Performance evaluation and testbed for delivering SRT live content using DASH Low Latency streaming systems

Román Belda, Pau Arce, Juan Carlos Guerri†
Institute of Telecommunications and Multimedia Applications
Universitat Politècnica de València, Spain
robelor@iteam.upv.es, paarvi@iteam.upv.es, jcguerri@iteam.upv.es

Ismael de Fez
Universidad Internacional de Valencia, Spain
idefez@universidadviu.com

ABSTRACT

The work presented in this paper focuses on the implementation of a testbed for the evaluation of content distribution systems using LL-DASH (*Low Latency DASH - Dynamic Adaptive Streaming over HTTP-*) and devices that provide real-time sources or servers using real-time protocols, such as RTSP (*Real Time Streaming Protocol*) or SRT (*Secure Reliable Transport*). These protocols are widely used by IP (*Internet Protocol*) cameras or by production and transmission systems, such as OBS Studio or vMix. The objective is to show the necessary processes in detail (so they can be reproduced in future works related to low latency services) and to minimize the end-to-end delay, obtaining values in the order of 2 seconds or less. The implementation has been done using FFmpeg software, players like Dash.js or Shaka-Player and implementing a Python web server with LL-DASH support to optimize the transmission delay.

CCS CONCEPTS

• Network • Network performance evaluation • Network performance analysis

KEYWORDS

DASH Low Latency, Live streaming, Testbed, Multimedia Software Open Source

1 Introduction

The DASH (*Dynamic Adaptive Streaming over HTTP*) standard has become the most popular and widely used protocol in VoD (*Video on Demand*) streaming platforms [1] for several reasons: 1) scalability, since the content server can be a standard web server and the distribution is done over HTTP, being able to take full advantage of the benefits of CDNs (*Content Delivery Network*); 2) universality, since web technology can be played from browsers; and 3) compatibility with network equipment, since the widespread use of HTTP (*Hypertext Transfer Protocol*) in Internet browsing makes multimedia traffic transparent to intermediate routers.

In this context, there are more and more streaming platforms (Youtube Live, Twitch, …) that offer live content as well, such as in the e-learning sector, online conferences, or the gaming world, along with other use cases that require some kind of interactivity. For these cases, low latency solutions are needed. There are low-latency adaptive HTTP streaming proposals such as Low-Latency HLS (*HTTP Live Streaming*) [2] and Low-Latency DASH [1][3], which try to reduce the delay as much as possible from video packetization to playback. However, there are also other added delays, such as encoding delay or acquisition delay if the ingest is done through protocols such as RTMP (*Real Time Messaging Protocol*), SRT or RTSP, widely used in commercial IP cameras and in real-time content distribution systems (Teams, Zoom, IPTV, OBS, vMix, …). Moreover, in DASH, it is the client who requests video segments while the technique used in LL-DASH is based on the pushing of content while it is being generated, so the server cannot be a standard web server [4]. In the literature and online repositories, proposals for server and encoder implementations can be found [5], although there are no IP cameras that support the DASH standard at a commercial level, let alone LL-DASH, making the study and development in this field interesting.

In order to delve into the many factors that take part in the end-to-end delay, this paper presents a testbed for LL-DASH testing as well as an evaluation of the latency for different video origins and manifest parameters.

The rest of the paper is organized as follows: Section 2 presents the state of the art of the available open source tools for developing LL-DASH streaming; Section 3 presents the main

†Corresponding author

factors and protocols related to latency; Section 4 describes the communication architecture; Section 5 introduces the proposed testbed, while Section 6 presents the evaluation carried out using the aforementioned testbed; finally, Section 7 summarizes the main conclusions as well as the future work.

2 State of the art

LL-DASH technology is based on sending the segments in a different way than the DASH technology used in VoD systems. This is called "chunked transfer encoding". However, the disadvantage of using this type of transfer, which allows segments to be generated and sent simultaneously, is that traditional adaptation algorithms do not correctly calculate the available bandwidth. Therefore, in [7] authors present the ACTE (*Adaptive Streaming with Chunked Transfer Encoding*) algorithm (using a sliding window and an online linear adaptive filter) as an alternative to overcome this drawback. And in [8], also published by the same authors, the Automated Model for Prediction (AMP) algorithm is presented. Along the same lines, the authors of [9] present a new algorithm called Llama (*Low Latency Adaptive Media Algorithm*). This algorithm uses two independent measures of throughput on different time scales. The algorithm is compared with the typical algorithms used by DASH, showing its advantages using the measurements made with the P.1203 standard.

