applications for all the players. The applications were built using the latest player SDK releases as available in December 2020.

2.4 Performance Metrics

The reporting of metrics indicative of live streaming latency, playback speed, and re-buffering events has been instrumented in the video player applications. Other metrics such as stream bitrate, video resolution, and media data downloaded have been derived from the streaming servers' access logs. The processing of all collected metrics was done offline.

The player's streaming latency was calculated by following the method described in [9], which is common for both LL-DASH and LL-HLS. Essentially, at any time point, we take the difference between the elapsed presentation time and the elapsed wall clock time, from the beginning of a streaming session (Eq. 1):

$$PL = (WC - WCA) - (PT - PTA)/TS \qquad \text{(Eq. 1)}$$

where PL represents the live Presentation Latency, WC and PT represent the current Wall Clock time and the current Presentation Time, respectively. WCA and PTA represent the beginning wall clock time and the beginning presentation time, respectively. For LL-DASH, the above values have been obtained from the ProducerReferenceTime [9] element embedded in an MPD file, and W3C HTML5 video currentTime API [16], and/or a DASH MPD file. For LL-HLS, these values have been derived from the HLS m3u8 file and currentTime API.

Table 2: List of performance metrics collected

Metrics	Impact domain(s)
Streaming bitrate (kbps)	Efficiency, QoE
Video resolution (height)	QoE
Streaming latency (sec)	Latency, QoE
Variation of playback speed	QoE
Frequency of stream switches	QoE
Frequency of rebuffering events	QoE
Downloaded media data (Mbytes)	Efficiency
Media objects (chunks or segments) downloaded	Efficiency

The number of re-buffering events and the players' playback speed have been obtained by using the waiting event API [13] and the playbackRate API [13] respectively.

The playback speed variation is calculated as the Euclidean distance of all the measured playback speeds relative to the native speed (which equals 1):

$$playbackSpeedVariation = \frac{1}{n}\sqrt{\sum_{i=1}^{N}(s_i - 1)^2} \qquad \text{(Eq 2)}$$

The parameter N used in this formula denotes the number of playback speed measurements conducted during the session.

All other metrics including stream bitrate, video resolution, media data downloaded, number of bitrate switches have been derived from the server logs. The full list of metrics collected in our test system is summarized in Table 2.

2.5 Performance Metrics

We used the Mahimahi network emulator [19] to emulate various network conditions at the network interface level. Mahimahi is essentially a Linux container that can run an application inside of it. An application inside Mahimahi connects to the outside world through a virtual network interface that sends and receives bytes according to the running downlink and uplink traces. This way, the capacity of the network interface is limited by the running trace. We used traces that have been recorded from real-world mobile networks. When we run the test players inside Mahimahi, the player download speed is limited by the capacity of the virtual interface. Unlike using bandwidth throttling features in web browsers, Mahimahi provides more faithful network emulation by using real-world traces and throttling bandwidth at the network interface level. Additionally, the same network traces are replayed for all the test sessions. This allows a fair comparison of different players.

We have evaluated the test players using two 4G-LTE network traces from T-Mobile and Verizon respectively [19]. We provide visualizations of these traces in Figure 3. In Table 3 we further list basic statistics associated with them. We note that these network traces are quite challenging, capturing cases of mobile handoffs and other forms of impairments that may happen in practice.

Table 3: Network bandwidth statistics

Bandwidth statistics	T-Mobile	Verizon
Average bitrate (kbps)	1607.43	1323.97
St. deviation of bitrate (kbps)	1147.60	1075.80
Minimum bitrate (kbps)	148.5	1.178
Maximum bitrate (kbps)	7545	5433

