Performance of Low-Latency HTTP-based Streaming Players

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

Nabajeet Barman
Brightcove, Inc.
Boston, MA, USA
nbarman@brightcove.com

Yuriy Reznik
Brightcove, Inc.
Boston, MA, USA
yreznik@brightcove.com

ABSTRACT

Reducing end-to-end streaming latency is critical to HTTP-based live video streaming. There are currently two leading technologies in this domain: Low-Latency HTTP Live Streaming (LL-HLS) and Low-Latency Dynamic Adaptive Streaming over HTTP (LL-DASH). Many popular streaming players, including HLS.js, DASH.js, Video.js, Shaka player, THEO player, and others, now support these protocols. Furthermore, with some players, there are now several alternative adaptation algorithms. This paper evaluates the performance of such low-latency players and their adaptation methods. The evaluation is based on a series of live streaming experiments, repeated using identical video content, encoders, encoding profiles, and network conditions, emulated using traces of real-world networks. Several system performance metrics, such as average stream bitrate, the amounts of downloaded data, streaming latency, and buffering and switching statistics, have been captured and reported. These results are used to describe the observed differences in the performance of low-latency players and systems.

KEYWORDS

HTTP Adaptive Streaming, HLS, DASH, Low-Latency Streaming

1 INTRODUCTION

In the past few years, the video streaming industry has seen immense interest in Low-Latency streaming protocols, targeting sub-5-second end-to-end delay, comparable to the delay in live broadcast TV systems. Such low delay is critical for streaming live sports, gaming, online learning, interactive video applications, etc.

As well known, the delay in the conventional live streaming technologies such as HLS [1] and DASH [2] is much longer. It is caused by relatively long (4-10 seconds) segments and a segment-based delivery model, requiring complete delivery of each media segment before playback. Combined with buffering strategies used by the HLS or DASH streaming clients, this typically produces delays of 10 to 30sec, or even longer.

Low-Latency HLS (LL-HLS) [3,4] and Low-Latency DASH (LL-DASH) [2,5,6] are the recent evolutions of the HLS and DASH standards, designed to reduce the latency. They employ a new encoding and transmission process, effectively splitting each segment into several (typically 4-10) chunks and then using such "chunks" for transmission. Since each "chunk" is significantly shorter than a segment, this reduces the delay in the streaming system.

The support for LL-DASH and LL-HLS technologies is already quite broad, with many available streaming players, encoders, and packaging tools. The available implementations LL-DASH and LL-HLS players include Apple's AVPlayer [7], HLS.js [8], DASH.js [9], Video.js [10], Shaka player [11], and THEO player [12]. With some players, such as DASH.js and HLS.js there are also several advanced adaptation algorithms, such as Low on Latency (LoL) [13], LoL+ [14], and Learn2Adapt-LowLatency (L2ALL) [15] that may be deployed [16]. The available encoding and packaging tools include Apple's HLS reference tools [17], FFmpeg [18], node-gpac-dash [19], and others.

In our prior study [20, 21], we evaluated some of these players when they introduced the support for LL-HLS and LL-DASH. In this work, we extend this study to include the latest versions of all players and the latest available adaptation algorithms [14-16]. Notably, in this work, we also study the performance of the THEO player [12]. THEO is currently the only player supporting LL-HLS, LL-DASH, and High-Efficiency Streaming Protocol (HESP) – a new protocol submitted for consideration in IETF [22]. This protocol is claimed to have some additional delay advantages over LL-HLS and LL-DASH [23], although the principles of operation are very similar.