In [10] the authors review the performance of protocols such as DASH and real-time protocols (RTP/RTCP, SRT). In this article authors implement an adaptive bitrate algorithm for the SRT protocol. In this way, they manage to include one of the advantages of DASH technology, which is the adaptation of the bitrate depending on the state of congestion. The same authors focus the study in [11] considering the need to use the "chunked transfer encoding" transmission proposed in CMAF (*Common Media Application Format*) to reduce the delay in 5G network applications. On the other hand, the authors of [12] also focus on the evaluation of the delay improvement introduced by the use of HTTP chunked delivery. To do so, they carry out a study on a real implementation and through simulations.

From the point of view of the most widely used DASH players, Dash.js is one of them. It is the result of an initiative of the DASH Industry Forum. In [13] it is presented an implementation of an algorithm called SARA (*Segment Aware Rate Adaptation*), in which VBR (*Variable Bitrate*) coding is considered and therefore with a different size of the generated segments.

Regarding the different low latency solutions, in [14] an in-depth comparison is made between the low latency solution proposed by Apple (HLS LL) in 2019, the standard LL-DASH solution of 2019 and Low-latency HTTP Live Streaming (LHLS) that was first introduced by Twitter's Periscope in 2018 and then improved by Twitch in 2019, also describing the differences between the different solutions.

From the point of view of end-to-end delay, in [15] the authors focus on providing an end-to-end solution through what they call HxL3 architecture (*HTTP/x-based Low-Latency Live streaming architecture*). This solution is agnostic to the codecs (H.264, H.265, ...), application protocols (HTTP/1.1 or HTTP/2.0), streaming format (DASH, HLS), transport protocol (TCP -Transport Control Protocol- or UDP -User Datagram Protocol-) and CDNs. And in [16], the authors focus primarily on the study of the overhead introduced by streaming content using LL-DASH. The use of transmission using "chunked transfer encoding" introduces an increase in headers and therefore overhead in the network.

Unlike other papers that also present results based on their own test-beds [12][15][16], this work presents an open-source web service that supports chunked transmissions as well as FFmpeg scripts to quickly deploy a LL-DASH evaluation scenario. In addition, this work is an extension of previous work carried out by the authors regarding the DASH protocol, specifically on the evaluation of QoE (*Quality of Experience*) in DASH [17] and the impact of DASH streaming on Energy Efficient Ethernet [18].

3 Low Latency: factors and protocols

According to a report by Bitmovin [19], on the main concerns of companies in the video streaming sector, the problem of latency comes first (41%), second is the controlling cost (e.g., bandwidth, storage, …) (33%), and third is device compatibility (32%).

Low latency definition depends to a large extent on the application in which the distribution system is framed. In fact, depending on the application (video on demand, live streaming, videoconferencing, etc.) the latency requirements (in addition to bandwidth requirements, loss tolerance, etc.) are different. To give an example at each end of the multimedia applications, it can be seen how video conferencing or security applications need much lower latency than VoD applications (such as Netflix, HBO, Youtube...) since video conferencing is an interactive and real-time application with very demanding delay requirements (around 150-200 ms).

Generally, real-time, ultra-low latency and low latency solutions involve using protocols such as WebRTC, RTP/RTSP or SRT (protocols used by applications such as Teams, Zoom, Skype, or IPTV services). While at the other extreme we can find HLS and DASH protocols that offer latencies in the order of 20-30 seconds typically (used by applications such as Netflix, Youtube, HBO, etc.). However, we can see how in the 1-5 second range there is the possibility of using all protocols. At this point it is important to note that while WebRTC or RTP/RTSP protocols are implicitly designed to offer these low latency performances, HLS or DASH protocols need new functionalities both on the encoding and packaging side.

Some aspects that contribute to the final latency observed in the player are discussed below.