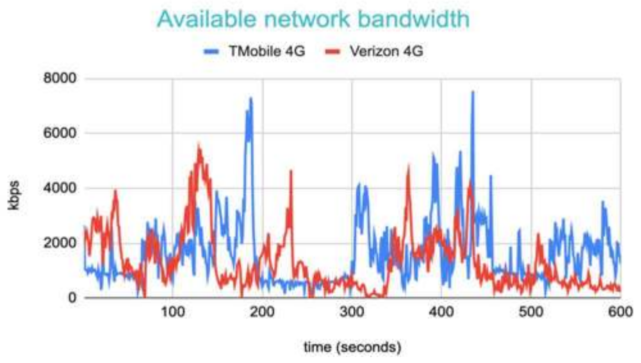

Figure 3: Network bandwidth traces used in the experiments

3 THE RESULTS

3.1 Results for Verizon 4G LTE network

First, we review the results obtained by using traces of the Verizon 4G LTE network. Table 4 offers summary metrics. The dynamics of changes of bitrates and latencies achieved by LL-HLS players are illustrated in Figure 4 and Figure 5 respectively.

Based on Figure 5, we note, that the latency of the Shaka player was almost flat across the entire session, with an average value of 7.28 seconds. This is higher than the 4.32 seconds achieved by the HLS.js but is significantly lower than the 15.96 seconds achieved by the AVPlayer. Even though HLS.js has lower average latency, its behavior across the session is not stable: it varies very significantly, producing large spikes of latency in the mid-session. Such spikes, in our view, should be avoided.

When latency varies, a player has to play faster or slower than the stream's native speed to stay on the stream's live edge. This is evidenced by the playback speed variation numbers reported in Table 4. The lower values of playback speed variation are indicative of better QoE.

Based on Figure 5, we also note that AVPlayer was able to stream at low latency (about 4.8 seconds) in the first 260 seconds. When the first major bandwidth fluctuation hit (i.e., time interval [250 - 340] in Figure 3), AVPlayer experienced buffer underruns and was not able to maintain low latency after re-buffering and resuming the playback. The latency went up to 12 seconds, then to 17 seconds until the end.

In terms of streaming bitrate (cf. Figure 4), we note that Shaka player has achieved the highest average value (1228 kbps) over the 10 minutes session, followed by AVPlayer (1136 kbps) and HLS.js (849 kbps). As also seen from Figure 4, the Shaka player was able to stream at the highest bitrate most of the time while HLS.js often hesitated to switch up to higher bitrates, or it switched down to lower bitrates when other players still stuck to higher bitrates. This can be seen particularly in the time interval [350 – 470]. In general, a higher average bitrate should result in better QoE to viewers.

Next, we note that the Shaka player has the lowest number of bitrate switches, only twice in 10 minutes. AVPlayer switched 130 times, most of which happened during the time interval [470 - 570]. During this time, the available bandwidth was not only low but also fluctuated significantly (Figure 3). In reaction to the dynamic network condition, AVPlayer adapted quickly by switching bitrate for almost every segment it downloaded. Though AVPlayer was prompt in switching up to a higher bitrate when the available bandwidth allows, it was forced to switch back to lower bitrate when the bandwidth dropped. Generally, overly frequent switching may hurt QoE.

Table 4: Summary of players metrics – Verizon 4G LTE

Metrics	LL-HLS players			LL-DASH players		
	HLS.js	Shaka	AVplayer	DASH.js	LoL	L2All
Avg. bitrate (kbps)	849	1228	1136	1165	595	1073
Avg. height (pixels)	328	426	404	410	262	387
Avg. latency (secs)	4.32	7.28	15.96	3.71	3.2	3.9
Var. playback speed	3.97	0	0	0.19	0.39	0.44
# of switches	48	2	130	6	29	3
# of rebufferings	36	12	2	5	79	56
Downloaded MBs	85	90	99	88	45	81
Downloaded objects (chunks + segments)	673 (662+ 11)	587 (587 + 0)	669 (611 + 58)	152	151	152

