UNIVERSAL CMAF CONTAINER FOR EFFICIENT CROSS-FORMAT LOW LATENCY DELIVERY

W. Law[1], E. Toullec[2] and Mickael Raulet[3]

[1]Akamai Technologies, USA and [2,3]ATEME, France

ABSTRACT

Current implementations of low latency adaptive streaming over http utilize a diversity of presentation formats, containers, origins and caching behaviours. For efficient delivery, a common origin and a single set of cacheable media objects provides the highest performance and scalability across a delivery system. This paper introduces a means of leveraging the Common Media Application Format (CMAF) as a standardized container format, combined with specific content encoding constraints and addressing mode constraints in Low latency HLS (LL-HLS) and Low latency DASH (LL-DASH), to provide a cross-format solution that maximizes edge cache efficiency and minimizes origin storage costs and client request rate.

INTRODUCTION

The year 2020 saw the release of updates to two HTTP Adaptive Streaming (HAS) standards targeting end-to-end latency in the 2s range: Low Latency DASH (LL-DASH) (1) and Low Latency HLS (LL-HLS) (2). Both these standards and modes of operation were developed independently, and while they can be deployed as separate streams in a content delivery system, there are performance and cost gains to be had for packagers, origins, CDNs, and players if both streaming formats can be served by a single set of media objects. The Common Media Application Format (CMAF) (3) is a standardized container format developed by MPEG for media delivery applications and formalized as ISO/IEC 23000-19. Specifically, CMAF uses the ISO Base Media File Format (ISOBMFF) container—with common encryption (CENC); support for H.264, HEVC, and other codecs; Web Video Text Tracks Format (WebVTT) and IMSC-1 captioning. This paper investigates the use of CMAF as a file container, combined with the byte-range addressing syntax within LL-HLS and additional encoding constraints, to solve the problem of delivering low latency video with high performance and scalability across the general internet.

CACHE EFFICIENCY

Caching is the primary means by which CDNs scale up HAS streams. The more content that can be cached, the better the performance and the lower the costs. If we imagine an LL-HLS stream with 4s segments and 1s parts, Figure 1 shows all the objects that will need to be cached at the edge within a 4 second window. Some are larger than others and we can highlight this difference by scaling them graphically such that the area is proportional to the size. Figure 1 shows that the video segments take up the largest amount of space.

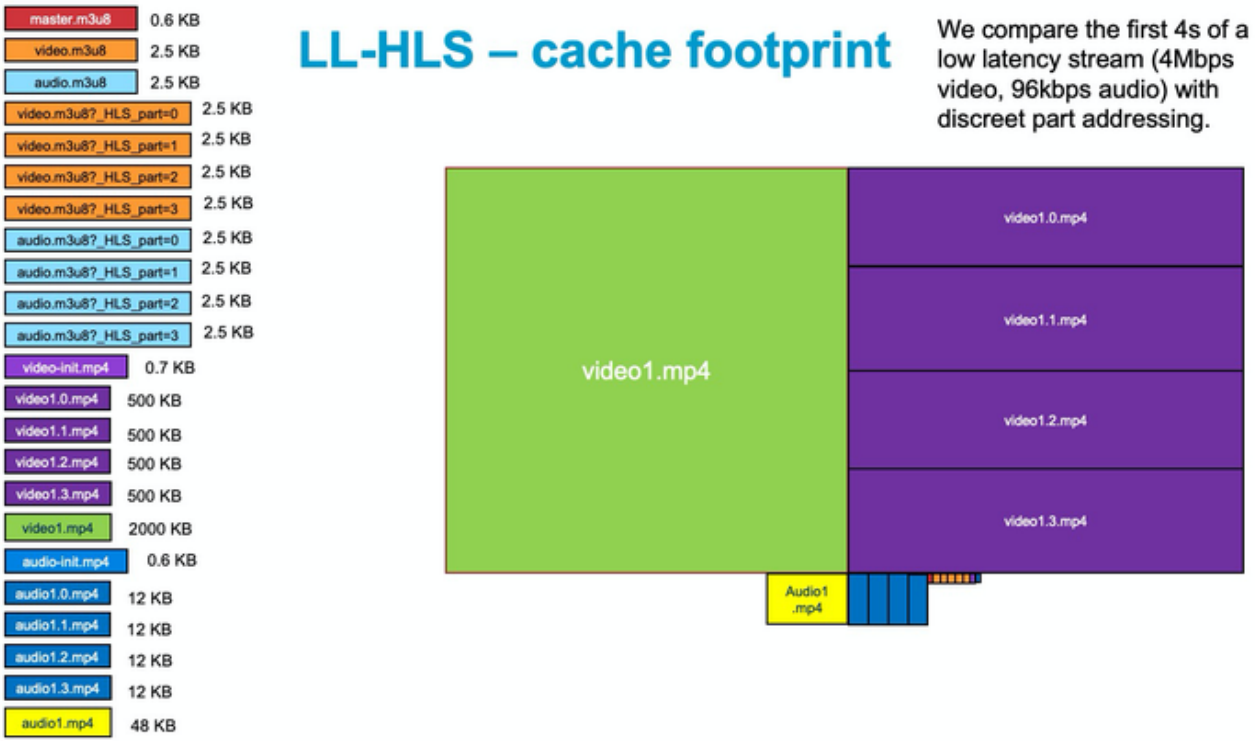

Figure 1 - Representation of the edge cache footprint of a LL-HLS stream

Notice there is duplication in content between the parts (purple), which are consumed by a low latency client playing at the live edge, and the contiguous media segments (green), which are consumed by standard latency clients, or low latency clients scrubbing behind the live edge. If we were to add in the DASH footprint, we would see that we have three silos of files, all holding the same media content, yet competing with one another for cache space. Our goal is to reduce these down to a single silo. This will lower origin storage by a factor of three and triple the cache efficiency for the CDN. This can be achieved using byte-range addressing.