To evaluate all players, we use a common evaluation framework, with identical content, the same encoding ladder and codec-specific constraints to encode it, identical network conditions simulated in all cases, and consistent set of metrics collected and processed. This is done to ensure an accurate and fair comparison. In Section 2, we will describe the details of 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 Video encoding and 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 FFmpeg [18] and Apple's HLS reference tools [17]. To generate LL-DASH

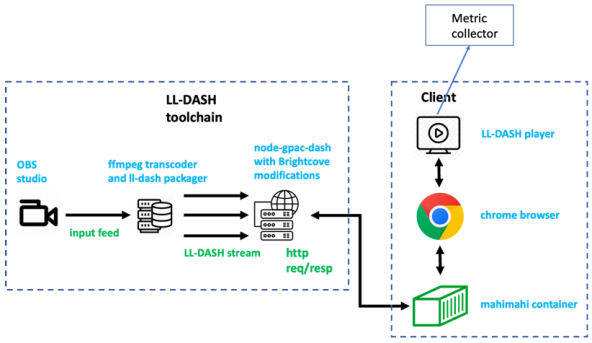

Figure 1: Toolchain used for testing LL-DASH players

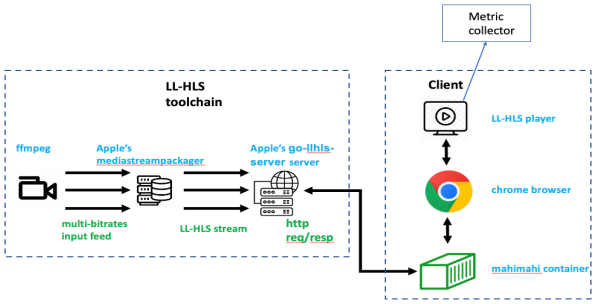

Figure 2: Toolchain used for testing LL-HLS players

streams, we used OBS studio [24], FFmpeg [18], and a modified version of node-gpac-dash [19]. Additional details about our setups can be found in [25,26]. The LL-HLS stream was served dynamically by Apple's go-llhls-server [17]. The LL-DASH stream was served dynamically by node-gpac-dash [19].

As shown in Figures 1 and 2, the input feed(s) were sent to the low-latency packagers (mediastreamsegmenter [17] for LL-HLS, and FFmpeg for LL-DASH). The outputs of low-latency packagers are the chunked video segments and the manifest files describing how the players can consume the streams in low-latency mode. Next, the output stream files are served by the low-latency video servers (go-llhls-server [17] for LL-HLS, node-gpac-dash [19] for LL-DASH) to players in a chunked manner.

2.2 Content and video encoding parameters

As a test video sequence, we used a 1080p version of the Big Buck Bunny [27] video. This sequence was looped to generate a continuous live source stream in MPEG-TS format carried over UDP transport. Four live transcoded variant streams (from SD up to full HD resolution) were created, with encoding parameters listed in Table 1. For all renditions, the video frame rate was 25fps, the segment length was set to 4 seconds, the low-latency chunk length was set to 1 second, and the segment container format was fragmented mp4.

To minimize fluctuations in the encoding bitrates, constant bitrate (CBR) encoding mode has been utilized. H.264 encoder operating in the Main profile has been used. Lookahead processing was disabled to minimize the encoding delays. The segment and fragment durations were set to 4secs and 1sec, respectively,

Table 1: Encoding profile parameters

Rendition	Resolution [pixels]	Codec / profile	Bitrate [kbps]
Low	768 x 432	H.264 / Main	949
Mid	1024 x 576	H.264 / Main	1854
High	1600 x 900	H.264 / Main	3624
Top	1920 x 1080	H.264 / Main	5166

Table 2. Streaming players / configurations used

Player (protocol)	Player / SDK / access date	Adap. method
DASH.js default	DASH.js, Oct 2022, [9]	default
DASH.js LolP	DASH.js, Oct 2022, [9]	LoL+ [14]
DASH.js L2all	DASH.js, Oct 2022, [9]	L2All [15]
Shaka player (dash)	Shaka player, Jan 2023, [11]	default
THEO player (dash)	THEO player, Jan 2023 [12]	default
Hls.js default 2020	HLS.js, Oct 2020, [8]	default
Hls.js LolP 2020	HLS.js, Oct 2020, [8]	LoL+ [14, 16]
Hls.js L2all 2020	HLS.js, Oct 2020, [8]	L2All [15, 16]
Hls.js default 2023	HLS.js, Jan 2023, [8]	default
Shaka player (hls)	Shaka player, Jan 2023, [11]	default

matching the default values used in Apple's streaming tools for LL-HLS [17]. The same encoding parameters have been used to generate LL-DASH and LL-HLS streams.