Table 5: Summary of players metrics – T-Mobile 4G LTE

Metrics	LL-HLS players			LL-DASH players		
	HLS.js	Shaka	AVplayer	DASH.js	LoL	L2All
Avg. bitrate (kbps)	783	1043	1037	1225	537	1251
Avg. height (pixels)	311	378	378	426	248	432
Avg. latency (secs)	5.82	4.48	7.78	3.06	1.78	2.28
Var. playback speed	3.62	0	0	0.23	1.62	0.42
# of switches	50	8	72	4	28	0
# of rebufferings	43	18	1	1	69	13
Downloaded MBs	156	81	92	93	42	94
Downloaded objects (chunks + segments)	965 (743 + 222)	621 (621 + 0)	703 (698 + 5)	151	152	151

Another important QoE factor is the number of buffer underruns and consequently re-buffering events. AVPlayer recorded the least re-buffering events, however, this is because its average live latency (15.96 seconds) was much higher than the other two. Higher latency means more buffering, and higher resilience to bandwidth fluctuation. HLS.js and Shaka players were playing closer to the stream's live edge, as a result, they are more prone to re-buffering (36 times for HLS.js, 12 times for Shaka player) than AVPlayer. Among these three, Shaka player seems to achieve a better balance between latency and re-buffering.

Finally, we look at the amount of downloaded data by the players in 10 minutes sessions. Shaka player downloaded 587 media objects, all of them were video chunks which means Shaka player maintained low latency throughout the session. Since 600 chunks should be downloaded in 10 minutes, Shaka skipped 13 chunks. AVPlayer downloaded 669 media objects including 611 chunks and 58 whole segments. The whole segments were downloaded when AVPlayer was not able to download partial chunks on the live edge, and fell back to download earlier whole segments. HLS.js downloaded 662 chunks and 11 whole segments. Unlike Shaka player, AVPlayer and HLS.js downloaded more than 600 media objects. In terms of downloaded data in bytes, Shaka downloaded 90.16 MB, more than HLS.js (85.36 MB) because of

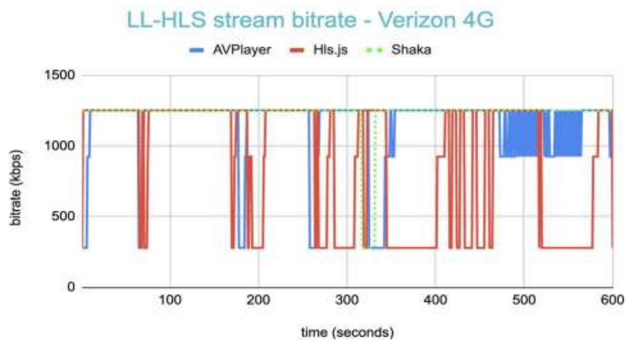

Figure 4: Bitrate over time – LL-HLS / Verizon 4G

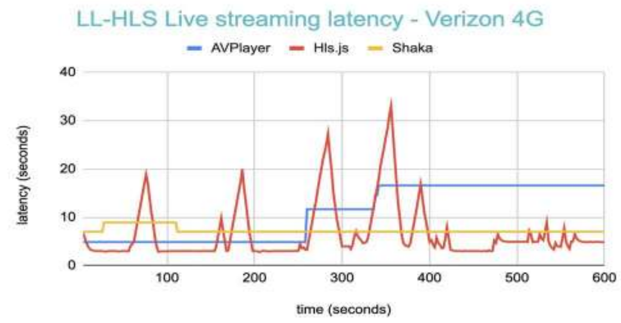

Figure 5: Live latency over time – LL-HLS / Verizon 4G

its higher average stream bitrate, less than AVPlayer (98.52 MB) because of fewer downloaded media objects.

Next, we switch our attention to the LL-DASH players. The dynamics of changes of bitrates and latencies achieved by such players are illustrated in Figure 6 and Figure 7 respectively.