BYTE-RANGE ADDRESSING

Within a LL-HLS media playlist, a partial segment (part) is described discreetly using a unique URL for every part. For example

```
#EXT-X-PART:DURATION=0.500,URI="segment1000-6.m4s"
```

This same part can alternatively be described using the BYTERANGE syntax

```
#EXT-X-PART:DURATION=0.500,URI="segment1000.m4s",BYTERANGE=251022@2079557
```

which specifies the length and offset at which a part is located within a media segment. For PRELOAD HINT parts, for which the last-byte-position is not yet known, only the start of the byte range is signalled:

```
#EXT-X-PRELOAD-HINT:TYPE=PART,URI="segment1000.m4s",BYTERANGE-START=2005479
```

A LL-HLS origin will block the response of a PRELOAD HINT entry until the part is ready, at which point it can be delivered at line speed. For an open-ended range request, the origin will return all remaining parts in the segment, enforcing the delivery guarantee for each one in turn. The consequence of this behaviour is that *a single request will return all the parts remaining in that segment*. Figure 2 illustrates how we can use this fact to derive a common workflow between LL-HLS and LL-DASH.

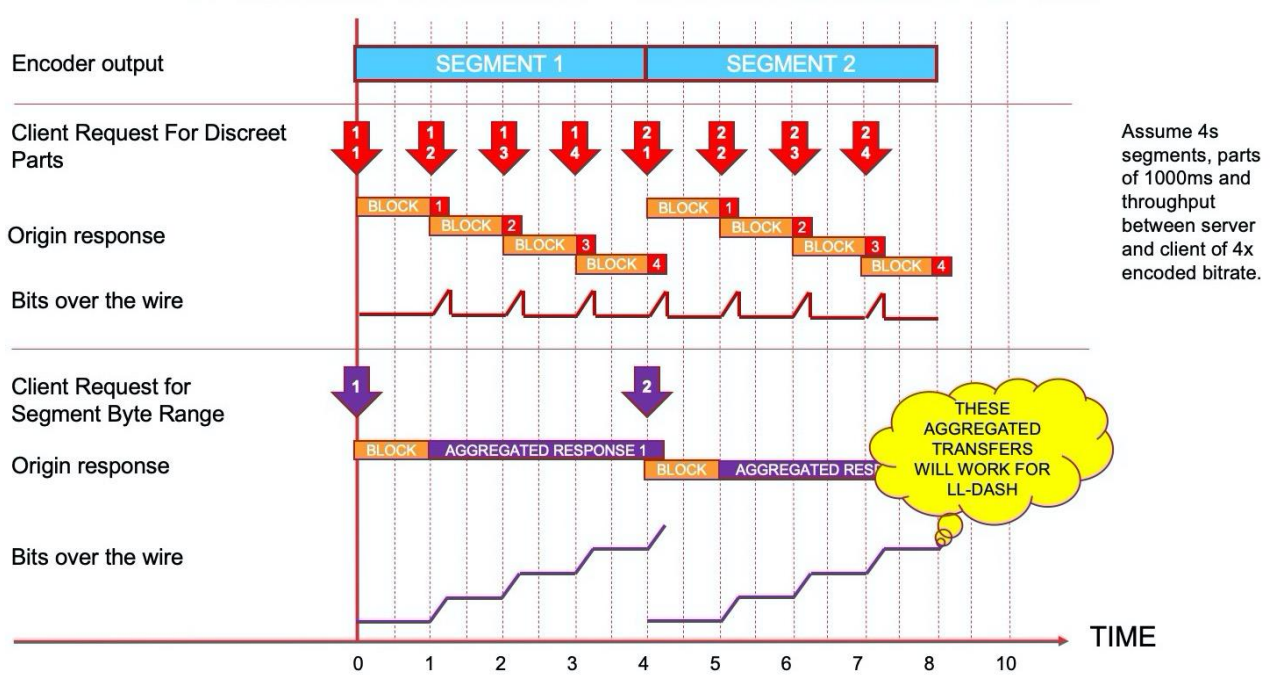

Figure 2 – A common workflow between LL-HLS and LL-DASH

The lower half of Figure 2 represents the workflow for a client using byte range addressing. At time 0, it makes an open-ended range request against segment 1. The origin blocks the response until the entirety of part 1 is available and then it begins an aggregated response back to the client. The term "aggregated" is used intentionally here. If this were http/1.1, it would be a chunked transfer response, however since LL-HLS mandates the use of http/2, and http/2 has framing, this is simply an aggregating http/2 response. Notice that bytes are injected into the byte-addressed response at the exact same time as they are released down the wire for the discreet-addressed parts. The two approaches are latency equivalent. Also, importantly -- the aggregating response in the byte-addressed case is exactly what an LL-DASH client is expecting. DASH clients do not have the constraint that the part (or "chunk" in their context) must be burst, but this bursting does not negatively affect them and in fact it helps considerably with their bandwidth estimation.