First of which is the encoder. Currently there are different encoders widely used by devices (cell phones, cameras, tablets, etc.) such as H.264/AVC (*Advanced Video Coding*) and H.265/HEVC (*High Efficiency Video Coding*), and recently the specifications of other encoders, such as H.266/VVC (Versatile Video Coding), have been published. From the point of view of the R-D (rate-distortion) curve, it is possible to make an evaluation and check how each encoder fulfills its announced improvements. For example, Figure 1 shows the result of measuring the VMAF (*Video MultiMethod Assessment Fusion*) parameter with respect to the bitrate for Blender's Agent327 sequence (www.blender.org), with a resolution of 1280x720 pixels and encoded at 24 fps. It can be seen, for example, that for a VMAF value equal to 90, we need about 190 kbps with H.266, 420 kbps with H.265 and 790 kbps with H.264. However, if we consider the encoding delay, we obtain a difference in the encoding time of 1180 times in H.266 and 3 times in H.265, with respect to the encoding time of H.264 (currently making unfeasible the use of H.266 for the transmission of events in real time).

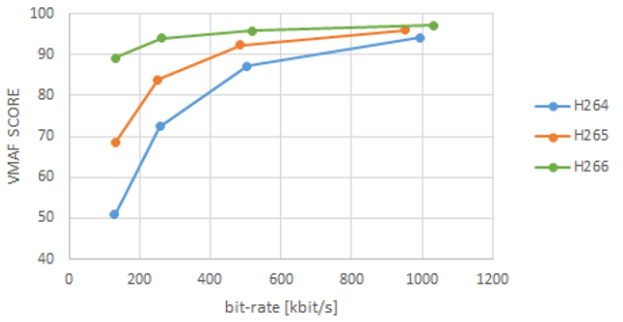

Figure 1: R-D curve for different video encoders

Moreover, the GoP (*Group of Pictures*) represents the pattern of appearance of the frames (I, P, B) in the encoded video. There are multiple possible GoP configurations: periodic/non-periodic; with B frames/without B frames; open/closed; large size/small size; etc. Regarding latency, whether to use B-frames and the size of the GoP are of great importance in the GoP configuration. And regarding GoP size and latency, this feature has less influence on RTP-based protocols (WebTRTC) than on HLS or DASH protocols. The reason is that protocols such as RTP transmit frame by frame, while HLS or DASH protocols transmit segments (groups of frames) that usually consist of a whole number of GoPs.

DASH and HLS are based on the transmission of segments using the HTTP protocol and adaptive algorithms. The fact of using segments introduces an intrinsic delay in the system, since it is necessary to wait for the generation of the segment in order to be able to transmit it. Typical values for the duration of such segments are 2 s, 4 s, 6 s or 10 s. Without considering the fact of having to use a buffer in the receiver, the segmentation process already introduces an unacceptable delay for many interactive or real-time applications. Figure 2 shows schematically the structure of a 6 s segment in the fMP4 (fragmented MP4) format. When using fMP4 file format, the encoded file is divided into segments that can be downloaded and played back. However, to start playback, it is necessary for the player to download the complete segment (mdat).

Figure 2. Traditional fMP4 segment (e.g. 6 seconds)

To reduce playback delay, a new container format called CMAF (*Common Media Application Format*) has been specified, which allows the segment to be divided into chunks and to start playback while new chunks are still being generated. Figure 3 shows how a segment has been split into one chunk per frame, ideal for real-time applications. However, there is a trade-off between the latency reduction achieved and other aspects such as the probability of interruptions due to congestion or the increase in requests for access to the chunks.

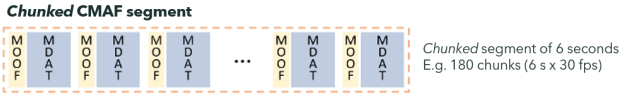

Figure 3. CMAF chunked segment (e.g. 180 chunks)

Finally, to minimize the effect of congestion, all players use buffers to store a certain number of frames or segments before starting playback (Figure 4). Again, there is a trade-off between buffer size and the probability of service interruptions. Typical values are in the range of 30 s buffer. However, depending on the applications (and specifically for real time) these values must be reduced to obtain a valid service.