As can be observed from Table 4 and Figure 6, the original Dash.js player achieved a much higher stream bitrate (1165 kbps) than LoL (595 kbps), and L2ALL (1073 kbps) variants. LoL had significantly more bitrate switches (29 times) than Dash.js (6 times) and L2ALL (3 times). LoL achieved the lowest average bitrate and the number of bitrate switches. However, as seen from Figure 7, LoL was also able to achieve a lower average latency (3.2 seconds) than Dash.js (3.71 seconds) and L2ALL (3.9 seconds). The LoL player re-buffered 79 times, which is higher than L2ALL (56 times) and the original Dash.js (5 times).

The original Dash.js also had the lowest playback speed variation (0.19). In terms of downloaded data, all three LL-DASH players downloaded about 150 objects, all of them were whole segments. This is because LL-DASH players depend on the streaming servers to push segments chunk by chunk using HTTP/1.1 chunked transfer encoding [9], instead of requesting individual chunks as LL-HLS players do. In other words, LL-DASH players only request whole segments. Finally, we observe that the number of segments downloaded by all LL-DASH players was almost the same and that the total amounts of data downloaded were proportional to the average bitrates used by such players.

3.2 Results for T-Mobile 4G LTE network

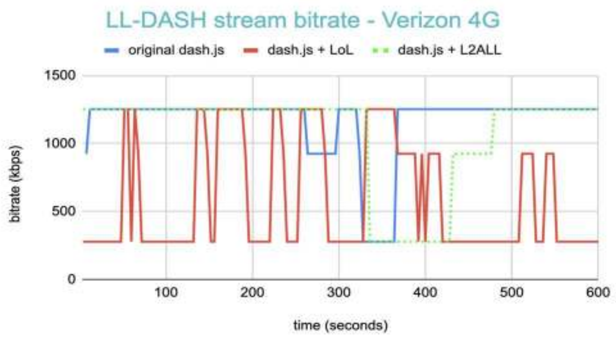

Figure 6: Bitrate over time – LL-DASH / Verizon 4G

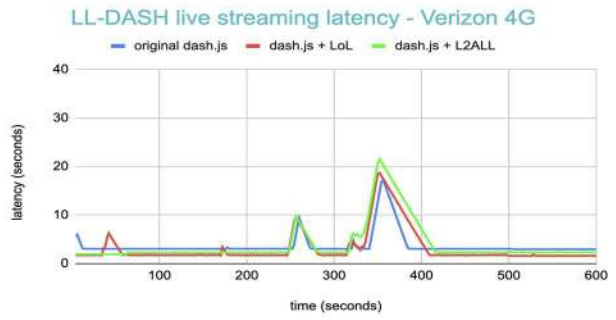

Figure 7: Live latency over time – LL-DASH / Verizon 4G

We next review the results obtained by using traces of the T-Mobile 4G LTE network. Table 5 offers summary metrics. The dynamics of changes of bitrates and latencies achieved by LL-HLS players are illustrated in Figure 8 and Figure 9 respectively.

Based on Table 5 and Figure 8, we notice that Shaka player and AVPlayer have achieved higher average bitrates than HLS.js. This can be observed in multiple intervals in Figure 8, where HLS.js seems struggling to select the right bitrate while other players were able to play at higher bitrates.

Based on Figure 9, we also note that HLS.js and Shaka players have achieved lower latency than AVPlayer. The latency line of AVPlayer was low and flat for more than half of the session but went up higher towards the end. Like when the Verizon trace was used, HLS.js had variable latency throughout the session. Shaka player had both lower and more stable latency (averaged at 7.78 seconds) when compared to the other two.

We also note that HLS.js downloaded a lot more media objects (965) during the session than the other two players, and also more than itself when the Verizon trace was used. This is probably due to more re-buffering events and bitrate switches it had experienced in the T-Mobile network. Because of more media object downloads, HLS.js also downloaded more bytes (155.54 MB).

In terms of the number of re-buffering and bitrate switches, the Shaka player again experienced fewer re-buffering events (18 times) and the least switches (only 8 times). Finally, HLS.js had a large value of playback speed variation due to its highly variable latency. It was observed several times that HLS.js had to play at 1.5 times speed to catch up to the live edge.