REQUEST RATE BENEFITS

Let's examine the start-up behaviour of a byte-range-addressed LL-HLS client. Consider a client faced with the following media playlist at start-up:

```
#EXTINF:8.000000,
606368.mp4
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="126172@0",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="100047@126172"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="140500@226219",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="104850@366719"
```

```
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="128784@471569",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="102325@600353"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="139024@702678",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="94084@841702"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="134672@935786",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="110055@1070458"
#EXT-X-PRELOAD-HINT:TYPE=PART,URI="606369.mp4",BYTERANGE-START=1180513
```

It could simply act as a discreet-addressed client would, which is to make eleven independent requests for each individual part. The last request would be an open one for the PRELOAD part. Another option, however, is that it could simply make a single open ended range request:

GET / 606369.mp4 range=0-

This single request would return all the parts, in the correct sequence, all at line speed and including all the future parts that will follow the PRELOAD part. This is exactly what the player needs and (for this ratio of part duration to segment duration) it can be accomplished with an eleven-fold decrease in media object requests.

THE PROBLEM WITH OPEN-ENDED RANGE REQUESTS

Open-ended range requests against aggregating sources present a dilemma for edge servers. Consider an edge server receiving a client request for range=0 against an object whose size is not yet known. Let's imagine the actual size is 1000B and that the first 100B have been received at the edge. Does the server:

1. Wait until it has received an EOF signal and return a 200-response code with content-length 1000? or
2. Immediately return the 100B in an open-ended 206-response and close the response once the 1000th byte is delivered?

The first behaviour is how most CDNs would behave today, yet the second is the behaviour we need for low latency streaming to work. Since both are valid use-cases, how can an edge server tell what behaviour to enact? Luckily, there is an RFC to the rescue! RFC8673 (4) says that the client should never make an open-ended range request if it is expecting an aggregated response from a fixed offset. It should instead send a request with a very large number as the last-byte-pos in the range request. 9007199254740991 has been proposed as a candidate (this equals Number.MAX_SAFE_INTEGER for 64-bit systems). This would signal the proxy-server (or origin) to begin a 206 response that starts at the requested offset and aggregates over time until the object is completely transferred. Note that this convention is only required when the start-byte-pos of the range request is non-zero. If the range being requested starts at zero, then a standard (non-range) GET request can be used, as the origin will naturally provide the aggregating response. We can now examine our hypothetical start-up situation for this playlist excerpt:

```
#EXTINF:8.000000,
606368.mp4
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="126172@0",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="100047@126172"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="140500@226219",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="104850@366719"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="128784@471569",INDEPENDENT=YES
```

```
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="102325@600353"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="139024@702678",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="94084@841702"
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="134672@935786",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.500000,URI="606369.mp4",BYTERANGE="110055@1070458"
#EXT-X-PRELOAD-HINT:TYPE=PART,URI="606369.mp4",BYTERANGE-START=1180513
```

The client may wish a two second starting buffer. Hence it would want to start with the independent segment located at byte offset 702678. The client would first ask for

> GET / 606369.mp4 HTTP/2
>
> Range: bytes=702678-9007199254740991

The server would respond with

> HTTP/2 206 Partial Content
>
> Content-Range: bytes 702678-9007199254740991/*

The origin responds by acknowledging the convention established by RFC8673 in the content-range header, along with signalling the content length as * since it is not yet known. It would then burst the bytes from 702678 to 1180513 and release the remainder as each part boundary became available.

STEADY STATE

If we were to examine all the requests crossing the wire after the player has started, they would look like this:

```
GET / 606369.mp4 HTTP/2 Range: bytes=702678-9007199254740991
GET / 606370.mp4 HTTP/2
GET / 606371.mp4 HTTP/2
GET / 606372.mp4 HTTP/2
...
```

Aside from the very first request, which uses the RFC8673 convention due to the non-zero starting offset, these are all standard GET requests without range headers. Surprisingly, we can make the general observation that an LL-HLS client using byte range addressing need only make one request per segment duration for each media type. This is a good performance gain for LL-HLS, which is otherwise a verbose format. Note that the client must still refresh its media playlists at the respective part duration interval, as those provide it with information on the changing state of the stream. The reduction in overall request rate is dependent on the ratio of part duration to segment duration. Table 1 shows the number of requests made per segment duration of wall clock interval for an LL-HLS client using either discrete or range-based part addressing.

Mode	Requests per segment duration					Gain
	Video playlist	Audio playlist	Video segment/Part	Audio segment/part	Total	% Reduction
4s segment 1s part Discreet	4	4	4	4	16	0
4s segment 1s part Range-based	4	4	1	1	10	37.5%
4s segment 0.5s part Discreet	8	8	8	8	32	0
4s segment 0.5s part Range-based	8	8	1	1	18	43%

Table 1 – Request rate reduction for differing ratios of segment-to-part duration

For the case of 4s segments and 1s parts, we see a 37.5% reduction in the overall number of requests every 4s. If the parts are reduced to 0.5s in duration, then that reduction rises to 43%. For a million connected clients, having 430,000 fewer requests every 4s is a material difference. Each request against a CDN has a cost -- in connections, compute, and power. For maximum distribution efficiency, we want to minimize our requests while maximizing the end user's quality of experience.