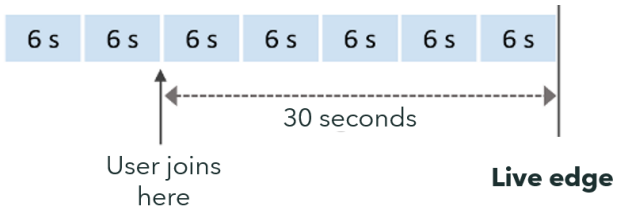

Figure 4. Segments in player buffer (e.g. 30 seconds)

Once the factors that affect latency have been analyzed, the protocols used for video transmission must be considered, as well as the protocols used for distribution to the end customers. In fact, each option will have different answers regarding latency, scalability, etc. that should be evaluated.

4 Real-time multimedia content distribution scenarios

The starting point of the presented work is based on the advantages of real time protocols (SRT, RTSP, ...) for live content ingest, and the advantages of using DASH (and specifically LL-DASH) as a technology for the distribution of multimedia content to end users. These advantages are well known and can be summarized as follows:

- **Reduced management complexity**: By allowing HTTP video transmission using TCP ports 80 and 443 and eliminating the need to deploy and manage a separate caching infrastructure. In addition, in most corporate networks, some level of restriction at the protocol and port level is used to minimize the likelihood of attacks. However, ports 80 and 443 are almost always open for generic web traffic flow and therefore for HTTP video. On the other hand, ports using other video streaming protocols such as RTMP, SRT, RTSP, etc. are not always open, hindering or blocking these protocols.

- **Cost reduction**: Non-HTTP transmission protocols increase the cost of the infrastructure as they require specific hardware and software in the server, forming a parallel infrastructure to the network of the rest of the services. In addition, inefficient content caching can increase the amount of bandwidth required to transmit popular videos over the network. However, HTTP technology leverages the existing HTTP server network, allowing organizations to save costs that would otherwise be spent on specialized hardware and software. And as access to video content increases, HTTP caching proxies dramatically reduce bandwidth costs over accessing uncached video.

- **Improved Scalability**: The ubiquity of HTTP servers and the protocol's native support for perimeter caching make HTTP the ideal choice for streaming large-scale live events and on-demand content for very frequent access.

To summarize, we are faced with a situation in which protocols designed for real-time transmission (from cameras, transmission equipment or production software (OBS [20], vMix [21], etc.)) such as RTSP or SRT protocols, and the advantages of using LL-DASH technology for content distribution must coexist. Therefore, it is necessary to integrate both technologies to take advantage of the benefits of both. Figure 5 depicts schematically the proposed scenario for this purpose.

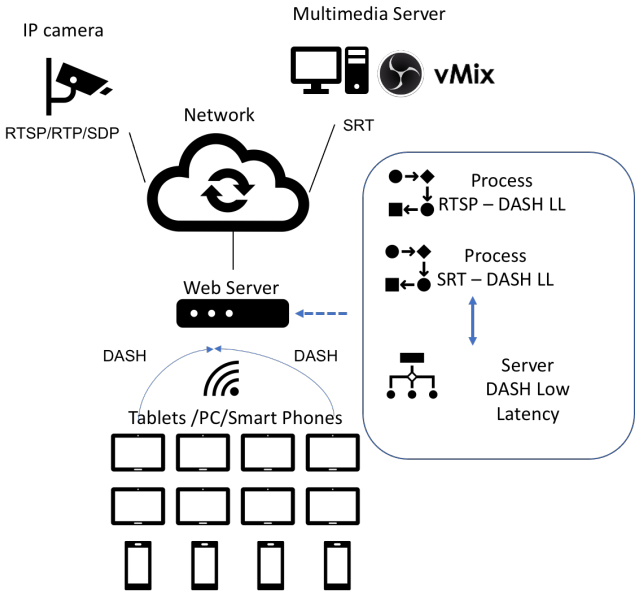

Figure 5: Scenario - Real Time Production – LL-DASH Distribution

Figure 5 shows video content sources (cameras or content servers). The output of these sources uses RTSP or SRT. The server includes the developed processes that take care of the reception of these streams and their efficient transformation into video segments complying with the CMAF format for their use by the LL-DASH technology. On the other hand, a Web server has been implemented in Python that supports the distribution of content using the LL-DASH standard by sending chunks. Finally, the devices play the contents through DASH clients (such as DASH.js, Shaka-Player, ...).