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

2.3 Streaming players

Overall, we have evaluated 10 different configurations of low-latency streaming players, with parameters listed in Table 2.

For LL-DASH, we used DASH.js [9], Shaka [11], and THEO [12] players. With DASH.js we used three different low-latency ABR algorithms: DASH.js default [9], LoL+ algorithm [14], and L2All algorithm [15]. For DASH.js and Shaka players, we used versions of the player SDKs downloaded in Oct 2022. For THEO player, we used their public testing player [12].

For LL-HLS, we used Shaka player [11], the latest HLS.js [8] version available in Jan 2023, and a 2020 version of HLS.js with three different low-latency ABR algorithms (Hls.js 2020 default, Lol+ and L2all) which were re-implemented and integrated in HLS.js by the University of Klagenfurt [16]. For all players, except THEO player, we have implemented simple test applications. The THEO player was already pre-built and hosted by THEO technologies [12].

2.4 Performance metric collection and processing

We have instrumented the player applications to periodically collect metrics indicative of live streaming latency, playback speed, and re-buffering events. We collect all the metrics at a rate of one second. So, for a 600 seconds streaming session, we had 600 data points for all the metrics.

Table 3. Performance metrics collected

Metrics [units]	Dimensions
Streaming bitrate [kbps]	Efficiency, QoE
Video resolution [height]	QoE
Live latency [secs]	Latency, QoE
Variation of playback speed [x 1 speed]	QoE
The number of bitrate switches	QoE
The number of rebuffering events	QoE
Rebuffering ratio [%]	QoE
Downloaded media data [Mbytes] and media objects [chunks or segments]	Efficiency

Table 4: Network bandwidth statistics

Statistics	T-Mobile LTE	Verizon LTE
Avg. Bandwidth [kbps]	12258	10565
St. dev of bitrate [kbps]	9314	8619
Min. bandwidth [kbps]	12	12
Max. bandwidth [kbps]	59460	42804

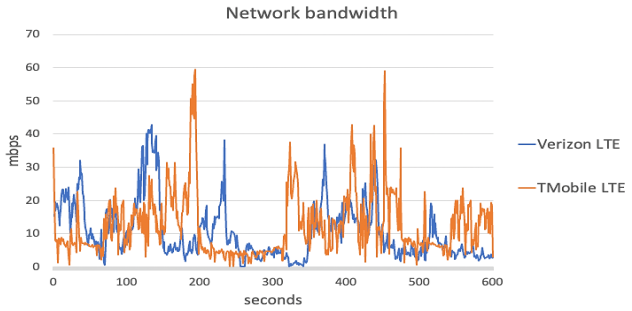

Figure 3. Network traces used for the experiments.

All the player SDK(s) provide API methods for monitoring live latency. We used these methods to collect latency data in the player applications. The exception is the THEO player, hosted on a proprietary domain. However, since the hosted player calculates latency internally in the player application, we used Chrome's live expression feature to extract its latency reports and print them to the developer console every 250ms. Playback speed data was collected using the built-in API method in the HTML5 video element [28]. HTML5 video element also provides methods for monitoring player buffer length, so we use player buffer length to detect playback stalls and re-buffering events. Specifically, stream re-buffering events are detected by tracking the player buffer length every second. When the player's buffer length drops below 0.3 seconds at time **t0**, the player reports one re-buffering event (playback stall). After time **t0**, we count the number of seconds when the player's buffer length remains below 0.3 seconds until it buffers more than 0.3 seconds at time **t1**. This indicates one re-buffering event of **T = t1 - t0** seconds. For the entire 600 seconds session, we count the total number of re-buffering events and the total number of seconds when the player buffers less than 0.3 seconds.

Other metrics, such as stream bitrate, video resolution, and media data downloaded, were derived from the streaming servers' access logs. Specifically, we instrumented both LL-HLS and LL-DASH servers to report (every second) the most recent rendition that the player requested. The processing of all collected metrics was done offline. Table 3 provides the complete list of metrics collected in our test system. The re-buffering ratio is the buffering time to the play-time ratio observed during the session.