SEGMENT STRUCTURE

Early versions of the LL-HLS origins produced parts that were all independent (i.e., each one contained a keyframe) and then had contiguous segments with a single keyframe, as represented in Figure 3.

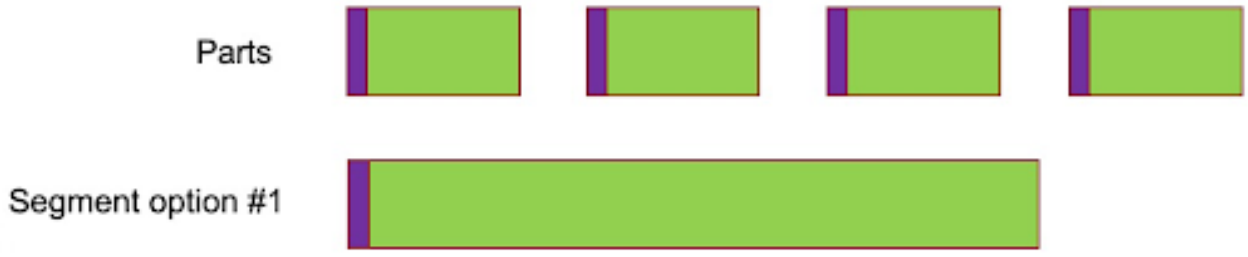

Figure 3 – Segment structure with a single GOP

The reason for this is encoding efficiency -- there is a small gain in encoding efficiency by moving to the longer GOP. However, this arrangement breaks the portability of having a single object be stored in cache from which we can serve both parts and segments. To achieve a unified cache, our segment must be a direct concatenation of our parts, as represented in Figure 4.

Figure 4 – Segment structure showing a concatenation of parts

The benefits to be gained by halving the cache footprint far outweigh the small encoding efficiency gains to be had by having two bit-different objects.

While standard CMAF segments are made of a single pair of moof+mdat atoms, low latency CMAF segments are a concatenation of CMAF chunks, with each chunk comprised of a moof and a mdat atom as described in Figure 5.

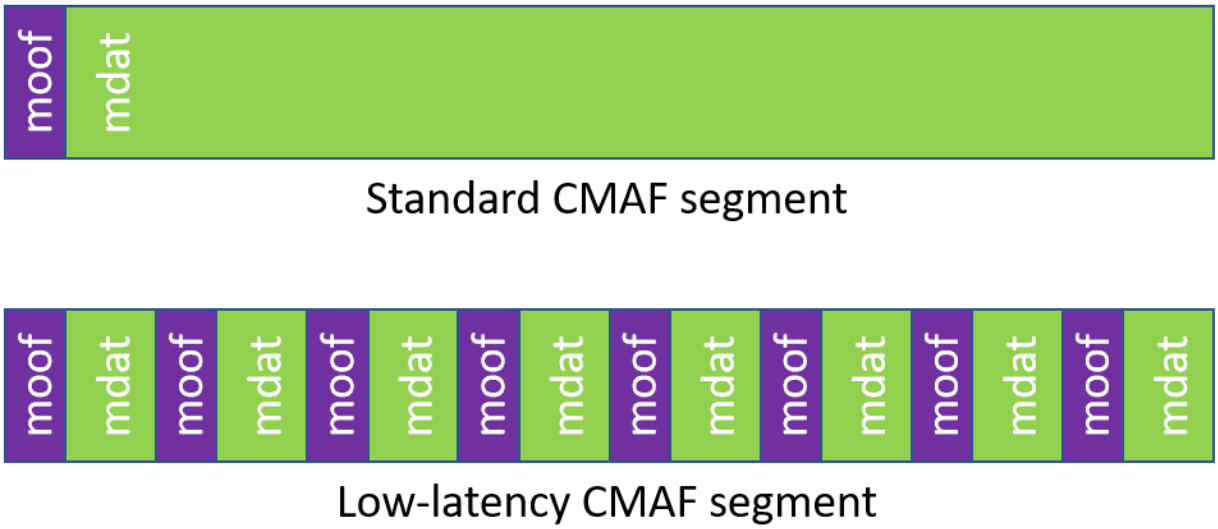

Figure 5 – Standard segment vs. low latency segment structure

Obviously, decoders must find a key frame to start decoding. The presence of a key frame is mandatory at the beginning of each segment but for the remainder of the segment there is no constraint on the GOP structure. Hence, the more key frames inside the segment, the more access points the decoder can find to start decoding earlier, which directly impacts overall latency. At the same time, key frames are also intra-encoded and consequently have a high bitrate cost, so encoders will have to decrease the visual quality of the video to satisfy a multi-key-frame structure. This leads to a trade-off between playout latency and video quality that must be made for each set of application requirements.