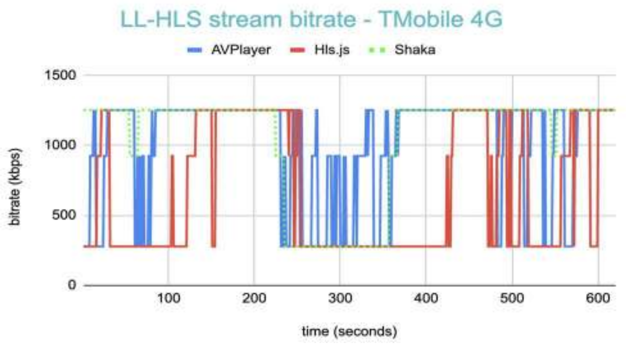

Figure 8: Bitrate over time – LL-HLS / T-Mobile 4G

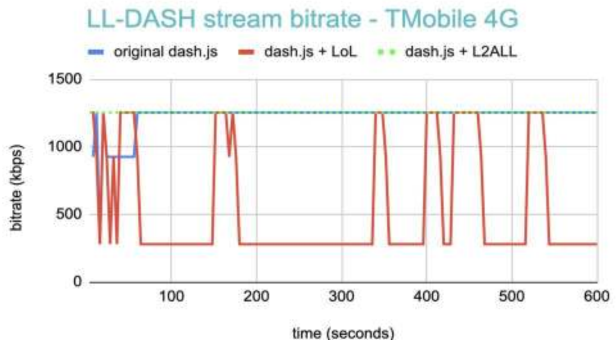

Figure 10: Bitrate over time – LL-DASH / T-Mobile 4G

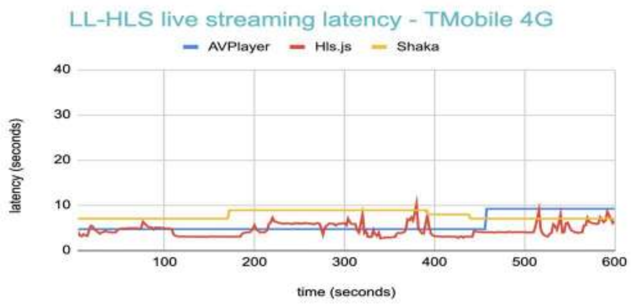

Figure 9: Live latency over time – LL-HLS / T-Mobile 4G

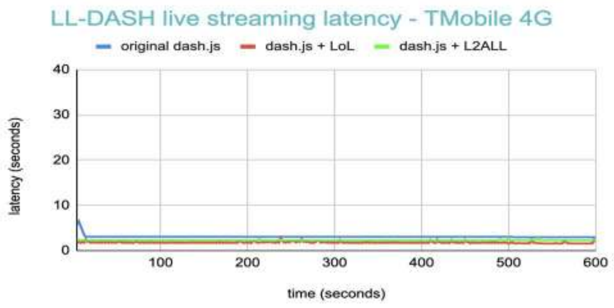

Figure 11: Live latency over time – LL-DASH / T-Mobile 4G

Finally, we look at the behavior of LL-DASH players. The dynamics of changes of bitrates and latencies achieved by such players are illustrated in Figure 10 and Figure 11 respectively.

Based on Table 5 and Figure 10, we note that the original Dash.js and L2ALL were able to pull content encoded at much higher bitrates than LoL player: 1224.83 kbps and 1250.83 kbps vs 537.21 kbps respectively. LoL player used the lowest bitrate for the majority of the session while the other two players were at the highest bitrate.

Based on Table 5 and Figure 11, we also note that LoL and L2All both have achieved lower latency than the original Dash.js (3.06 seconds), at 1.78 seconds and 2.28 seconds, respectively. It appears that these ABR algorithms seem to favor lower latency than other metrics when choosing the bitrate.