5 Low Latency DASH testbed

The use of standard HTTP servers for DASH is a key benefit of the technology but imposes a defined sequence of events where segment files must be available on the server prior to any request could be successfully handled. This sequence poses no inconvenience for VoD, where segments can be available on the servers in advance, or even on live streaming, where players can be playing some segments behind, but generates a minimum delay between the generation and the consumption of the content. Figure 6 shows how DASH clients must request, at least, a segment behind the sequence of generated segments ($Y<X$) in order to avoid HTTP errors.

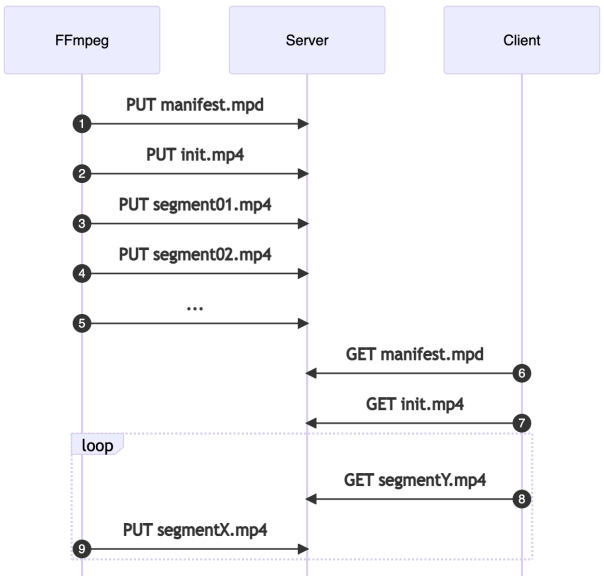

Figure 6: DASH client-server sequence

This requirement generates a minimum delay equal to the sum of the time length of the segment, the time of the transmission to the server and the backoff time to avoid HTTP errors as Figure 7 depicts.

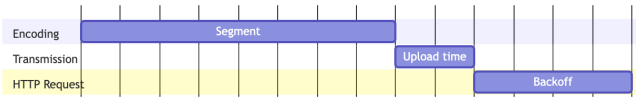

Figure 7: DASH minimum client delay

As previously stated, this minimum delay may suit some live streaming scenarios but not those which require low latency. For achieving low latency when using DASH HTTP servers can no longer be static content servers but dynamically handle segment requests.

Figure 8 shows the required behavior of HTTP servers to handle LL-DASH clients. First, manifest, and initial segment must be uploaded to the server (events 1 and 2) before it is accessible to the client (events 3 and 4). In live DASH, manifest is periodically generated and uploaded to the HTTP server, but it is omitted from the figure for clarity. Next, the client will begin to request segments based on the manifest, the target latency, and the current time. In LL-DASH those segment requests will reach the HTTP server even before the segment transmission to the server has started. In this scenario, the HTTP server must retain the HTTP request long enough to wait the incoming segment or timeout otherwise, as represented between events 5 and 6 in Figure 8.

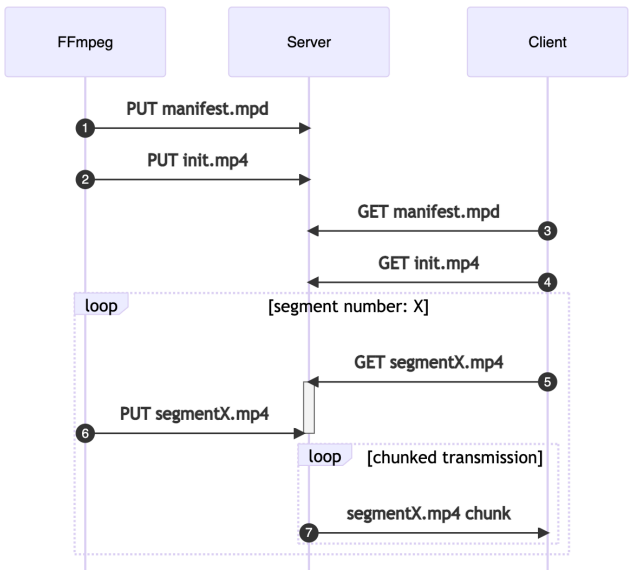

Figure 8: DASH client-server sequence

The HTTP server, when the requested segment reception begins (event 6), starts sending the content of the segment to the pending requests, as it arrives, in the form of chunked transmission.