The chunking process implies constraints on the structure of the stream. A chunk should be self-contained i.e., decoding is possible without a dependency on the prior chunk. Thus, intra decoding refresh (IDR) frames are inserted at the beginning of each chunk. The shorter the chunks, the more IDR are inserted, which negatively impacts the coding performance. The same situation arises in traditional broadcast where IDR frames must be frequent enough to avoid a long “zapping” time. Another effect is related to the structure of the group of pictures (GOP). Due to interoperability issues, one may switch from open GOP to closed GOP. An open GOP starts with a simple I frame instead of an IDR which allows some frames from current GOP to be predicted using the I frame of next GOP resulting in smooth transitions on GOP edges. However, with closed GOP, the borders are IDR frames and so

the last frames of the current GOP cannot rely on the first IDR of the next GOP for prediction. This sharp cut of temporal dependencies may lead to a specific artifact, called pumping or intra flicker, if not well controlled by the encoder. Finally, the rate-control behavior can be impacted if low latency requires a shortening of the video buffer verifier (VBV). In general, the chunking process can negatively impact the coding performance and it is of paramount importance to rely upon an encoder specifically tuned for the low latency context.

Even though segment GOP structure has a significant impact on latency on the client side, there is more room for reducing end-to-end low latency on the encoder and ingest side. This opportunity occurs when the encoder/packager starts pushing the segment to the ingest point before it is completely encoded. HTTP chunked transfer is an excellent candidate for such use case. Chunked transfer encoding is an HTTP 1 mechanism allowing the progressive transmission of data through an HTTP connection using HTTP chunks as illustrated in Figure 6. For each segment, the packager opens an HTTP connection with the origin server. As soon as enough data is available to be sent, an HTTP chunk is created and sent without closing the request handler. The origin server answers with an HTTP '201 Continue' response, except for the last HTTP chunk of the segment which requires an HTTP '200 OK' response and the closure of the HTTP connection. Even though Chunked transfer Encoding is specific to HTTP/1.1, the identical aggregation capability is provided by the frames of an HTTP/2.0 response.

The chunked transfer will impact the latency of all transfer steps in the delivery flow since the whole pipeline will be chunked-based instead of segment-based: The encoder starts sending chunks before the end of the whole segment encoding, the CDN starts caching at the reception of the chunks and the client starts decoding/displaying as soon as the received data are decodable.

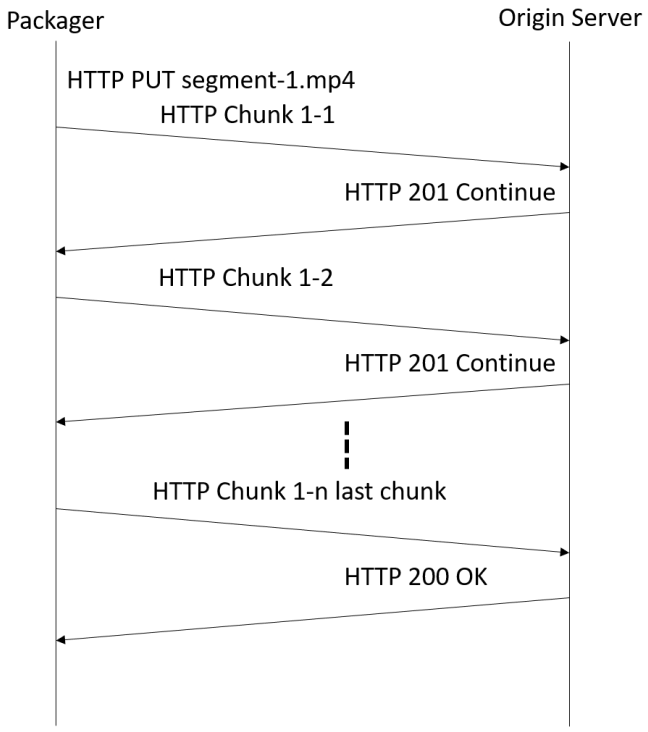

Figure 6 – HTTP Chunked Transfer Encoding

ORIGIN SERVER LOGIC

The origin server has two main roles to enable end-to-end low latency. The first one is to act as an HTTP Chunked Transfer Encoding proxy able to ingest and serve data as soon as it is available. This mechanism consists of storing the HTTP chunks received from the packager in such a way that it will be able to send them to the CDN when requested.
The second role is to manage the requests for playlists and media segments. The origin server is responsible of blocking the request to HLS playlists if needed. When a segment and a part number are included in the request, the response will be delayed until the playlist contains at least the specified part of the segment. Regarding the request to media segments, the origin server support open range requests already described in this document. More generally and independently of the request range, the origin server will immediately serve the bytes that are already available in his cache and the send the remaining data as soon as it is received while keeping the request handler opened if the whole payload has not been sent.

ESTIMATING THROUGHPUT

All HTTP adaptive streaming clients must use the download of the media segments to estimate the available throughput and thereby allow their ABR algorithm to switch-up. With discreet part delivery, this is done by measuring the bits received and dividing by the time taken to receive them. Since the objects are fully available at the server, the rate at which they are delivered is limited by the line speed and hence can be used to estimate how much throughput overhead is available. If the same logic is followed for an aggregating range-addressing response, it will provide an incorrect response. The bit numerator will be correct, but the denominator will include the time the origin was blocking delivery, as shown in Figure 7 on the left side.