In terms of media downloads, all three players have received about 150 whole segments. The original Dash.js and L2ALL downloaded more bytes for their higher average bitrate.

In terms of the number of re-buffering events and bitrate switches, the original Dash.js performed the best among the three. It had the lowest number of re-buffering events (only once) and very few bitrate switches (only 4 times). Finally, the original Dash.js had the smallest playback speed variation (0.23), lower than LoL (1.62) and L2Aall (0.42).

In general, we observe that the original Dash.js had the best performance among the three players. Though L2All performed slightly better in bitrate, latency, and bitrate switch frequency, it also has experienced more re-buffering events.

4 CONCLUSIONS

In this study, we have evaluated 6 different LL-HLS and LL-DASH players, using multi-bitrate-encoded low-latency HLS and DASH streams and network emulation process, employing traces obtained for T-Mobile 4G LTE and Verizon 4G LTE networks.

We noted that both LL-HLS and LL-DASH systems were able to operate at significantly lower latencies compared to the traditional HLS and DASH streaming. Thus, in the majority of cases, we have observed LL-DASH players can maintain latencies in the range of 3-4 seconds, except for a couple of short segments in streaming over the Verizon 4G network, when latencies have increased to almost 20 seconds. For LL-HLS players, we have observed somewhat broader variation in streaming latencies across different player implementations, but with the majority of data points fitting in the 4-10 second range.

However, we also noted many deficiencies in the behavior of both LL-DASH and LL-HLS players when operating in our testing framework. These include:

- high stream switching and buffering rates,
- the inability of some players to select high renditions,
- the inability of some players to maintain playback speed,
- more requests sent to the CDNs (especially for LL-HLS),
- the inability of some players to maintain low delay, etc.

Based on these observations, we believe that there is certainly more work that still needs to be done to improve player algorithms and other aspects of the system to make low-latency streaming more robust and suitable for mass deployment in practice.

REFERENCES

[1] Hls.js player, https://github.com/video-dev/hls.js/

[2] Shaka player, https://github.com/google/shaka-player

[3] AVFoundation, https://developer.apple.com/av-foundation/

[4] Dash.js player, https://github.com/Dash-Industry-Forum/dash.js

[5] M. Lim, M. N. Akcay, A. Bentalab, A. C. Begen, R. Zimmermann, "When they go high, we go low: low-latency live streaming in dash.js with LoL," in *Proc ACM Multimedia Systems Conference (MMsys'20)*, Online, June 8-11, 2020.

[6] T. Karagkioules, R. Mekuria, D. Griffioen, A. Wagenaar, "Online Learning for Low-Latency Adaptive Streaming," in *Proc ACM Multimedia Systems Conference (MMsys'20),* Online, June 8-11, 2020.

[7] HTTP Live Streaming 2nd Edition, https://tools.ietf.org/html/draft-pantos-hls-rfc8216bis-08, 2019

[8] Apple's HTTP Live Streaming Tools, https://developer.apple.com/documentation/http_live_streaming/about_apple_s_http_live_streaming_tools

[9] DASH Industry Forum, "Low-Latency Modes for DASH", https://dashif.org/docs/CR-Low-Latency-Live-r8.pdf

[10] ETSI technical specification, "MPEG-DASH Profile for Transport of ISO-BMFF Based DVB Services over IP Based Networks", https://www.etsi.org/deliver/etsi_ts/103200_103299/103285/01.03.01_60/ts_103285v010301p.pdf

[11] Blender Foundation, Big Buck Bunny video, https://download.blender.org/

[12] Open Broadcast Software, https://obsproject.com/

[13] FFMPEG, https://www.ffmpeg.org/

[14] DASH Low Latency Server, https://github.com/maxutility2011/node-gpac-dash

[15] NGINX web server, https://www.nginx.com/

[16] HTML5 video, https://www.w3.org/TR/2011/WD-html5-20110113/ video.html

[17] B. Zhang, "Setting up your Own Low-Latency HLS Server to Stream from any Source Inputs", https://bozhang-26963.medium.com/setting-up-your-low-latency-hls-server-to-stream-from-any-source-inputs-de1e757a6688

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

[19] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, H. Balakrishnan, "Mahimahi: accurate record-and-replay for HTTP," in *Proc. USENIX Annual Technical Conference (USENIX ATC '15)*, Santa Clara, CA, July 8-10, 2015.