Aiming to test the described behavior, it has been developed a Python HTTP server for LL-DASH based on FastAPI framework called Fast-ll. Figure 9 shows an excerpt of the server where a generator is created to add received chunks to the response of clients while they are being received. Fast-ll handles manifest and segments in-memory so no copy is stored on disk. The HTTP server must be fed using HTTP PUT requests, to add content, and DELETE requests to remove content (when the segments exceed the windows defined in the manifest thus no client will request it). The process of using HTTP methods to manage the content on the Fast-ll can be done by the FFmpeg tool **Error! Reference source not found.** when the appropriate set of parameters is provided.

```
async def generate_partial_segment(seg):
    aux = 0
    chunks = seg["chunks"]
    while not seg["complete"]:
        size = len(chunks)
        r = range(aux, size)
        for i in r:
            aux += 1
            yield chunks[i]
        await asyncio.sleep(5e-3)
```

Figure 9: Fast-ll generate_partial_segment function

For reference, Figure 10 includes a complete command-line that uses FFmpeg to access an SRT or RTSP stream, recodes the video stream, generates a low latency DASH stream and

uploads the different objects (manifest and segments) to the locally running Fast-ll server.

```
ffmpeg \
-fflags nobuffer \
-flags low_delay \

# SRT source
-i "srt://127.0.0.1:5000" \

# or RTSP source
-avioflags direct \
-f rtsp \
-i rtsp://root:pass@10.0.0.96:554/\
axis-media/media.amp \

-c:v libx264 \
-x264opts keyint=25:minkeyint=25:scenecut=-1 \
-tune zerolatency \
-profile:v baseline \
-preset veryfast -bf 0 -refs 3 \
-b:v 500k -bufsize 500k \
-utc_timing_url "https://time.akamai.com/?iso" \
-use_timeline 0 \
-format_options "movflags=cmaf" \
-frag_type duration \
-adaptation_sets "id=0, seg_duration=1, \
frag_duration=0.1, streams=v" \
-streaming 1 \
-ldash 1 \
-export_side_data prft \
-write_prft 1 \
-target_latency 0.5 \
-window_size 5  \-extra_window_size 10 \
-remove_at_exit 1 \
-method PUT \
-f dash \
http://localhost:8000/test/manifest.mpd
```

Figure 10: FFmpeg command line to transform RTSP into LL-DASH

Specifications for all parameters in Figure 10 can be found in [22]. Among all parameters, there are three of particular relevance: 1) `-ldash 1`: Specifies the LL-DASH mode; 2) `-target_latency 0,5`: The latency that the client will try to achieve; and 3) `-format_options "movflags=cmaf"`: The container format.

The source code of Fast-ll server and scripts used in the work can be found in the git repository https://github.com/robelor/fast-ll [22].

6 Evaluation

With the development introduced in this work it is straightforward to setup a test environment to evaluate LL-DASH tools and clients.

To demonstrate the suitability of the testbed an evaluation of the parameter "target latency" when generating LL-DASH content using FFmpeg has been performed. The evaluation is carried out using three different sources: an RTSP camera (RTSP), an FFmpeg test source (Gen) and SRT source (SRT) generated also using FFmpeg. Regarding the target latency parameter, we have evaluated values of 0.2, 0.5, 1 and 2 seconds.

FFmpeg uses the value, in seconds, of this parameter to generate the media presentation description (MPD) accordingly. For example, when 0.5 is specified as target latency parameter the resulting MPD will incorporate `"<Latency target="500"/>"` inside the `"ServiceDescription"` tag as the MPD defines the value to be in milliseconds.

Figure 11 shows a screenshot of the testbed. Labeled as 1 it is the system clock. Label 2 locates the system log of Fast-ll service and number 3 labels the Dash.js 4.4.0 player properly configured to use LL-DASH. The screenshot depicts one of the tests with the RTSP IP camera pointed to the system clock to quickly identify the overall system delay.