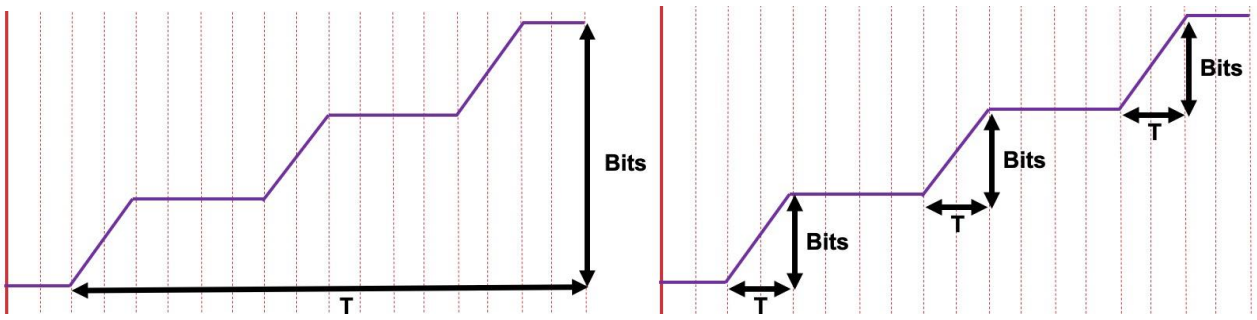

Figure 7 – Calculating throughput over an aggregating response

The player will keep dividing the total bits of the media segment by the delivery time, which is essentially the media playback time of the segment. This result will always return that the estimated throughput is equal to the encoded bitrate of the object -- a useless result that will be both inaccurate and prevent the player from ever switching up to a higher bitrate tier. What the player must do instead is only estimate throughout when the bits-across-the-wire are increasing, as shown on the right side in Figure 7. How can the player do this? Well, conveniently, the media playlist described the part boundaries as ranges and the origin and edge server are required to always burst parts. Therefore, if the player monitors its receive buffer it can mark the wall-clock time at which the part boundaries are received and hence calculate the throughput over the correct portion of the aggregation window.

REAL WORLD TEST AND VERIFICATION

To validate the concepts described in this paper, ATEME, a France-based provider of encoder and origin servers collaborated with Akamai Technologies, a US provider of CDN services. ATEME mounted an encoder and LL-HLS origin in an Amazon Web Services (AWS) instance in the state of Virginia in the United States. The Akamai CDN was placed on top of this and used to stream to a client located in San Francisco, California, as shown in Figure 8.

Figure 8 – test setup

The player was a test harness written in JavaScript, so that it could be run in a web browser. Figures 9 and 10 show screenshots of the player in action.