2.5 Network emulation

We used the Mahimahi network emulator to emulate various network conditions at the network interface level [29]. Mahimahi is essentially a Linux container that can run an application inside. 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. When we run the test players inside Mahimahi, the player download speed is limited by the capacity of the virtual interface. Unlike 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. Such a methodology allows a fair comparison of different players.

We have evaluated the test players using two 4G-LTE network traces from T-Mobile and Verizon networks, respectively [29]. We provide the visualizations of these traces and the related statistics in Figure 3 and Table 4.

3 THE RESULTS

3.1 The results for T-Mobile LTE network

The performance metrics collected using T-Mobile LTE network are shown in Table 5. Figures 4 and 5 illustrate the variations of player-selected bitrates, Figures 6 and 7 illustrate the changes in playback latency, and Figures 8 and 9 illustrate changes in playback speeds as observed during test sessions.

As can be observed, in terms of network bandwidth consumption the highest numbers are achieved by THEO player. It also has shown the best buffering performance among LL-DASH players, and had 0 playback speed variation. However, in terms of latency, THEO player was almost 2x behind DASH.js [9], operating with a 6.16sec delay vs. 3.06 sec for the latter. Among HLS players, we note that HLS.js, using its new (2023) default adaptation algorithm, has achieved the highest average bitrate. But it also operated at a longer latency of 8.93 sec. HLS.js showed the best latency among HLS-based players with the LOL+ algorithm [14,16]. HLS.js with L2ALL [14,16] was the second best in this dimension. We also note that the Shaka player has shown reasonably solid and consistent behavior with LL-HLS and LL-DASH streams. Shaka and THEO players were also the only players maintaining the constant playback speed.

Table 5: Performance statistics – T-Mobile LTE network

Player/Algorithm	Avg. bitrate [kbps]	Avg. height [pixels]	Avg. latency [secs]	Latency var. [secs]	Speed var. [%]	Number of switches	Buffer events	Buffer ratio [%]	MBs loaded	Objects loaded
DASH.js default	2770	726	**3.06**	0.21	10.4	93	38	7.99	352.2	256
DASH.js LolP	3496	853	5.65	4.59	22.7	70	53	21.96	369.4	210
DASH.js L2all	3699	908	4.14	3.18	19.9	5	19	7.99	368	147
Shaka player (dash)	3818	916	4.92	2.06	**0**	16	5	4.66	360.3	155
THEO player (dash)	**4594**	**993**	**6.16**	0.01	**0**	27	**0**	0	**418.7**	152
HLS.js default 2020	1763	562	10.08	10.91	8.1	26	2	9.8	130.7	**589**
HLS.js LolP 2020	1756	560	**5.97**	0.2	6.1	24	**0**	0	148.1	**688**
HLS.js L2all 2020	1752	560	6	0.23	5.9	34	**0**	0	133.1	**686**
HLS.js default 2023	**3971**	895	8.93	1.13	**0**	8	**0**	0	**360.8**	**613**
Shaka player (hls)	3955	**908**	7.18	2.23	**0**	14	7	3.8	230	**475**