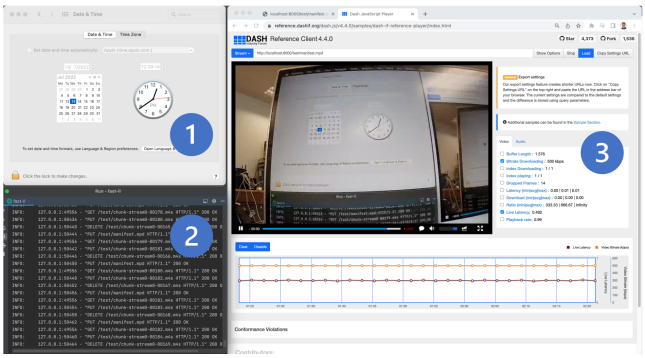

Figure 11: Screenshot of the testbed

Figure 12 shows the average delay identified by Dash.js for each protocol and target delay. Numeric values can be seen in Table 1. This measured delay refers to the time between the segmentation process and the display time of the frames. Naturally, the overall delay will include delays introduced by the video sources, transport, and segmentation.

	0.2 s	*0.5 s*	*1 s*	*2 s*
Gen	0.21078	0.41942	0.95998	1.87212
RTSP	0.35892	0.48984	0.97928	1.98286
SRT	0.20634	0.49404	1.01040	2.02898

Table 1: Latency measured by the Dash.js

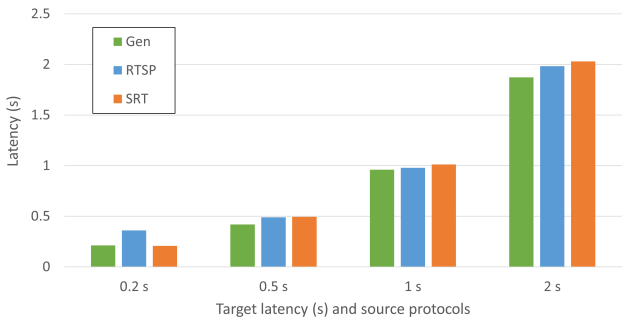

Figure 12: Latency measured by the Dash.js client for different target latency values and source protocols

7 Conclusions

Currently, integration of real-time sources (using protocols such as SRT or RTSP) for distribution over the Internet or CDNs, using LL-DASH technology, is an interesting topic from a performance analysis point of view. Although solutions have been proposed, it is difficult to find a complete system that can be replicated to compare adaptation mechanisms, measure delay, and evaluate QoE. This paper presents a system that includes all the necessary and tested processes (available at [22]) to have a LL-DASH streaming system with SRT or RTSP sources.

An interesting future work is to develop a content source agnostic measurement system to be able to measure delay in an automated way.

ACKNOWLEDGMENTS

This work is supported by the Centro para el Desarrollo Tecnológico Industrial (CDTI) from the Government of Spain under the project "Nueva plataforma a bordo basada en redes 5G y Wi-Fi 6 para medios de transporte terrestre" (CDTI IDI-20210624).

REFERENCES

[1] Abdelhak Bentaleb, Bayan Taani, Ali C. Begen, Christian Timmerer, and Roger Zimmermann. 2019. A survey on bitrate adaptation schemes for streaming media over HTTP. *IEEE Communications Surveys and Tutorials, vol. 21, no. 1, pp. 562–585*. DOI: https://doi.org/10.1109/COMST.2018.2862938

[2] Roger Pantos. 2022. HTTP Live Streaming 2nd Edition (Internet-Draft).

[3] Thomas Stockhammer, Chris Poole, Thomas Swindells, Will Law, Iraj Sodagar, Ali Begen, Thorsten Lohmar, and Kilroy Hughes. 2017. DASH-IF/DVB Report on Low-Latency Live Service with DASH.

[4] DASH Industry Forum. 2018. Guidelines for Implementation: DASH-IF Interoperability Points.

[5] Will Law. 2020. Meeting live broadcast requirements – the latest on DASH Low Latency. DASH Industry Forum.

[6] DASH Industry Forum. 2020. Low-latency Modes for DASH. Retrieved from https://dashif.org/docs/CR-Low-Latency-Live-r8.pdf.

[7] Abdelhak Bentaleb, Christian Timmerer, Ali C. Begen, and Roger Zimmermann. 2019. Bandwidth Prediction in Low-Latency Chunked Streaming. *In 29th ACM SIGMM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'19), Amherst, MA, USA. ACM, New York, NY, USA, 7 pages*. https://doi.org/10.1145/ 3304112.3325611.