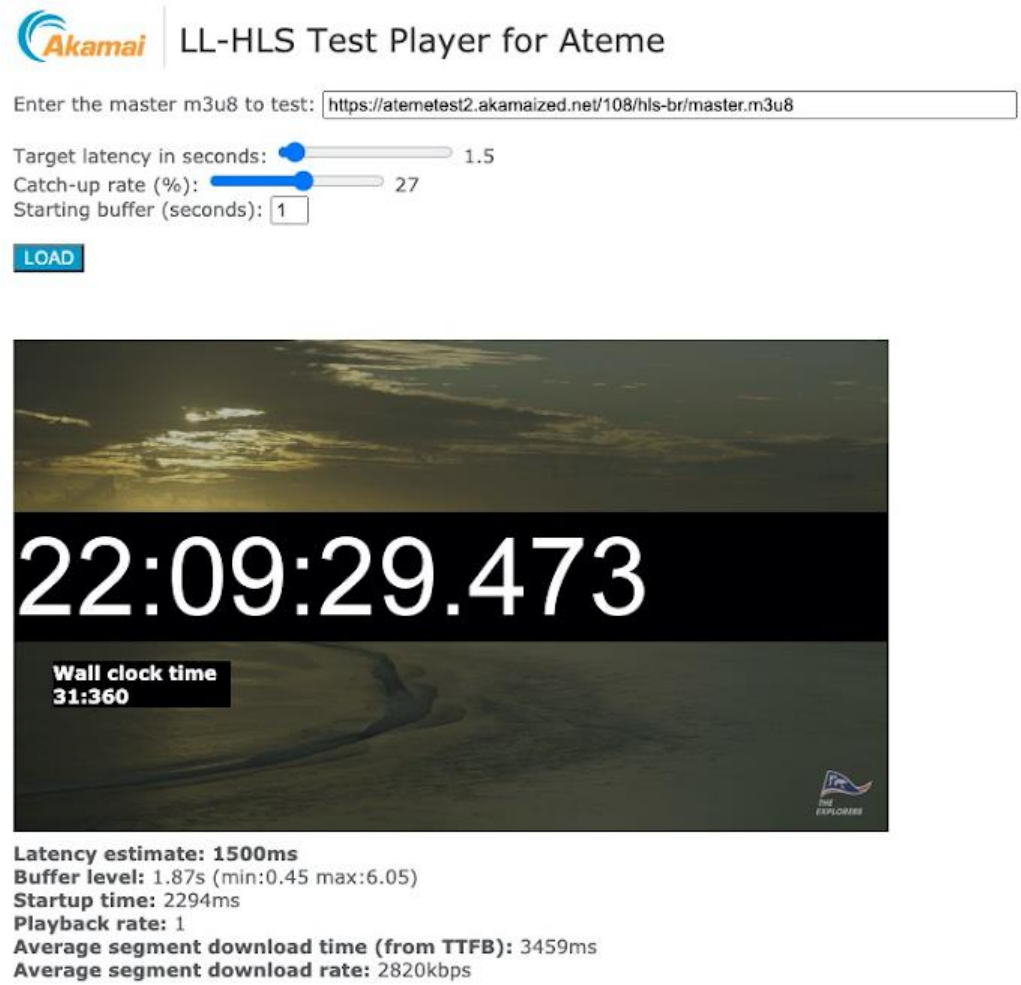

Figure 9 – Livestream in action

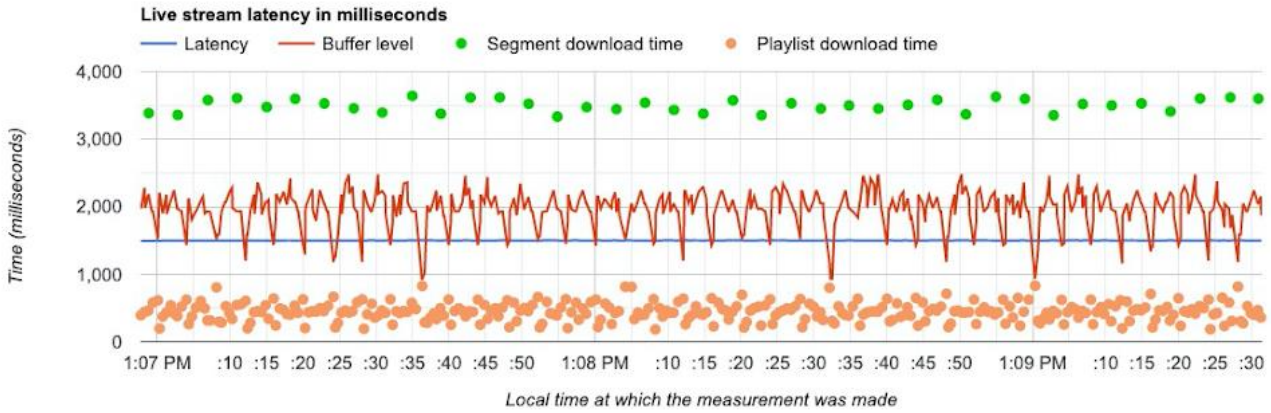

Figure 10 – Chart output of the live player

This stream contained 4s segments with 0.5s parts. It was operating at its target end-to-end latency of 1.5s. In the chart on the right the green dots show the completion of each media segment request. These all take just under 4s, as expected. The orange dots represent the media playlist updates, which are occurring every 500ms. By examining the video object requests in Figure 11,

Name	Method	Status	Protocol	Type	Size	Time	Connection ID
v1_1-401325243.m4s	GET	200	h2	fetch	1.2 MB	1.50 s	374583
v1_1-401325244.m4s	GET	200	h2	fetch	1.2 MB	3.04 s	374583
v1_1-401325245.m4s	GET	200	h2	fetch	1.3 MB	3.25 s	374583
v1_1-401325246.m4s	GET	200	h2	fetch	1.3 MB	3.33 s	374583
v1_1-401325247.m4s	GET	200	h2	fetch	1.2 MB	3.58 s	374583
v1_1-401325248.m4s	GET	200	h2	fetch	1.3 MB	3.50 s	374583
v1_1-401325249.m4s	GET	200	h2	fetch	1.2 MB	3.38 s	374583
v1_1-401325250.m4s	GET	200	h2	fetch	1.3 MB	3.52 s	374583
v1_1-401325251.m4s	GET	200	h2	fetch	1.2 MB	3.56 s	374583
v1_1-401325252.m4s	GET	200	h2	fetch	1.4 MB	3.59 s	374583
v1_1-401325253.m4s	GET	200	h2	fetch	1.2 MB	3.45 s	374583
v1_1-401325254.m4s	GET	200	h2	fetch	1.2 MB	3.61 s	374583
v1_1-401325255.m4s	GET	200	h2	fetch	1.2 MB	3.58 s	374583
v1_1-401325256.m4s	GET	200	h2	fetch	1.2 MB	3.52 s	374583
v1_1-401325257.m4s	GET	200	h2	fetch	1.3 MB	3.36 s	374583
v1_1-401325258.m4s	GET	200	h2	fetch	1.2 MB	3.62 s	374583

Figure 11 – Video object requests

we can see that the requests are only made against the segments and that each receives a 200 response from the edge server and takes just under 4s to complete. It is a curious fact that even though we are using range-based addressing with LL-HLS, under steady playback the client does not need to make any range-based requests! Figure 12 shows our three target players all playing together from the same origin and edge server. At the top is a LL-DASH player. Beneath it a LL-HLS player and below that a standard latency HLS player, represented by HLS.js.