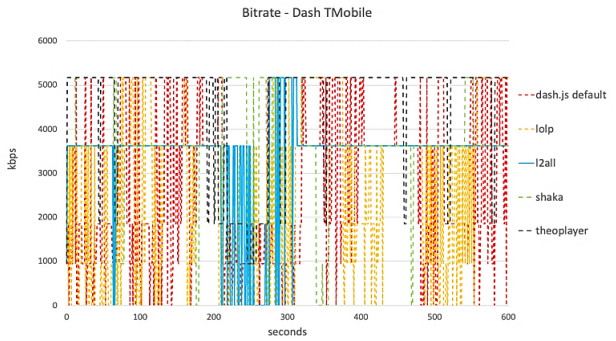

Figure 4. Stream bitrate – DASH – T-Mobile LTE

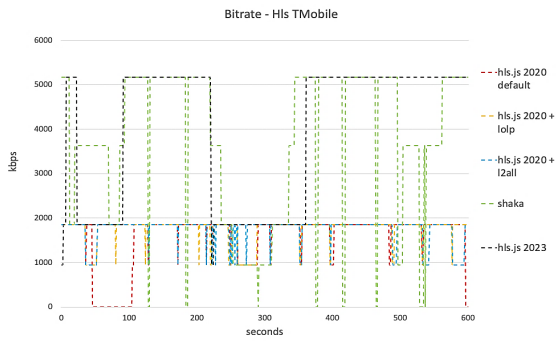

Figure 5. Stream bitrate – HLS – T-Mobile LTE

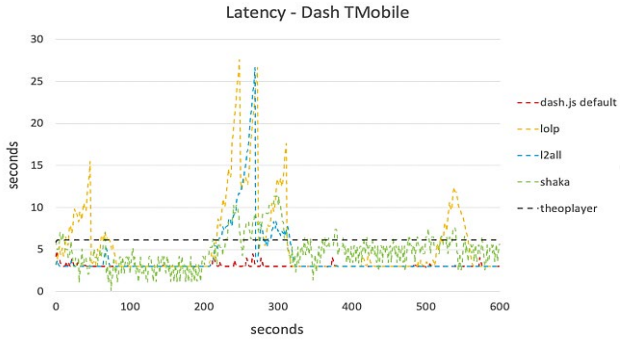

Figure 6. Live latency – DASH – T-Mobile LTE

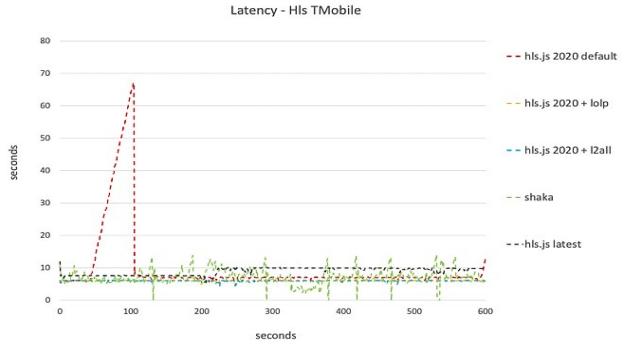

Figure 7. Live latency – HLS – T-Mobile LTE.

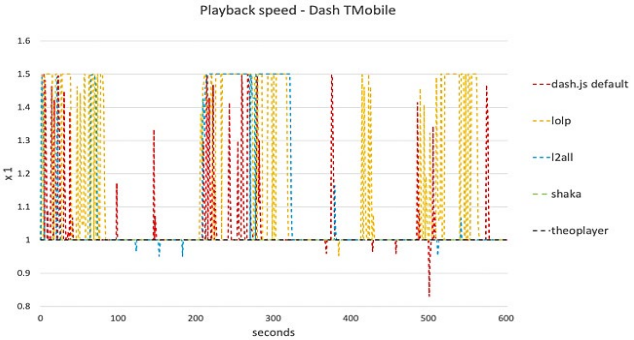

Figure 8. Playback speed – DASH – T-Mobile LTE

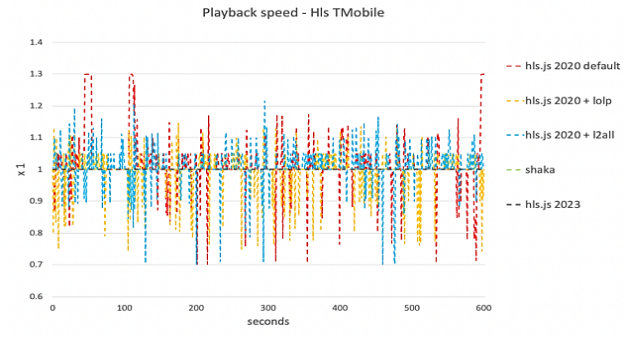

Figure 9. Playback speed – HLS – T-Mobile LTE.