[20] D. Wu, Y. Hou, W. Zhu, Y-Q. Zhang, and J. Peha, "Streaming video over the internet: approaches and directions," *IEEE Trans. CSVT*, vol. 11, no. 3, 2001, pp. 282-300.

[21] G. Conklin, G. Greenbaum, K. Lillevold, A. Lippman, and Y. Reznik, "Video coding for streaming media delivery on the internet," *IEEE Trans. CSVT*, vol. 11, no. 3, 2001, pp. 269-281.

[22] B. Girod, M. Kalman, Y.J. Liang, and R. Zhang, "Advances in channel-adaptive video streaming," *Wireless Comm. and Mobile Comp*., vol. 2, no. 6, 2002, pp. 573-584.

[23] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, R. Zimmermann, "A Survey on Bitrate Adaptation Schemes for Streaming Media Over HTTP," in *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, 2019, pp. 562-585.

[24] A. Bentaleb, A. Begen, S. Harous, R. Zimmermann, "Data-Driven Bandwidth Prediction Models and Automated Model Selection for Low Latency," *IEEE Transactions on Multimedia*, August 2020.

[25] A. Bentaleb, C. Timmerer, A. C. Begen, R. Zimmermann, "Bandwidth prediction in low-latency chunked streaming," in *Proc. ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '19)*, Amherst, MA, June 21, 2019, pp. 7–13.

[26] A. Bentaleb, C. Timmerer, A. C. Begen, R. Zimmermann, "Performance Analysis of ACTE: A Bandwidth Prediction Method for Low-latency Chunked Streaming," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol 16, no. 2s, 2020.

[27] I. Ozcelik, C. Ersoy, "Low-Latency Live Streaming Over HTTP in Bandwidth-Limited Networks," *IEEE Communications Letters*, vol. 25, no. 2, 2021, pp. 450-454.

[28] K. Durak, M. Akcay, Y. Erinc, B. Pekel, A. C. Begen, "Evaluating the Performance of Apple's Low-Latency HLS," in *Proc. IEEE Int. Workshop on Multimedia Signal Processing (MMSP'20)*, September 21-24, 2020, pp.1-6.

[29] D. Talon, L. Attanasio, F. Chiariotti, M. Gadaleta, A. Zanella, M. Rossi, "Comparing DASH Adaptation Algorithms in a Real Network Environment," in *European Wireless 2019; 25th European Wireless Conference*, 2019.

[30] C. Storck, F. Figueiredo, "A Performance Analysis of Adaptive Streaming Algorithms in 5G Vehicular Communications in Urban Scenarios," in *Proc. of IEEE Symposium on Computers and Communications (ISCC)*, 2020, pp.1-7.

[31] D. Raca, Y. Sani, C. J. Sreenan, J. J. Quinlan, "DASHbed: a testbed Framework for Large Scale Empirical Evaluation of Real-Time DASH in Wireless Scenarios," in *Proc. ACM Multimedia Systems Conference (MMSys'19)*, Amherst, MA, June 18 - 21, 2019, pp285–290.

[32] I. Ayad, Y. Im, E. Keller, S. Ha, "A Practical Evaluation of Rate Adaptation Algorithms in HTTP-based Adaptive Streaming", *Elsevier Computer Networks* vol. 133, 2018, pp. 90-103

[33] C. Midoglu, A. Zabrovskiy, O. Alay, D. Holbling-Inzko, C. Griwodz, C. Timmerer, "Docker-Based Evaluation Framework for Video Streaming QoE in Broadband Networks," in *Proc. ACM International Conference on Multimedia*, Nice, France, 21 - 25 October 2019, pp. 2288–2291.