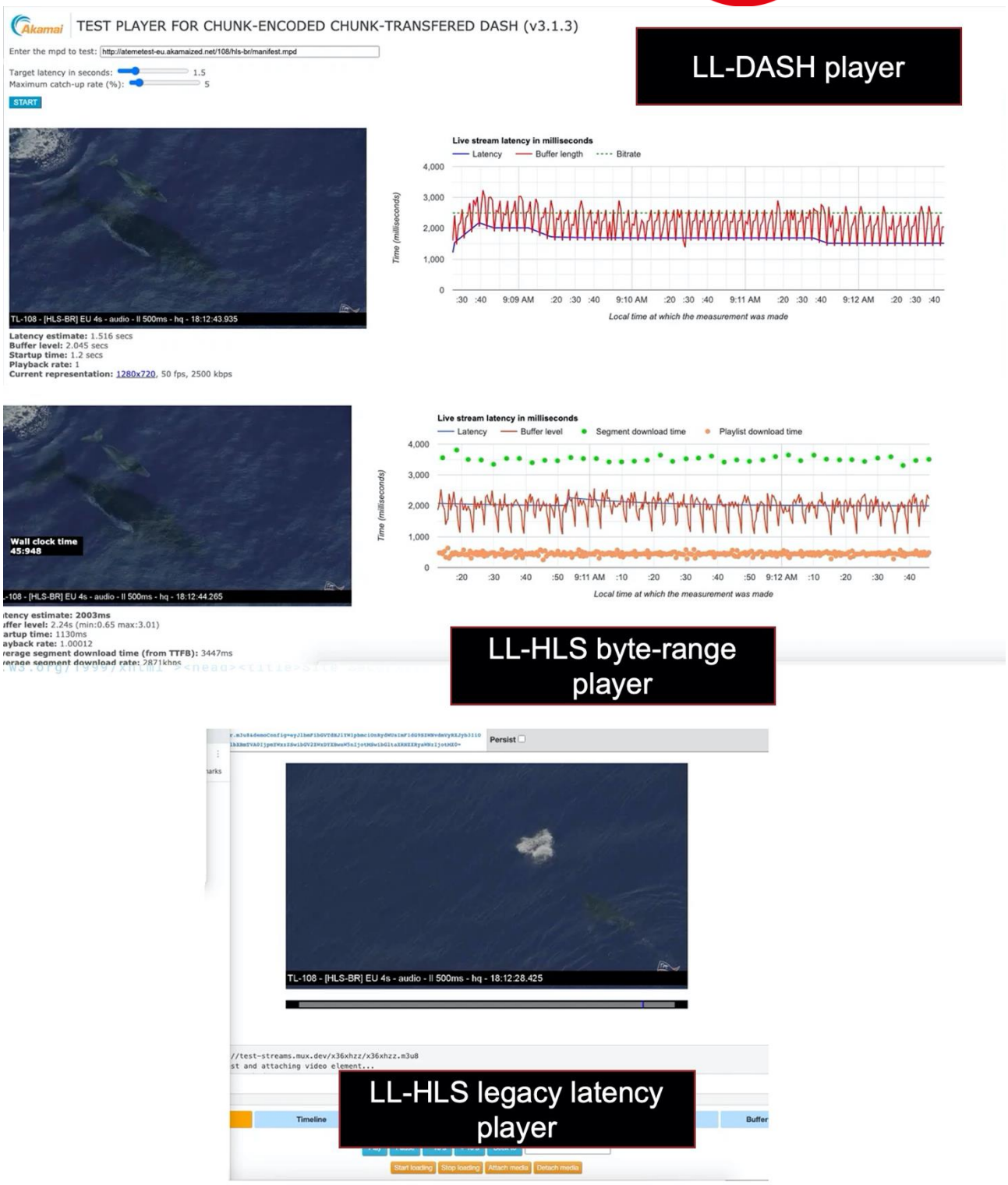

Figure 12 – Composite view of three different players all playing the same binary content

This standard latency player is playing the exact same stream as the LL-HLS player but is 12s behind since it ignores the parts and instead buffers three of the 4s segments before starting. Figure 13 represents the final validation of the approach described by this document. It shows the network panels of the three players arranged adjacent to one

another. You'll notice that each player is pulling the same media segment from the edge; 1-401326000.m4s - for example.

Figure 13 – Network request panels of three different players

CONCLUSION

The advent of CMAF-based packaging, chunked encoding and range-based addressing for LL-HLS enables multiple performance and efficiency benefits for distributors of cross-format low latency livestreams:

- Increased cache efficiency at origin and CDN distribution tiers, which increases performance and lowers operating costs.
- Decreased request rate from clients. We showed reductions of 30% to 40% for typical encoding configurations, which allows increased CDN-supported scale, lowers operating costs, and reduces the incidence between request errors.
- An LL-HLS client under steady-state playback does not need to make any range-requests against the origin even when range-based addressing is used in the playlist. This removes the CORS pre-flight requirements for browser-based clients, improving the latency with which playlists and segments can be returned.
- Interoperability among four types of clients: low latency HLS clients, standard latency HLS clients (also equivalent to LL-HLS clients scrubbing back from live), low latency DASH clients, and standard latency DASH clients
- If segments are generated with mid-segment independent parts and clients may start playback at these parts, then RFC 8673 is necessary to disambiguate the desired server response mode.

REFERENCES

1. ISO/IEC 23009-1:2019 Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats.
2. R. Pantos, W. May RFC 8261 HTTP Live Streaming — Internet Engineering Task Force. April 2021.
3. ISO/IEC 23000-19:2018 Information technology — Multimedia application format (MPEG-A) — Part 19: Common media application format (CMAF) for segmented media.
4. C. Pratt, D. Thakore, B. Stark RFC 8673 HTTP Random Access and Live Content — Internet Engineering Task Force. November 2019.