3.2 The results for the Verizon LTE network

The performance metrics collected using the T-Mobile LTE network are shown in Table 5. Figures 10 and 11 illustrate the variations of player-selected bitrates, Figures 12 and 13 illustrate the changes in playback latency, and Figures 14 and 15 show changes in playback speeds as observed during test sessions

Table 6: Performance statistics – Verizon LTE network

Player/Algorithm	Avg. bitrate [kbps]	Avg. height [pixels]	Avg. latency [secs]	Latency var. [secs]	Speed var. [%]	Number of switches	Buffer events	Buffer ratio [%]	MBs loaded	Objects loaded
DASH.js default	2131	627	3.79	3.16	14.9	91	23	7.99	260	207
DASH.js LolP	3368	829	7.29	6.8	23.9	106	51	21.96	351	221
DASH.js L2all	3672	905	6.47	5.36	23.3	7	7	7.99	338	135
Shaka player (dash)	3653	886	6.96	7.08	0	26	5	4.66	329	148
THEO player (dash)	**4153**	**909**	**18.19**	10.08	0	33	**2**	0	383	153
HLS.js default 2020	2085	610	11.66	10.25	11.4	28	3	9.8	140	**606**
HLS.js LolP 2020	1890	580	7.86	2.63	7.0	24	**2**	0	146	**569**
HLS.js L2all 2020	1803	567	8.84	4.43	10.7	26	**2**	0	145	**547**
HLS.js default 2023	3541	822	16.78	9.13	0	9	5	0	280	**598**
Shaka player (hls)	**3669**	860	7.98	2.41	0	22	9	3.8	260	**570**