[34] B. Taraghi, A. Zabrovskiy, C. Timmerer, H. Hellwagner, "CAdViSE: Cloud-based Adaptive Video Streaming Evaluation Framework for the Automated Testing of Media Players," in *Proc ACM Multimedia Systems Conference (MMsys'20)*, Online, June 8-11, 2020.

[35] A. Zabrovskiy, E. Kuzmin, E. Petrov, C. Timmerer, C. Mueller, "AdViSE: Adaptive Video Streaming Evaluation Framework for the Automated Testing of Media Players," in *Proc ACM Multimedia Systems Conference (MMsys'17)*, Taipei, Taiwan, June 20-23, 2017.

[36] A. Mondal, B. Palit, S. Khandelia, N. Pal, J. Jayatheerthan, K. Paul, N. Ganguly, Sandip Chakraborty, "EnDASH - A Mobility Adapted Energy Efficient ABR Video Streaming for Cellular Networks," in *Proc. IFIP Networking Conference*, 2020, pp. 127-135.

[37] G. Ribezzo, L. D. Cicco, V. Palmisano, S. Mascolo, "A DASH 360° immersive video streaming control system," in *Internet Tech. Letters*, vol. 3, no. 5, 2020.

[38] S. Sengupta, N. Ganguly, S. Chakraborty, P. De, "HotDASH: Hotspot Aware Adaptive Video Streaming using Deep Reinforcement Learning," in *Proc IEEE International Conference on Network Protocols (ICNP'18*), 2018, pp.165-175.

[39] Jiang, V. Sekar, H. Zhang, "Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE," in *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, Feb. 2014, pp. 326-340.

[40] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, D. Oran, "Probe and Adapt: Rate-Adaptation for HTTP Video Streaming at Scale," *IEEE Journal on Selected Areas in Communications*, Vol. 32, No. 4, April 2014, pp. 719-733.

[41] K. Spiteri, R. Urgaonkar, R. K. Sitaraman, "BOLA: Near-Optimal Bitrate Adaptation for Online Videos," in *Proc. Annual IEEE International Conference on Computer Communications (INFOCOM'16)*, 2016, pp. 1-9.

[42] T. Huang, R. Johari, N. Mckeown, M. Trunnell, M. Watson, "A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service," in *Proc. ACM SIGCOMM*, 2014, pp. 187–198.

[43] K. Spiteri, R. Sitaraman, D. Sparacio, "From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player," *ACMTrans. Multimedia Comput. Commun. Appl.*, vol. 15, no. 2s, Article 67, 2019.

[44] S. Hesse, "Design of scheduling and rate-adaptation algorithms for adaptive HTTP streaming," in *Proc. SPIE 8856, Applications of Digital Image Processing XXXVI*, 88560M, 2013.

[45] C. Zhou, X. Zhang, L. Huo, and Z. Guo, "A control-theoretic approach to rate adaptation for dynamic HTTP streaming," in *Proc. Visual Comm. Image Processing (VCIP'12)*, San Diego, CA, 2012, pp. 1-6.

[46] X. Yin, A. Jindal, V. Sekar, B. Sinopoli, "A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP," *SIGCOMM Comput. Commun. Rev.* no. 45, vol. 4, 2015, pp. 325–338.

[47] T. Huang, C Ekanadham, A. Berglund, Z. Li, "Hindsight: evaluate video bitrate adaptation at scale," in *Proc. ACM Multimedia Systems Conference (MMSys'19)*, Amherst, MA, June 18 - 21, 2019, pp 86–97.

[48] D. Raca, J. Quinlan, A. Zahran, and C. Sreenan. "Beyond throughput: A 4G LTE dataset with channel and context metrics," in *Proc. ACM Multimedia Systems Conference (MMSys '18)*, New York, NY, USA, 2018, pp 460–465.