[8] Abdelhak Bentaleb, Ali C. Begen, Saad Harous, and Roger Zimmermann. 2021. Data-Driven Bandwidth Prediction Models and Automated Model Selection for Low Latency. *IEEE Transactions on Multimedia*, *23*, 2588–2601. DOI: https://doi.org/10.1109/TMM.2020.3013387.

[9] Tomasz Lyko, Matthew Broadbent, Nicholas Race, Mike Nilsson, Paul Farrow, and Steve Appleby. 2020. Llama - Low Latency Adaptive Media Algorithm. *Proceedings - 2020 IEEE International Symposium on Multimedia, ISM 2020*, 113–121. https://doi.org/10.1109/ISM.2020.00027.

[10] Roberto Viola, Ángel Martín, Juan F. Mogollón, Álvaro Gabilondo, Javier Morgade, Mikel Zorrilla, Jon Montalbán, and Pablo Angueira. 2020. Adaptive rate control for live streaming using SRT protocol. *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting, BMSB*. https://doi.org/10.1109/BMSB49480.2020.9379708.

[11] Roberto Viola, Álvaro Gabilondo, Ángel Martín, Juan F. Mogollón, Mikel Zorrilla, and Jon Moltalbán. 2019. QoE-based enhancements of Chunked CMAF over low latency video streams. *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting, BMSB*. https://doi.org/10.1109/BMSB47279.2019.8971894.

[12] Ali El Essaili, Thorsten Lohmar, and Mohamed Ibrahim. 2018. Realization and Evaluation of an End-to-End Low Latency Live DASH System. *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting, BMSB*. https://doi.org/10.1109/BMSB.2018.8436922.

[13] Ali C. Begen, Mehmet N. Akcay, Abdelhak Bentaleb, and Alex Giladi. 2022. Adaptive Streaming of Content-Aware-Encoded Videos in dash.js. *SMPTE Motion Imaging Journal*, *131*(4), 30–38. DOI: https://doi.org/10.5594/JMI.2022.3160560.

[14] Kerem Durak, Mehmet N. Akcay, Yigit K. Erinc, Boran Pekel, and Ali C. Begen. 2020. Evaluating the Performance of Apple's Low-Latency HLS, *IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. https://doi.org/10.1109/MMSP48831.2020.9287117.

[15] Farzad Tashtarian, Abdelhak Bentaleb, Alireza Erfanian, Hermann Hellwagner, Christian Timmerer, and Roger Zimmermann. 2022. HxL3: Optimized Delivery Architecture for HTTP Low-Latency Live Streaming. *IEEE Transactions on Multimedia*. DOI: https://doi.org/10.1109/TMM.2022.3148587.

[16] Nassima Bouzakaria, Cyril Concolato, and Jean Le Feuvre. 2014. Overhead and performance of low latency live streaming using MPEG-DASH. *IISA 2014 - 5th International Conference on Information, Intelligence, Systems and Applications*, 92–97. https://doi.org/10.1109/IISA.2014.6878732.

[17] Paola Guzmán, Pau Arce, and Juan Carlos Guerri Cebollada. 2019. Automatic QoE Evaluation of DASH Streaming using ITU-T Standard P.1203 and Google Puppeteer. *In Proceedings of the 16th ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks (PE-WASUN '19). Association for Computing Machinery, New York, NY, USA, 79–86*. https://doi.org/10.1145/3345860.3361519.

[18] Tito R. Vargas, Juan Carlos Guerri, and Pau Arce. 2021. Study on the Impact of DASH Streaming Services using Energy Efficient Ethernet. *In Proceedings of the 18th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks (PE-WASUN '21). Association for Computing Machinery, New York, NY, USA, 89–94*. https://doi.org/10.1145/3479240.3488527.

[19] Bitmovin Video Developer Report. 2021. Retrived from https://bitmovin.com.

[20] OBS studio. 2022. Retrieved from https://obsproject.com.

[21] vMix. 2022. Retrieved from https://www.vmix.com.

[22] Jean-Baptiste Kempf. 2020. Implementing DASH low latency in FFmpeg. DASH Industry Forum.

[23] Git repository – robelor. 2022. Fast-II. Retrieved from https://github.com/robelor/fast-ll.