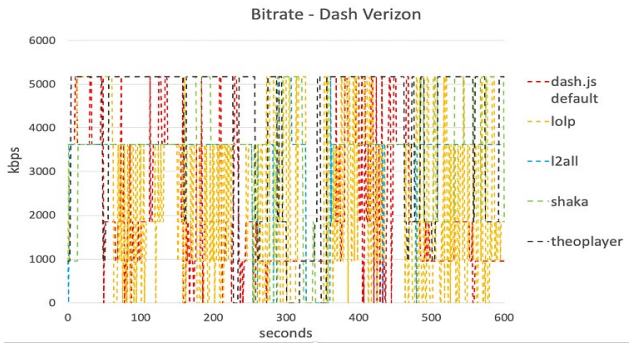

Figure 10. Stream bitrate – DASH – Verizon LTE

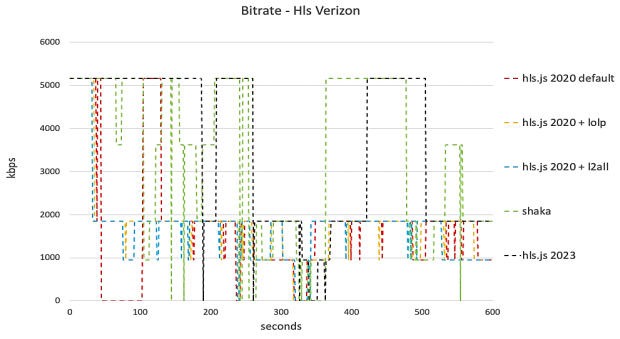

Figure 11. Stream bitrate – HLS – Verizon LTE

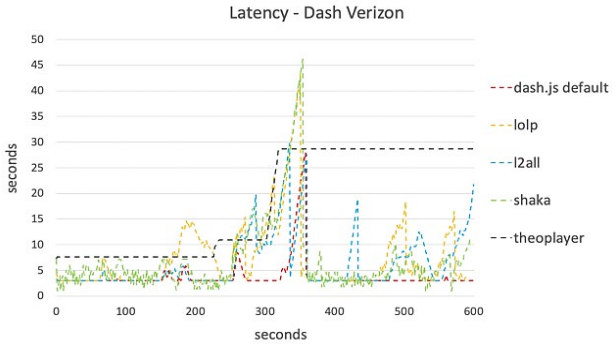

Figure 12. Live latency – DASH –Verizon LTE

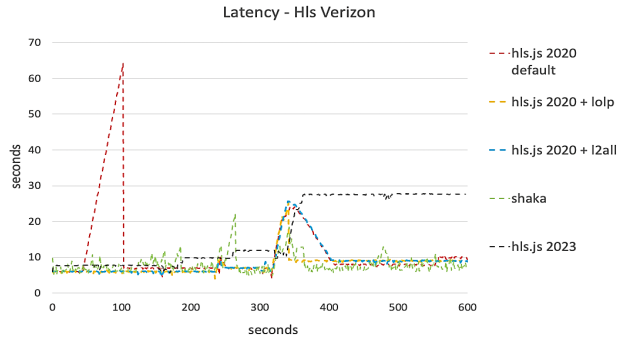

Figure 13. Live latency – HLS – Verizon LTE.

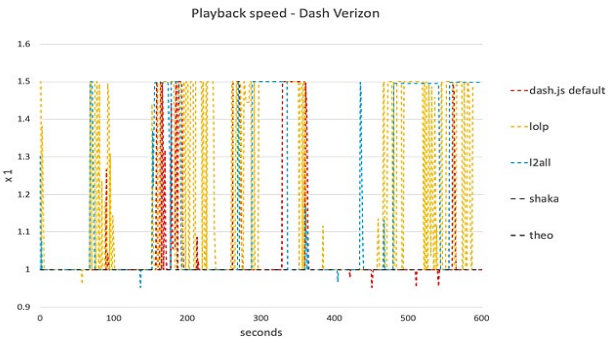

Figure 14. Playback speed – DASH – Verizon LTE

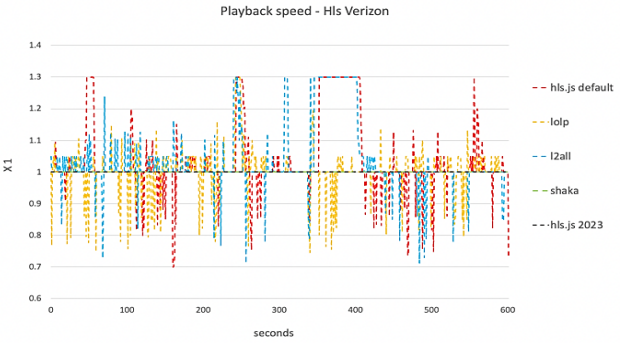

Figure 15. Playback speed – HLS – Verizon LTE.

Compared to the T-Mobile network study, we note that the Verizon network is considerably more challenging, resulting in higher buffering and longer delay statistics. We also notice that under such difficult conditions, some players switch operations from low-

delay to standard (20-30sec delay) regime. We observe this behavior with the THEO player for LL-DASH and with the Shaka player for LL-HLS. But many of our earlier observations remain the same. E.g., THEO player still manages to pull most data and buffer less, DASH.js operates with the shortest delay, Shaka delivers similar performance for both LL-HLS and LL-DASH, etc. In both cases, we also note that LL-HLS players produce many more object load requests than LL-DASH players.

6 Conclusions

This study evaluated ten recent implementations of LL-HLS and LL-DASH streaming players, operating under identical network conditions and with consistently encoded and packaged content.

Our experiments confirmed that LL-HLS and LL-DASH could deliver significantly lower latencies than traditional HLS and DASH streaming systems. With LL-DASH players, we have observed that streaming is possible with delays as short as 3-4 sec, while with LL-HLS streams, we noted that delays of 6-8 sec could be attainable.

However, we have also noticed that achieving such short delays typically comes with compromises in QOE, efficiency factors, or both. The parameters that are usually affected are:

- stream switching and buffering rates,
- the ability of players to select high renditions,
- the ability of players to sustain constant playback speed,
- the ability of players to sustain low delay,
- the rate of requests sent to CDNs and origin servers,
- the amount of data transmitted, etc.

The performed comparison shows how different existing player implementations achieve different tradeoffs along all these dimensions. And while we note considerable progress in improving such player algorithms since the introduction of LL-HLS and LL-DASH around 2016-2019, we believe that additional performance improvements are still possible and much needed for the success of these technologies in practice.

REFERENCES

[1] IETF RFC 8216, "HTTP Live Streaming", https://tools.ietf.org/html/rfc8216, 2017.
[2] ISO/IEC 23009-1:2012, "Information technology - Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats," February 2012.
[3] IETF RFC 8216, HTTP Live Streaming, 2nd Edition, https://tools.ietf.org/html/draft-pantos-hls-rfc8216bis-08, 2019.
[4] Apple, Enabling Low-Latency HLS, https://developer.apple.com/documentation/http_live_streaming/enabling_low-latency_hls
[5] ETSI technical specification, "MPEG-DASH Profile for Transport of ISO-BMFF Based DVB Services over IP Based Networks", May 2015, https://www.etsi.org/deliver/etsi_ts/103200_103299/103285/01.01.01_60/ts_103285v010101p.pdf
[6] DASH Industry Forum, "Low-Latency Modes for DASH", https://dashif.org/docs/CR-Low-Latency-Live-r8.pdf
[7] AVFoundation, https://developer.apple.com/av-foundation/
[8] Hls.js player, https://github.com/video-dev/hls.js/
[9] DASH.js player, https://github.com/Dash-Industry-Forum/DASH.js
[10] Video.js player https://videojs.com/
[11] Shaka player, https://github.com/google/shaka-player
[12] THEO player, https://www.THEOplayer.com/test-your-stream-hls-dash-hesp
[13] 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," ACM Multimedia Systems Conference, Online, June 8-11, 2020.
[14] A. Bentaleb, M. N. Akcay, M. Lim, A. C. Begen, and R. Zimmermann, "Catching the Moment with LoL+ in Twitch-Like Low-Latency Live Streaming Platforms," IEEE Trans. Multimedia, vol. 24, pp. 2300-2314, 2022.
[15] T. Karagkioules, R. Mekuria, D. Griffioen, A. Wagenaar, "Online Learning for Low-Latency Adaptive Streaming," ACM Multimedia Systems Conference, Online, June 8-11, 2020.
[16] A. Bentaleb, Z. Zhan, F. Tashtarian, M. Lim, S. Harous, C. Timmerer, H. Hellwagner, and R. Zimmermann, "Low Latency Live Streaming Implementation in DASH and HLS," ACM Int. Conference Multimedia, Lisbon, Portugal, October 10-14, 2022
[17] HLS tools, https://developer.apple.com/documentation/http_live_streaming/about_apple_s_http_live_streaming_tools
[18] FFmpeg, https://www.ffmpeg.org/
[19] DASH Low Latency Server, https://github.com/maxutility2011/node-gpac-dash
[20] B. Zhang, T. Teixeira, Y. Reznik, "Performance of Low-Latency HTTP-based Streaming Players," Proc. ACM Multimedia Systems Conference (MMSys'21), Istanbul, Turkey, Sept. 28 - Oct. 1, 2021.
[21] B. Zhang, T. Teixeira, and Y. Reznik, "Performance of Low-Latency DASH and HLS Streaming in Mobile Networks," SMPTE Motion Imaging Journal, vol. 131, no. 7, 2022.
[22] HESP, IETF draft proposal: https://datatracker.ietf.org/doc/draft-THEO-hesp/
[23] HESP Alliance web page: https://www.hespalliance.org/
[24] Open Broadcast Software, https://obsproject.com/
[25] B. Zhang, Setting up your Own Low-Latency HLS Server to Stream from any Source Inputs, https://bozhang-26963.medium.com/setting-up-your-lowlatency-hls-server-to-stream-from-any-source-inputs-de1e757a6688
[26] B. Zhang, Low-Latency DASH Streaming Using Open-Source Tools, https://bozhang-26963.medium.com/low-latency-dash-streaming-using-opensource-tools-f93142ece69d
[27] Blender Foundation, Big Buck Bunny video, https://download.blender.org/
[28] W3C, HTML5 video element, https://www.w3.org/TR/2011/WD-html5-20110113/video.html
[29] 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.